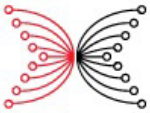


A Formal Specification of the Cardano Ledger

Deliverable SL-D5

Jared Corduan <jared.corduan@iohk.io>

Polina Vinogradova <polina.vinogradova@iohk.io>

Matthias Güdemann <matthias.gudemann@iohk.io>

Project: Shelley Ledger

Type: Deliverable
Due Date: 15th October 2019

Responsible team: Formal Methods Team
Editor: Jared Corduan, IOHK
Team Leader: Philipp Kant, IOHK

Version 1.0
8th October 2019

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

Contents

List of Figures

Change Log

Rev.	Date	Who	Team	What
1	2019/10/08	Jared Corduan, Polina Vinogradova and Matthias Güdemann	FM (IOHK)	Initial version (0.1).
2	2019/10/08	Kevin Hammond	FM (IOHK)	Added cover page.
3	2020/11/17	Jared Corduan	FM (IOHK)	Removed unused deliverable outline image, set version to 1.0, and changed the reward calculation so that η takes d into account.
4	2021/05/17	Jared Corduan	FM (IOHK)	Added Example Illustration.
5	2021/06/14	Jared Corduan	FM (IOHK)	Added an errata section, fixed a typo in leader check and in the LEDGERS rule index, and synced the reward calculation with the implementation
6	2021/06/17	Jared Corduan	FM (IOHK)	Allow the pool influence parameter a_0 to be zero, remove all mentions of deposit decay, sync varibale names with code.
7	2021/08/27	Jared Corduan	FM (IOHK)	Fixed definitions in the header-only validation properties. CHAIN transition did not need to use previous protocol parameters.
8	2021/10/08	Jared Corduan	FM (IOHK)	The function createRUpd should get the pool parameters from the go snapshot. The TICKN rule was missing from the dependency diagram.
9	2021/11/08	Jared Corduan	FM (IOHK)	Fixed typo in the description of variable length encodings.
10	2021/12/13	Jared Corduan	FM (IOHK)	Re-wrote the MIR transitions to be more compact.
11	2022/01/20	Jared Corduan	FM (IOHK)	Fixed error in counting new pools for deposits and an error in the POOLREAP rule.
12	2022/01/26	Jared Corduan	FM (IOHK)	Specify seed operation and seedToSlot.
13	2022/01/31	Jordan Millar, Jared Corduan	FM (IOHK)	Fixed prose regarding the hash used in the epoch nonce. Added an item to the errata regarding this same hash.
14	2022/08/25	Jared Corduan	FM (IOHK)	Specify the txid function in the Appendix. Warn about re-serializing.

A Formal Specification of the Cardano Ledger

Jared Corduan
jared.corduan@iohk.io

Polina Vinogradova
polina.vinogradova@iohk.io

Matthias Güdemann
matthias.gudemann@iohk.io

January 4, 2023

Abstract

This document provides a formal specification of the Cardano ledger for use in the upcoming Shelley implementation. It is intended to underpin a Haskell executable specification that will be the basis of the initial Shelley release, and represents a core design and quality assurance document. It will be used to define properties and tests, and to provide the basis for strong formal assurance using mathematical proof techniques. The document defines the rules for extending the ledger with transactions that will affect both UTxO and stake delegation. Key properties that have been identified include the preservation of balances, absence of double spend, stakepool registration, and reward splitting.

List of Contributors

Nicolás Arqueros, Dan Bornside, Nicholas Clarke, Duncan Coutts, Ruslan Dudin, Sebastien Guillemot, Kevin Hammond, Vincent Hanquez, Ru Horlick, Michael Hueschen, Christian Lindgren, Yun Lu, Philipp Kant, Jordan Millar, Jean-Christophe Mincke, Damian Nadales, Ashish Prajapati, Nicolas Di Prima, Andrew Westberg.

1 Introduction

This document is a formal specification of the functionality of the ledger on the blockchain. This includes the blockchain layer determining what is a valid block, but does not include any concurrency issues such as chain selection. The details of the background and the larger context for the design decisions formalized in this document are presented in [SL-D1].

In this document, we present the most important properties that 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 must be accounted for, and the total amount is unchanged by every transaction and epoch change. In particular, every coin is accounted for by exactly one of the following categories:

- Circulation (UTxO)
- Deposit pot
- Fee pot
- Reserves (monetary expansion)
- Rewards (account addresses)
- 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.

Rewards Reward calculation and distribution.

Updates The update mechanism for Shelley protocol parameters and software.

While the blockchain protocol is a reactive system that is driven by the arrival of blocks causing updates to the ledger, the formal description is a collection of rules that compose a static description of what a *valid ledger* is. A valid ledger state can only be reached by applying a sequence of inference rules and any valid ledger state is reachable by applying some sequence of these rules. The specifics of the semantics we use to define and apply the rules we present in this document are explained in detail in [FM-TR-2018-01] (this document will really help the reader's understanding of the contents of this specification).

The structure of the rules that 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 to take place). This property ensures that the specification is totally precise — no choice is left to the implementor and any two implementations must behave the same when it comes to deciding whether a block is valid.

2 Notation

The transition system is explained in [FM-TR-2018-01].

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. 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 $\{a \mid f a = b\}$.

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 $m\ a = b$. The $\emptyset$ symbol is also used to represent the empty map as well.

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 the 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 $b \in B$.

Option type An option type in type A is denoted as $A^? = A + \diamond$. The A case corresponds to the case when there is a value of type A and the $\diamond$ case corresponds to the case when there is no value.

:= We abuse the **:=** symbol here to mean propositional equality. In the style of semantics we use in this formal spec, definitional equality is not needed. It is meant to make the spec easier to read in the sense that each time we use it, we use a fresh variable as shorthand notation for an expression, e.g. we write

$$s := slot + \mathsf{StabilityWindow}$$

Then, in subsequent expressions, it is more convenient to write simply s. It is not meant to shadow variables, and if it does, there is likely a problem with the rules that must be addressed.

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

$$
\begin{aligned}
\mathit{set} \lhd \mathit{map} &= \{k \mapsto v \mid k \mapsto v \in \mathit{map},\ k \in \mathit{set}\} && \text{domain restriction}\\
\mathit{set} \lhd\!\!\!\!/\ \mathit{map} &= \{k \mapsto v \mid k \mapsto v \in \mathit{map},\ k \notin \mathit{set}\} && \text{domain exclusion}\\
\mathit{map} \rhd \mathit{set} &= \{k \mapsto v \mid k \mapsto v \in \mathit{map},\ v \in \mathit{set}\} && \text{range restriction}\\
\mathit{map} \rhd\!\!\!\!/\ \mathit{set} &= \{k \mapsto v \mid k \mapsto v \in \mathit{map},\ v \notin \mathit{set}\} && \text{range exclusion}\\
A \triangle B &= (A \setminus B) \cup (B \setminus A) && \text{symmetric difference}\\
M \underset{\rightarrow}{\cup} N &= (\operatorname{dom} N \lhd\!\!\!\!/\ M) \cup N && \text{union override right}\\
M \underset{\leftarrow}{\cup} N &= M \cup (\operatorname{dom} M \lhd\!\!\!\!/\ N) && \text{union override left}\\
M \cup_{+} N &= (M \triangle N) \cup \{k \mapsto v_1 + v_2 \mid k \mapsto v_1 \in M \wedge k \mapsto v_2 \in N\} && \text{union override plus}\\
& && \text{(for monoidal values)}
\end{aligned}
$$

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. Their exact implementation remains open to interpretation and we do not rely on any additional properties of public key cryptography that are not explicitly stated in this document. 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, signatures and multi-signature scripts. 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.

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. The type Ser denotes the serialized representation of a term of any serializable type.

Abstract types

$$
\begin{array}{rcll}
sk & \in & \mathsf{SKey} & \text{private signing key} \\
vk & \in & \mathsf{VKey} & \text{public verifying key} \\
hk & \in & \mathsf{KeyHash} & \text{hash of a key} \\
\sigma & \in & \mathsf{Sig} & \text{signature} \\
d & \in & \mathsf{Ser} & \text{data} \\
script & \in & \mathsf{Script} & \text{multi-signature script} \\
hs & \in & \mathsf{ScriptHash} & \text{hash of a script}
\end{array}
$$

Derived types

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

Abstract functions

$$
\begin{array}{rcll}
\mathsf{hashKey} & \in & \mathsf{VKey} \to \mathsf{KeyHash} & \text{hash a verification key} \\
\mathsf{verify} & \in & \mathbb{P}\ (\mathsf{VKey} \times \mathsf{Ser} \times \mathsf{Sig}) & \text{verification relation} \\
\mathsf{sign} & \in & \mathsf{SKey} \to \mathsf{Ser} \to \mathsf{Sig} & \text{signing function} \\
\mathsf{hashScript} & \in & \mathsf{Script} \to \mathsf{ScriptHash} & \text{hash a serialized script}
\end{array}
$$

Constraints

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

Notation for serialized and verified data

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

Figure 2: Cryptographic definitions

When we get to the blockchain layer validation, we will use key evolving signatures (KES).

This is another asymmetric key cryptographic scheme, also relying on the use of public and private key pairs. These signature schemes provide forward cryptographic security, meaning that a compromised key does not make it easier for an adversary to forge a signature that allegedly had been signed in the past. Figure 3 introduces the additional cryptographic abstractions needed for KES.

In KES, the public verification key stays constant, but the corresponding private key evolves incrementally. For this reason, KES signing keys are indexed by integers representing the step in the key's evolution. This evolution step parameter is also an additional parameter needed for the signing (denoted by $\mathsf{sign}_{\mathsf{ev}}$) and verification (denoted by $\mathsf{verify}_{\mathsf{ev}}$) functions.

Since the private key evolves incrementally in a KES scheme, the ledger rules require the pool operators to evolve their keys every time a certain number of slots have passed, as determined by the global constant SlotsPerKESPeriod.

Abstract types

$$\begin{array}{rcll} sk & \in & \mathbb{N} \to \mathsf{SKey}_{\mathsf{ev}} & \text{private signing keys} \\ vk & \in & \mathsf{VKey}_{\mathsf{ev}} & \text{public verifying key} \end{array}$$

Notation for evolved signing key

$$sk_n = sk\ n \qquad\qquad n\text{-th evolution of } sk$$

Derived types

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

Abstract functions

$$\begin{array}{rcll} \mathsf{verify}_{\mathsf{ev}} & \in & \mathbb{P}\ (\mathsf{VKey} \times \mathbb{N} \times \mathsf{Ser} \times \mathsf{Sig}) & \text{verification relation} \\ \mathsf{sign}_{\mathsf{ev}} & \in & (\mathbb{N} \to \mathsf{SKey}_{\mathsf{ev}}) \to \mathbb{N} \to \mathsf{Ser} \to \mathsf{Sig} & \text{signing function} \end{array}$$

Constraints

$$\begin{array}{c} \forall n \in \mathbb{N}, (sk_n, vk) \in \mathsf{KeyPair}_{\mathsf{ev}},\ d \in \mathsf{Ser},\ \sigma \in \mathsf{Sig} \cdot \\ \mathsf{sign}_{\mathsf{ev}}\ sk\ n\ d = \sigma \implies \mathsf{verify}_{\mathsf{ev}}\ vk\ n\ d\ \sigma \end{array}$$

Notation for verified KES data

$$\mathcal{V}^{\mathsf{KES}}_{vk}[\![d]\!]^n_\sigma = \mathsf{verify}_{\mathsf{ev}}\ vk\ n\ d\ \sigma \qquad \text{shorthand notation for } \mathsf{verify}_{\mathsf{ev}}$$

Figure 3: KES Cryptographic definitions

Figure 4 shows the types for multi-signature schemes. Multi-signatures effectively specify one or more combinations of cryptographic signatures which are considered valid. This is realized in a native way via a script-like DSL which allows for defining terms that can be evaluated. Multi-signature scripts is the only type of script (for any purpose, including output-locking) that exist in Shelley.

The terms form a tree like structure and are evaluated via the evalMultiSigScript function. The parameters are a script and a set of key hashes. The function returns True when the supplied key hashes are a valid combination for the script, otherwise it returns False. The following are the four constructors that make up the multisignature script scheme:

RequireSig : the signature of a key with a specific hash is required;

RequireAllOf : signatures of all of the keys that hash to the values specified in the given list are required;

RequireAnyOf : a single signature is required, by a key hashing to one of the given values in the list (this constructor is redundant and can be expressed using RequireMOf);

RequireMOf : m of the keys with the hashes specified in the list are required to sign

MultiSig Type

$$\begin{aligned} \mathsf{MSig} &\subseteq \mathsf{Script} \\ \\ msig \in \mathsf{MSig} &= \mathsf{RequireSig\ KeyHash} \\ &\uplus \mathsf{RequireAllOf\ [Script]} \\ &\uplus \mathsf{RequireAnyOf\ [Script]} \\ &\uplus \mathsf{RequireMOf\ } \mathbb{N}\ \mathsf{[Script]} \end{aligned}$$

Functions

$$\begin{aligned} &\mathsf{evalMultiSigScript} \in \mathsf{MSig} \to \mathbb{P}\ \mathsf{KeyHash} \to \mathsf{Bool} \\ &\mathsf{evalMultiSigScript}\ (\mathsf{RequireSig}\ hk)\ vhks = hk \in vhks \\ &\mathsf{evalMultiSigScript}\ (\mathsf{RequireAllOf}\ ts)\ vhks = \forall t \in ts : \mathsf{evalMultiSigScript}\ t\ vhks \\ &\mathsf{evalMultiSigScript}\ (\mathsf{RequireAnyOf}\ ts)\ vhks = \exists t \in ts : \mathsf{evalMultiSigScript}\ t\ vhks \\ &\mathsf{evalMultiSigScript}\ (\mathsf{RequireMOf}\ m\ ts)\ vhks = \\ &\qquad\qquad m \leq \Sigma\ ([\mathsf{if}\ (\mathsf{evalMultiSigScript}\ t\ vhks)\ \mathsf{then}\ 1\ \mathsf{else}\ 0 | t \leftarrow ts]) \end{aligned}$$

Figure 4: Multi-signature via Native Scripts

Figure 5 shows the cryptographic abstractions needed for Verifiable Random Functions (VRF). VRFs allow key-pair owners, $(sk, vk) \in \mathsf{KeyPair}$, to evaluate a pseudorandom function in a provable way given a randomness seed. Any party with access to the verification key, vk, the randomness seed, the proof and the generated randomness can indeed verify that the value is pseudorandom.

Abstract types

$$\begin{array}{rcll} \mathit{seed} & \in & \mathsf{Seed} & \text{seed for pseudo-random number generator} \\ \mathit{prf} & \in & \mathsf{Proof} & \text{VRF proof} \end{array}$$

Abstract functions (T an arbitrary type)

$$\begin{array}{rcll} \star & \in & \mathsf{Seed} \to \mathsf{Seed} \to \mathsf{Seed} & \text{binary seed operation} \\ \mathsf{vrf}_{\mathsf{T}} & \in & \mathsf{SKey} \to \mathsf{Seed} \to \mathsf{T} \times \mathsf{Proof} & \text{verifiable random function} \\ \mathsf{verifyVrf}_{\mathsf{T}} & \in & \mathsf{VKey} \to \mathsf{Seed} \to \mathsf{Proof} \times \mathsf{T} \to \mathsf{Bool} & \text{verify vrf proof} \end{array}$$

Derived Types

$$\mathsf{PoolDistr} = \mathsf{KeyHash}_{pool} \mapsto \Big([0,1] \times \mathsf{KeyHash}_{vrf}\Big) \qquad \text{stake pool distribution}$$

Constraints

$$\forall (sk, vk) \in \mathsf{KeyPair},\ \mathit{seed} \in \mathsf{Seed},\ \mathsf{verifyVrf}_T\ vk\ \mathit{seed}\ \ (\mathsf{vrf}_T\ sk\ \mathit{seed})$$

Constants

$$\begin{array}{ll} 0_{\mathit{seed}} \in \mathsf{Seed} & \text{neutral seed element} \\ \mathsf{Seed}_{\ell} \in \mathsf{Seed} & \text{leader seed constant} \\ \mathsf{Seed}_{\eta} \in \mathsf{Seed} & \text{nonce seed constant} \end{array}$$

Figure 5: VRF definitions

4 Addresses

Addresses are described in section 4.2 of the delegation design document [SL-D1]. The types needed for the addresses are defined in Figure 6. They all involve a credential, which is either a key or a multi-signature script. There are four types of UTxO addresses:

- Base addresses, $\mathsf{Addr}_{\mathsf{base}}$, containing the hash of a payment credential and the hash of a staking credential. Note that the payment credential hash is the hash of the key (or script) which has contol of the funds at this address, i.e. is able to witness spending them. The staking credential controls the delegation decision for the Ada at this address (i.e. it is used for rewards, staking, etc.). The staking credential must be a (registered) delegation credential (see Section 9 for a discussion of the delegation mechanism).
- Pointer addresses, $\mathsf{Addr}_{\mathsf{ptr}}$, containing the hash of a payment credential and a pointer to a stake credential registration certificate.
- Enterprise addresses, $\mathsf{Addr}_{\mathsf{enterprise}}$, containing only the hash of a payment credential (and which have no staking rights).
- Bootstrap addresses, $\mathsf{Addr}_{\mathsf{bootstrap}}$, corresponding to the addresses in Byron, behaving exactly like enterprise addresses with a key hash payment credential.

Where a credential is either a key or a multi-signature script. Together, these four address types make up the Addr type, which will be used in transaction outputs in Section 8. The notations $\mathsf{Credential}_{pay}$ and $\mathsf{Credential}_{stake}$ do not represent distinct types. The subscripts are annotations indicating how the credential is being used.

Section 5.5.2 of [SL-D1] provides the motivation behind enterprise addresses and explains why one might forgo staking rights. Bootstrap addresses are needed for the Byron-Shelley transition in order to accommodate having UTxO entries from the Byron era during the Shelley era.

There are also subtypes of the address types which correspond to the credential being either a key hash (the *vkey* subtype) or a script hash (the *script* subtype). So for example $\mathsf{Addr}^{script}_{base}$ is the type of base addresses which have a script hash as pay credential. This approach is used to facilitate expressing the restriction of the domain of certain functions to a specific credential type.

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

Finally, there is an account style address $\mathsf{Addr}_{\mathsf{rwd}}$ which contains the hash of a staking credential. These account addresses will only be used for receiving rewards from the proof of stake leader election. Appendix A of [SL-D1] explains this design choice. The mechanism for transferring rewards from these accounts will be explained in Section 8 and follows the approach outlined in the document [Zah18].

Note that, even though in the Cardano system, most of the accounting is UTxO-style, the reward addresses are a special case. Their use is restricted to only special cases (e.g. collecting rewards from them), outlined in the rules in Sections 8 and Section 11. For each staking credential, we use the function $\mathsf{addr}_{\mathsf{rwd}}$ to create the reward address corresponding to the credential, or to access an existing one if it already exists. Note that $\mathsf{addr}_{\mathsf{rwd}}$ uses the global constant NetworkId to attach a network ID to the given stake credential.

Base, pointer and enterprise addresses contain a payment credential which is either a key hash or a script hash. Base addresses contain a staking credential which is also either a key hash or a script hash.

Abstract types

$$
\begin{array}{rcll}
slot & \in & \mathsf{Slot} & \text{absolute slot} \\
ix & \in & \mathsf{Ix} & \text{index} \\
net & \in & \mathsf{Network} & \text{either Testnet or Mainnet}
\end{array}
$$

Derived types

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

Address subtypes

$$
\begin{array}{rcll}
addr^{vkey}_{base} & \in & \mathsf{Addr}^{vkey}_{base} & = \quad \mathsf{KeyHash} \lhd \mathsf{Addr}_{base} \\
addr^{script}_{base} & \in & \mathsf{Addr}^{vkey}_{base} & = \quad \mathsf{ScriptHash} \lhd \mathsf{Addr}_{base} \\
addr^{vkey}_{ptr} & \in & \mathsf{Addr}^{vkey}_{ptr} & = \quad \mathsf{KeyHash} \lhd \mathsf{Addr}_{ptr} \\
addr^{script}_{ptr} & \in & \mathsf{Addr}^{vkey}_{ptr} & = \quad \mathsf{ScriptHash} \lhd \mathsf{Addr}_{ptr} \\
addr^{vkey}_{enterprise} & \in & \mathsf{Addr}^{vkey}_{enterprise} & = \quad \mathsf{Addr}_{enterprise} \cap \mathsf{KeyHash} \\
addr^{script}_{enterprise} & \in & \mathsf{Addr}^{vkey}_{enterprise} & = \quad \mathsf{Addr}_{enterprise} \cap \mathsf{ScriptHash}
\end{array}
$$

$$
\begin{array}{rcll}
addr^{vkey} & \in & \mathsf{Addr}^{vkey} & = \quad \mathsf{Addr}^{vkey}_{base} \uplus \mathsf{Addr}^{vkey}_{ptr} \uplus \mathsf{Addr}^{vkey}_{enterprise} \uplus \mathsf{Addr}_{bootstrap} \\
addr^{script} & \in & \mathsf{Addr}^{script} & = \quad \mathsf{Addr}^{script}_{base} \uplus \mathsf{Addr}^{script}_{ptr} \uplus \mathsf{Addr}^{script}_{enterprise}
\end{array}
$$

$$
\begin{array}{rcll}
addr^{vkey}_{rwd} & \in & \mathsf{Addr}^{vkey}_{rwd} & = \quad \mathsf{Addr}_{rwd} \cap \mathsf{KeyHash} \\
addr^{script}_{rwd} & \in & \mathsf{Addr}^{script}_{rwd} & = \quad \mathsf{Addr}_{rwd} \cap \mathsf{ScriptHash}
\end{array}
$$

Accessor Functions

$$
\begin{array}{rcll}
\mathsf{paymentHK} & \in & \mathsf{Addr}^{\mathsf{vkey}} \to \mathsf{KeyHash}_{pay} & \text{hash of payment key from addr} \\
\mathsf{validatorHash} & \in & \mathsf{Addr}^{\mathsf{script}} \to \mathsf{ScriptHash} & \text{hash of validator script} \\
\mathsf{stakeCred_b} & \in & \mathsf{Addr}_{\mathsf{base}} \to \mathsf{Credential}_{stake} & \text{stake credential from base addr} \\
\mathsf{stakeCred_r} & \in & \mathsf{Addr}_{\mathsf{rwd}} \to \mathsf{Credential}_{stake} & \text{stake credential from reward addr} \\
\mathsf{addrPtr} & \in & \mathsf{Addr}_{\mathsf{ptr}} \to \mathsf{Ptr} & \text{pointer from pointer addr} \\
\mathsf{netId} & \in & \mathsf{Addr} \to \mathsf{Network} & \text{network Id from addr}
\end{array}
$$

Constructor Functions

$$
\mathsf{addr}_{\mathsf{rwd}} \in \mathsf{Credential}_{stake} \to \mathsf{Addr}_{\mathsf{rwd}} \qquad \text{construct a reward account, implicitly using NetworkId}
$$

Constraints

$$
hk_1 = hk_2 \iff \mathsf{addr}_{\mathsf{rwd}}\, hk_1 = \mathsf{addr}_{\mathsf{rwd}}\, hk_2 \quad (\mathsf{addr}_{\mathsf{rwd}} \text{ is injective})
$$

Figure 6: Definitions used in Addresses

5 Protocol Parameters

5.1 Updatable Protocol Parameters

The Shelley protocol parameters are listed in Figure 7. Some of the Shelley protocol parameters are common to the Byron era, specifically, the common ones are *a*, *b*, *maxTxSize*, and *maxHeaderSize* (see the document [BL-D1]).

The type Ppm represents the names of the protocol parameters, and $\mathsf{T}_{\mathsf{ppm}}$ is the type of the protocol parameter *ppm*. The type PParams is a finite map containing all the Shelley parameters, indexed by their names. We will explain the significance of each parameter as it comes up in the calculations used in transition rules. The type PParamsUpdate is similar to PParams, but is a partial mapping of the protocol parameters. It is used in the update system explained in Section 7.

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.

Some helper functions are defined in Figure 9. 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 $-_s$.

Lastly, there are two functions, epoch and firstSlot for converting between epochs and slots and one function kesPeriod for getting the cycle of a slot. Note that Slot is an abstract type, while the constants are integers. We use multiplication and division symbols on these distinct types without being explicit about the types and conversion.

5.2 Global Constants

In additon to the updatable protocol parameters defined in Section 5.1, there are ten parameters which cannot be changed by the update system in Section 7. We call these the global constants, as changing these values can only be done by updating the software, i.e. a soft or a hard fork. For the software update mechanism, see Section 13.

The constants SlotsPerEpoch and SlotsPerKESPeriod represent the number of slots in an epoch/KES period (for a brief explanation of a KES cryptography, see Section 3). The KES period is defined as a duration of slots which determines how often the KES private key must be evolved. The KES private key must be evolved once at the start of each new KES period. The duration of the KES period is determined by SlotsPerKESPeriod. Any given KES key can be evolved up to MaxKESEvo-many times before a new operational certificate must be issued. The constants StabilityWindow and RandomnessStabilisationWindow concern the chain stability. The maximum number of time a KES key can be evolved before a pool operator must create a new operational certificate is given by MaxKESEvo. **Note that if** MaxKESEvo **is changed, the KES signature format may have to change as well.**

The constant Quorum determines the quorum amount needed for votes on the protocol parameter updates and the application version updates.

The constant MaxMajorPV provides a mechanism for halting outdated nodes. Once the major component of the protocol version in the protocol parameters exceeds this value, every subsequent block is invalid. See Figures 74 and 75.

The constant MaxLovelaceSupply gives the total number of lovelace in the system, which is used in the reward calculation. It is always equal to the sum of the values in the UTxO, plus the sum of the values in the reward accounts, plus the deposit pot, plus the fee pot, plus the treasury and the reserves.

The constant ActiveSlotCoeff is the value f from the Praos paper [DGKR17].

Lastly, NetworkId determines what network, either mainnet or testnet, is expected. This value will also appear inside every address, and transactions containing addresses with an unexpected network ID are rejected.

Abstract types

p	$\in$	Ppm	protocol parameter
dur	$\in$	Duration	difference between slots
$epoch$	$\in$	Epoch	epoch
$kesPeriod$	$\in$	KESPeriod	KES period

Derived types

pp	$\in$ PParams	$=$	$\mathsf{Ppm} \to \mathsf{T}_{\mathsf{ppm}}$	protocol parameters
$ppup$	$\in$ PParamsUpdate	$=$	$\mathsf{Ppm} \mapsto \mathsf{T}_{\mathsf{ppm}}$	protocol parameter update
$coin$	$\in$ Coin	$=$	$\mathbb{Z}$	unit of value
pv	$\in$ ProtVer	$=$	$\mathbb{N} \times \mathbb{N}$	protocol version

Protocol Parameters

$a \mapsto \mathbb{Z}$	$\in$	PParams	min fee factor
$b \mapsto \mathbb{Z}$	$\in$	PParams	min fee constant
$maxBlockSize \mapsto \mathbb{N}$	$\in$	PParams	max block body size
$maxTxSize \mapsto \mathbb{N}$	$\in$	PParams	max transaction size
$maxHeaderSize \mapsto \mathbb{N}$	$\in$	PParams	max block header size
$poolDeposit \mapsto$ Coin	$\in$	PParams	stake pool deposit
$E_{max} \mapsto$ Epoch	$\in$	PParams	epoch bound on pool retirement
$n_{opt} \mapsto \mathbb{N}^{>0}$	$\in$	PParams	desired number of pools
$a_0 \mapsto [0, \infty)$	$\in$	PParams	pool influence
$\tau \mapsto [0, 1]$	$\in$	PParams	treasury expansion
$\rho \mapsto [0, 1]$	$\in$	PParams	monetary expansion
$d \mapsto \{0, 1/100, 2/100, \ldots, 1\}$	$\in$	PParams	decentralization parameter
$extraEntropy \mapsto$ Seed	$\in$	PParams	extra entropy
$pv \mapsto$ ProtVer	$\in$	PParams	protocol version
$minUTxOValue \mapsto$ Coin	$\in$	PParams	minimum allowed value of a new TxOut
$minPoolCost \mapsto$ Coin	$\in$	PParams	minimum allowed stake pool cost

Accessor Functions

a, b, maxBlockSize, maxTxSize, maxHeaderSize, keyDeposit, poolDeposit, emax, nopt, influence, tau, rho, d, extraEntropy, pv, minUTxOValue, minPoolCost

Abstract Functions

$(-_s) \in \mathsf{Slot} \to \mathsf{Slot} \to \mathsf{Duration}$ duration between slots

Figure 7: Definitions Used in Protocol Parameters

Global Constants

SlotsPerEpoch	$\in$	$\mathbb{N}$	- slots per epoch
SlotsPerKESPeriod	$\in$	$\mathbb{N}$	- slots per KES period
StabilityWindow	$\in$	Duration	- window size for chain growth guarantees, see in [DGKR17]
RandomnessStabilisationWindow	$\in$	Duration	- duration needed for epoch nonce stabilization
MaxKESEvo	$\in$	$\mathbb{N}$	- maximum KES key evolutions
Quorum	$\in$	$\mathbb{N}$	- quorum for update system votes
MaxMajorPV	$\in$	$\mathbb{N}$	- all blocks are invalid after this value
MaxLovelaceSupply	$\in$	Coin	- total lovelace in the system
ActiveSlotCoeff	$\in$	$(0, 1]$	- f in [DGKR17]
NetworkId	$\in$	Network	- the network, mainnet or testnet

Figure 8: Global Constants

Helper Functions

$$\mathsf{minfee} \in \mathsf{PParams} \to \mathsf{Tx} \to \mathsf{Coin}$$
$$\mathsf{minfee}\ pp\ tx = (\mathsf{a}\ pp) \cdot \mathsf{txSize}\ tx + (\mathsf{b}\ pp)$$

minimum fee

$$\mathsf{epoch} \in \mathsf{Slot} \to \mathsf{Epoch}$$
$$\mathsf{epoch}\ slot = slot\ /\ \mathsf{SlotsPerEpoch}$$

epoch of a slot

$$\mathsf{firstSlot} \in \mathsf{Epoch} \to \mathsf{Slot}$$
$$\mathsf{firstSlot}\ e = e\ \cdot\ \mathsf{SlotsPerEpoch}$$

first slot of an epoch

$$\mathsf{kesPeriod} \in \mathsf{Slot} \to \mathsf{KESPeriod}$$
$$\mathsf{kesPeriod}\ slot = slot\ /\ \mathsf{SlotsPerKESPeriod}$$

KES period of a slot

Figure 9: Helper functions for the Protocol Parameters

6 Transactions

Transactions are defined in Figure 10. A transaction body, TxBody, is made up of eight pieces:

- A set of transaction inputs. This 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 9.
- 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. For reward calculation rules, see Section 11.1, and for the rule for collecting rewards, see Section 8.1.
- An optional update proposals for the protocol parameters. The update system will be explained in Section 7.
- An optional metadata hash.

A transaction, Tx, consists of:

- The transaction body.
- A triple of:
  - A finite map from payment verification keys to signatures.
  - A finite map containing scripts as values, with their hashes as their indexes.
  - Optional metadata.

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 body.

Figure 11 shows the helper functions txinsVKey and txinsScript which partition the set of transaction inputs of the transaction into those that are locked with a private key and those that are locked via a script. It also defines validateScript, which validates the multisignature scripts.

Abstract types

$$
\begin{array}{rcll}
gkey & \in & \mathsf{VKey_G} & \text{genesis public keys} \\
gkh & \in & \mathsf{KeyHash_G} & \text{genesis key hash} \\
txid & \in & \mathsf{TxId} & \text{transaction id} \\
m & \in & \mathsf{Metadatum} & \text{metadatum} \\
mdh & \in & \mathsf{MetadataHash} & \text{hash of transaction metadata}
\end{array}
$$

Derived types

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

Derived types (update system)

$$
\begin{array}{rcll}
pup & \in & \mathsf{ProposedPPUpdates} = \mathsf{KeyHash_G} \mapsto \mathsf{PParamsUpdate} & \text{proposed updates} \\
up & \in & \mathsf{Update} = \mathsf{ProposedPPUpdates} \times \mathsf{Epoch} & \text{update proposal}
\end{array}
$$

Transaction Types

$$
\begin{array}{rcl}
txbody \in \mathsf{TxBody} & = & \mathbb{P}\ \mathsf{TxIn} \times (\mathsf{Ix} \mapsto \mathsf{TxOut}) \times \mathsf{DCert}^* \times \mathsf{Coin} \times \mathsf{Slot} \times \mathsf{Wdrl} \\
 & & \quad \times \mathsf{Update}^? \times \mathsf{MetadataHash}^? \\
wit \in \mathsf{TxWitness} & = & (\mathsf{VKey} \mapsto \mathsf{Sig}) \times (\mathsf{ScriptHash} \mapsto \mathsf{Script}) \\
tx \in \mathsf{Tx} & = & \mathsf{TxBody} \times \mathsf{TxWitness} \times \mathsf{Metadata}^?
\end{array}
$$

Accessor Functions

$$
\begin{array}{rcll}
\mathsf{txins} & \in & \mathsf{Tx} \to \mathbb{P}\ \mathsf{TxIn} & \text{transaction inputs} \\
\mathsf{txouts} & \in & \mathsf{Tx} \to (\mathsf{Ix} \mapsto \mathsf{TxOut}) & \text{transaction outputs} \\
\mathsf{txcerts} & \in & \mathsf{Tx} \to \mathsf{DCert}^* & \text{delegation certificates} \\
\mathsf{txfee} & \in & \mathsf{Tx} \to \mathsf{Coin} & \text{transaction fee} \\
\mathsf{txttl} & \in & \mathsf{Tx} \to \mathsf{Slot} & \text{time to live} \\
\mathsf{txwdrls} & \in & \mathsf{Tx} \to \mathsf{Wdrl} & \text{withdrawals} \\
\mathsf{txbody} & \in & \mathsf{Tx} \to \mathsf{TxBody} & \text{transaction body} \\
\mathsf{txwitsVKey} & \in & \mathsf{Tx} \to (\mathsf{VKey} \mapsto \mathsf{Sig}) & \text{VKey witnesses} \\
\mathsf{txwitsScript} & \in & \mathsf{Tx} \to (\mathsf{ScriptHash} \mapsto \mathsf{Script}) & \text{script witnesses} \\
\mathsf{txup} & \in & \mathsf{Tx} \to \mathsf{Update}^? & \text{protocol parameter update} \\
\mathsf{txMD} & \in & \mathsf{Tx} \to \mathsf{Metadata}^? & \text{metadata} \\
\mathsf{txMDhash} & \in & \mathsf{Tx} \to \mathsf{MetadataHash}^? & \text{metadata hash}
\end{array}
$$

Abstract Functions

$$
\begin{array}{rcll}
\mathsf{txid} & \in & \mathsf{TxBody} \to \mathsf{TxId} & \text{compute transaction id} \\
\mathsf{validateScript} & \in & \mathsf{Script} \to \mathsf{Tx} \to \mathsf{Bool} & \text{script interpreter} \\
\mathsf{hashMD} & \in & \mathsf{Metadata} \to \mathsf{MetadataHash} & \text{hash the metadata} \\
\mathsf{bootstrapAttrSize} & \in & \mathsf{Addr_{bootstrap}} \to \mathbb{N} & \text{bootstrap attribute size}
\end{array}
$$

Figure 10: Definitions used in the UTxO transition system

Helper Functions

$$\mathsf{txinsVKey} \in \mathbb{P}\ \mathsf{TxIn} \to \mathsf{UTxO} \to \mathbb{P}\ \mathsf{TxIn} \qquad \text{VKey Tx inputs}$$

$$\mathsf{txinsVKey}\ \mathit{txins}\ \mathit{utxo} = \mathit{txins} \cap \mathrm{dom}(\mathit{utxo} \rhd (\mathsf{Addr}^{\mathsf{vkey}} \times \mathit{Coin}))$$

$$\mathsf{txinsScript} \in \mathbb{P}\ \mathsf{TxIn} \to \mathsf{UTxO} \to \mathbb{P}\ \mathsf{TxIn} \qquad \text{Script Tx inputs}$$

$$\mathsf{txinsScript}\ \mathit{txins}\ \mathit{utxo} = \mathit{txins} \cap \mathrm{dom}(\mathit{utxo} \rhd (\mathsf{Addr}^{\mathsf{script}} \times \mathit{Coin}))$$

$$\mathsf{validateScript} \in \mathsf{Script} \to \mathsf{Tx} \to \mathsf{Bool} \qquad \text{validate script}$$

$$\mathsf{validateScript}\ \mathit{msig}\ \mathit{tx} = \begin{cases} \mathsf{evalMultiSigScript}\ \mathit{msig}\ \mathit{vhks} & \text{if } \mathit{msig} \in \mathsf{MSig} \\ \mathsf{False} & \text{otherwise} \end{cases}$$

$$\textbf{where}\ \ \mathit{vhks} := \{\mathsf{hashKey}\ \mathit{vk} | \mathit{vk} \in \mathrm{dom}(\mathsf{txwitsVKey}\ \mathit{tx})\}$$

Figure 11: Helper Functions for Transaction Inputs

7 Update Proposal Mechanism

The PPUP transition is responsible for the federated governance model in Shelley. The governance process includes a mechanism for core nodes to propose and vote on protocol parameter updates. In this chapter we outline rules for genesis keys *proposing* protocol parameter updates. For rules regarding the *adoption* of protocol parameter updates, see Section 11.7.

This chapter does not discuss authentication of update proposals. The signature for the keys in the proposal will be checked in the UTXOW transition, which checks all the necessary witnesses for a transaction, see Section8.3.

Genesis Key Delegations. The environment for the protocol parameter update transition contains the value *genDelegs*, which is a finite map indexed by genesis key hashes, and which maps to a pair consisting of a delegate key hash (corresponding to the cold key used for producing blocks) and a VRF key hash.

During the Byron era, the genesis nodes are all already delegated to some KeyHash, and these delegations are inherited through the Byron-Shelley transition (see Section 12.14). The VRF key hashes in this mapping will be new to the Shelley era.

The delegations mapping can be updated as described in Section 9, but there is no mechanism for them to un-delegate or for the keys to which they delegate to retire (unlike regular stake pools).

The types ProposedPPUpdates and Update were defined in Figure 10. The update proposal type Update is a pair of ProposedPPUpdates and Epoch. The epoch in the update specifies the epoch in which the proposal is valid. ProposedPPUpdates is a finite maps which is indexed by the hashes of the keys of entities proposing the given updates, $\mathsf{KeyHash}_{\mathsf{G}}$. We use the abstract type $\mathsf{KeyHash}_{\mathsf{G}}$ to represent hashes of genesis (public verification) keys, which have type $\mathsf{VKey}_{\mathsf{G}}$. Genesis keys are the keys belonging to the federated nodes running the Cardano system currently (also referred to as core nodes). The regular user verification keys are of a type VKey, distinct from the genesis key type, $\mathsf{VKey}_{\mathsf{G}}$. Similarly, the type hashes of these are distinct, KeyHash and $\mathsf{KeyHash}_{\mathsf{G}}$ respectively.

Currently, updates can only be proposed and voted on by the owners of the genesis keys. The process of decentralization will result in the core nodes gradually giving up some of their privileges and responsibilities to the network, eventually give them *all* up. The update proposal mechanism will not be decentralized in the Shelley era, however. For more on the decentralization process, see Section 12.3.

7.1 Protocol Parameter Update Proposals

The transition type PPUP is for proposing updates to protocol parameters, see Figure 12 (for the corresponding rules, see Figure 13). The signal for this transition is an optional update.

Protocol updates for the current epoch are only allowed up until $(2 \cdot \mathsf{StabilityWindow})$-many slots before the end of the epoch. The reason for this involves how we safely predict hard forks. Changing the protocol version can result in a hard fork, and we would like an entire stability period between when we know that a hard fork will necessarily happen and when the current epoch ends. Protocol updates can still be submitted during the last $(2 \cdot \mathsf{StabilityWindow})$-many slots of the epoch, but they must be marked for the following epoch.

The transition PPUP has three rules:

- PP-Update-Empty : No new updates were proposed, do nothing.
- PP-Update-Current : Some new updates *up* were proposed for the current epoch, and the current slot is not too far into the epoch. Add these to the existing proposals using a right override. That is, if a genesis key has previously submitted an update proposal, replace it with its new proposal in *pup*.

- PP-Update-Future : Some new updates *up* were proposed for the next epoch, and the current slot is near the end of the epoch. Add these to the existing future proposals using a right override. That is, if a genesis key has previously submitted a future update proposal, replace it with its new proposal in *pup*.

  The future update proposals will become update proposals on the next epoch, provided they contain no proposals for a protocol version which cannot follow from the current protocol version. See the NEWPP transition in Figure 43, and the function updatePpup from Figure 42.

This rule has the following predicate failures:

1. In the case that the epoch number in the signal is not appropriate for the slot in the current epoch, there is a *PPUpdateWrongEpoch* failure.
2. In the case of *pup* being non-empty, if the check $\mathrm{dom}\ pup \subseteq \mathrm{dom}\ genDelegs$ fails, there is a *NonGenesisUpdate* failure as only genesis keys can be used in the protocol parameter update.
3. If a protocol parameter update in *pup* contains a proposal for a protocol version which cannot follow from the current protocol version, there is a *PVCannotFollow* failure. Note that pvCanFollow is defined in Figure 12.

Derived types

$$\mathsf{GenesisDelegation} = \mathsf{KeyHash}_{\mathsf{G}} \mapsto (\mathsf{KeyHash} \times \mathsf{KeyHash}_{vrf}) \quad \text{genesis delegations}$$

Protocol Parameter Update environment

$$\mathsf{PPUpdateState} = \begin{pmatrix} pup & \in & \mathsf{ProposedPPUpdates} & \text{current proposals} \\ fpup & \in & \mathsf{ProposedPPUpdates} & \text{future proposals} \end{pmatrix}$$

$$\mathsf{PPUpdateEnv} = \begin{pmatrix} slot & \in & \mathsf{Slot} & \text{current slot} \\ pp & \in & \mathsf{PParams} & \text{protocol parameters} \\ genDelegs & \in & \mathsf{GenesisDelegation} & \text{genesis key delegations} \end{pmatrix}$$

Protocol Parameter Update transitions

$$\_ \vdash \_ \xrightarrow[\text{PPUP}]{} \_ \subseteq \mathbb{P}\ (\mathsf{PPUpdateEnv} \times \mathsf{PPUpdateState} \times \mathsf{Update}^{?} \times \mathsf{PPUpdateState})$$

Helper Functions

$$\begin{aligned} &\mathsf{pvCanFollow} \in \mathsf{ProtVer} \to \mathsf{ProtVer} \to \mathsf{Bool} \\ &\mathsf{pvCanFollow}\ (m,\ n)\ (m',\ n') = \\ &\quad (m+1,0) = (m',n') \vee (m,n+1) = (m',n') \end{aligned}$$

Figure 12: Protocol Parameter Update Transition System Types

$$\text{PP-Update-Empty}\ \frac{up = \diamond}{\begin{array}{r} slot \\ pp \\ genDelegs \end{array} \vdash \begin{pmatrix} pup_s \\ fpup_s \end{pmatrix} \xrightarrow[\text{PPUP}]{up} \begin{pmatrix} pup_s \\ fpup_s \end{pmatrix}} \tag{1}$$

$$\text{PP-Update-Current}\ \frac{\begin{array}{c} (pup,\ e) := up \qquad \operatorname{dom} pup \subseteq \operatorname{dom} genDelegs \\ \forall ps \in \operatorname{range} pup,\ pv \mapsto v \in ps \implies \mathsf{pvCanFollow}\ (\mathsf{pv}\ pp)\ v \\ slot < \mathsf{firstSlot}\ ((\mathsf{epoch}\ slot) + 1) - 2 \cdot \mathsf{StabilityWindow} \\ \mathsf{epoch}\ slot = e \end{array}}{\begin{array}{r} slot \\ pp \\ genDelegs \end{array} \vdash \begin{pmatrix} pup_s \\ fpup_s \end{pmatrix} \xrightarrow[\text{PPUP}]{up} \begin{pmatrix} \boldsymbol{pup_s \underset{\rightarrow}{\cup} pup} \\ fpup_s \end{pmatrix}} \tag{2}$$

$$\text{PP-Update-Future}\ \frac{\begin{array}{c} (pup,\ e) := up \qquad \operatorname{dom} pup \subseteq \operatorname{dom} genDelegs \\ \forall ps \in \operatorname{range} pup,\ pv \mapsto v \in ps \implies \mathsf{pvCanFollow}\ (\mathsf{pv}\ pp)\ v \\ slot \geq \mathsf{firstSlot}\ ((\mathsf{epoch}\ slot) + 1) - 2 \cdot \mathsf{StabilityWindow} \\ (\mathsf{epoch}\ slot) + 1 = e \end{array}}{\begin{array}{r} slot \\ pp \\ genDelegs \end{array} \vdash \begin{pmatrix} pup_s \\ fpup_s \end{pmatrix} \xrightarrow[\text{PPUP}]{up} \begin{pmatrix} pup_s \\ \boldsymbol{fpup_s \underset{\rightarrow}{\cup} pup} \end{pmatrix}} \tag{3}$$

Figure 13: Protocol Parameter Update Inference Rules

8 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 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), the rules in this section are defined 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.

8.1 UTxO Transitions

Figure 14 defines functions needed for the UTxO transition system. See Figure 10 for most of the definitions used in the 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 the total sum of all the reward 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 credentials. This calculation is a sum of all coin in the inputs of *tx*, reward withdrawals and stake credential deposit refunds. Some of the definitions used in this function will be defined in Section 8.2. In particular, keyRefunds is defined in Figure 17.
- 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 8.2. In particular, totalDeposits is defined in Figure 17.

For a transaction and a given ledger state, the preservation of value property holds 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 produced function takes the registered stake pools (*poolParams*) 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 9.

$$\mathsf{outs} \in \mathsf{TxBody} \to \mathsf{UTxO} \qquad \text{tx outputs as UTxO}$$
$$\mathsf{outs}\ tx = \{(\mathsf{txid}\ tx, ix) \mapsto txout \mid ix \mapsto txout \in \mathsf{txouts}\ tx\}$$

$$\mathsf{ubalance} \in \mathsf{UTxO} \to \mathsf{Coin} \qquad \text{UTxO balance}$$
$$\mathsf{ubalance}\ utxo = \sum_{(\_ \mapsto (\_,\ c)) \in utxo} c$$

$$\mathsf{wbalance} \in \mathsf{Wdrl} \to \mathsf{Coin} \qquad \text{withdrawal balance}$$
$$\mathsf{wbalance}\ ws = \sum_{(\_ \mapsto c) \in ws} c$$

$$\mathsf{consumed} \in \mathsf{PParams} \to \mathsf{UTxO} \to \mathsf{TxBody} \to \mathsf{Coin} \qquad \text{value consumed}$$
$$\mathsf{consumed}\ pp\ utxo\ tx =$$
$$\mathsf{ubalance}\ (\mathsf{txins}\ tx \lhd utxo) + \mathsf{wbalance}\ (\mathsf{txwdrls}\ tx) + \mathsf{keyRefunds}\ pp\ tx$$

$$\mathsf{produced} \in \mathsf{PParams} \to (\mathsf{KeyHash}_{pool} \mapsto \mathsf{PoolParam}) \to \mathsf{TxBody} \to \mathsf{Coin} \qquad \text{value produced}$$
$$\mathsf{produced}\ pp\ poolParams\ tx =$$
$$\mathsf{ubalance}\ (\mathsf{outs}\ tx) + \mathsf{txfee}\ tx + \mathsf{totalDeposits}\ pp\ poolParams\ (\mathsf{txcerts}\ tx)$$

Figure 14: Functions used in UTxO rules

The types for the UTxO transition are given in Figure 15. The environment, UTxOEnv, consists of:

- The current slot.
- The protocol parameters.
- The registered stake pools (also explained in Section 9, Figure 21).
- The genesis key delegation mapping.

The current slot and registrations are needed for the refund calculations described in Section 8.2.

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

- The current UTxO.
- The deposit pot.
- The fee pot.
- Proposed updates (see Section 7).

The signal for the UTxO transition is a transaction.

UTxO environment

$$\mathsf{UTxOEnv} = \left(\begin{array}{rcll} \mathit{slot} & \in & \mathsf{Slot} & \text{current slot} \\ \mathit{pp} & \in & \mathsf{PParams} & \text{protocol parameters} \\ \mathit{poolParams} & \in & \mathsf{KeyHash}_{\mathit{pool}} \mapsto \mathsf{PoolParam} & \text{stake pools} \\ \mathit{genDelegs} & \in & \mathsf{GenesisDelegation} & \text{genesis key delegations} \end{array}\right)$$

UTxO States

$$\mathsf{UTxOState} = \left(\begin{array}{rcll} \mathit{utxo} & \in & \mathsf{UTxO} & \mathsf{UTxO} \\ \mathit{deposited} & \in & \mathsf{Coin} & \text{deposits pot} \\ \mathit{fees} & \in & \mathsf{Coin} & \text{fee pot} \\ \mathit{ppup} & \in & \mathsf{PPUpdateState} & \text{proposed updates} \end{array}\right)$$

UTxO transitions

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

Figure 15: UTxO transition-system types

The UTxO transition system is given in Figure 16. Rule 4 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. We leave open the future possibility that transactions with larger fees can be prioritized.
- 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 PPUP transition is successful.
- The coin value of each new output must be at least as large as the minimum value specified by the protocol parameter *minUTxOValue*.
- The transaction size must be below the allowed maximum. Note that there is an implicit max transaction size given by the max block size, and that if we wished to allow a transaction to be as large as will fit in a block, this check would not be needed. Being able to limit the size below that of the block, however, gives us some control over how transactions will be packed into the blocks.

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 the transaction body *txb*.
  - Add all the *outputs* of *tx* to the UTxO, associated with the txid of the transaction body *txb*
- Add all new deposits to the deposit pot and subtract all deposit refunds.
- Add the transaction fee to the fee pot.
- Update the current update proposals.

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 16.

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 for 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 8.3, assures that the ledger state is updated correctly.

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

The UTXO rule has ten predicate failures:

$$txb := \mathsf{txbody}\ tx \qquad \mathsf{txttl}\ txb \geq slot$$
$$\mathsf{txins}\ txb \neq \emptyset \qquad \mathsf{minfee}\ pp\ tx \leq \mathsf{txfee}\ txb \qquad \mathsf{txins}\ txb \subseteq \mathrm{dom}\ utxo$$
$$\mathsf{consumed}\ pp\ utxo\ txb = \mathsf{produced}\ pp\ poolParams\ txb$$

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

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

$$\text{UTxO-inductive}\ \frac{\begin{matrix} refunded := \mathsf{keyRefunds}\ pp\ txb \\ depositChange := \mathsf{totalDeposits}\ pp\ poolParams\ (\mathsf{txcerts}\ txb) - refunded \end{matrix}}{\begin{matrix} slot \\ pp \\ poolParams \\ genDelegs \end{matrix} \vdash \begin{pmatrix} utxo \\ deposited \\ fees \\ ppup \end{pmatrix} \xrightarrow[\text{UTXO}]{tx} \begin{pmatrix} (\mathbf{txins}\ txb \lhd utxo) \cup \mathbf{outs}\ txb \\ deposited + depositChange \\ fees + \mathbf{txfee}\ txb \\ ppup' \end{pmatrix}} \tag{4}$$

Figure 16: UTxO inference rules

- In the case of any input not being valid, there is a *BadInput* failure.
- In the case of the current slot being greater than the time-to-live of the current transaction, there is an *Expired* failure.
- In the case that the transaction is too large, there is a *MaxTxSize* failure.
- In the case of an empty input set, there is a *InputSetEmpty* failure, in order to prevent replay attacks.
- If the fees are smaller than the minimal transaction fees, there is a *FeeTooSmall* failure.
- If the transaction does not correctly conserve the balance, there is a *ValueNotConserved* failure.
- If the transaction creates any outputs with the wrong network ID, there is a *WrongNetwork* failure.
- If the transaction contains any withdrawals with the wrong network ID, there is a *WrongNetworkWithdrawal* failure.
- If the transaction creates an output below the allowed minimum value, there is a *OutputTooSmall* failure.
- If the transaction creates any boostrap outputs whose attributes have size bigger than 64, there is a *OutputBootAddrAttrsTooBig* failure.

8.2 Deposits and Refunds

Deposits are described in appendix B.2 of the delegation design document [SL-D1]. These deposit functions were used above in the UTxO transition, 8.1. Deposits are used for stake credential registration and pool registration certificates, which will be explained in Section 9. In particular, the function cwitness , which gets the certificate witness from a certificate, will be defined later. Figure 17 defines the deposit and refund functions.

- The function totalDeposits returns the total deposits that have to be made by a transaction. This calculation uses two protocol parameters, namely the key deposit value and the pool deposit value. Note that 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 keyRefunds, calculates the total amount of returned deposits from stake key deregistration certificates.

  Note that keyRefunds uses the *current* protocol parameters. This means that any deposits made prior to a change in the deposit values will be refunded with the current value, not the one originally paid.

  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.

$$\mathsf{totalDeposits} \in \mathsf{PParams} \to (\mathsf{KeyHash}_{pool} \mapsto \mathsf{PoolParam}) \to \mathsf{DCert}^* \to \mathsf{Coin}$$ total deposits for a tx

$$\mathsf{totalDeposits}\ pp\ poolParams\ certs = (\mathsf{keyDeposit}\ pp) \cdot |certs \cap \mathsf{DCert}_{\mathsf{regkey}}| + (\mathsf{poolDeposit}\ pp) \cdot |\{\mathsf{cwitness}\ c \mid c \in newPools\}|$$

where

$$newPools = \{c \mid c \in certs \cap \mathsf{DCert}_{\mathsf{regpool}},\ \mathsf{cwitness}\ c \notin poolParams\}$$

$$\mathsf{keyRefunds} \in \mathsf{PParams} \to \mathsf{TxBody} \to \mathsf{Coin}$$ key refunds for a tx

$$\mathsf{keyRefunds}\ pp\ tx = (\mathsf{keyDeposit}\ pp) \cdot |\mathsf{txcerts}\ tx \cap \mathsf{DCert}_{\mathsf{deregkey}}|$$

Figure 17: Functions used in Deposits - Refunds

8.3 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, a transaction which includes a delegation certificate and a reward withdrawal corresponding to the same stake credential still only includes one signature.

Figure 18 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 credentials for reward withdrawals
- stake credentials for delegation certificates (except $\mathsf{DCert}_{\mathsf{mir}}$ and $\mathsf{DCert}_{\mathsf{regkey}}$)
- delegates of the genesis keys for any protocol parameter updates
- stake credentials for the pool owners in a pool registration certificate

The UTxOW transition system adds witnessing to the previous UTxO transition system. Figure 19 defines the type for this transition.

Figure 20 defines UTxOW transition. It has six 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 a subset of the set of needed (hashes of) verification keys.
- Every multisignature script is valid.
- The set of scripts given by the transaction is equal to the set of required scripts.
- Any instantaneous reward certificates have quorum agreement from the genesis nodes delegates.
- Either the transaction metadata hash and the transaction metadata are both absent, or the hash present in the body is actually equal to the hash of the metadata.

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

The UTXOW rule has eight predicate failures:

- In the case of an incorrect witness, there is a *InvalidWitnesses* failure.
- In the case of a missing witness, there is a *MissingVKeyWitnesses* failure.
- In the case of missing scripts, there is a *MissingScriptWitnesses* failure.
- In the case of an invalid script, there is a *ScriptWitnessNotValidating* failure.
- In the case of a lack of quorum on an instantaneous reward certificate, there is a *MIRInsufficientGenesisSigs* failure.

$$\mathsf{propWits} \in \mathsf{Update} \to \mathsf{GenesisDelegation} \to \mathbb{P}\ \mathsf{KeyHash} \qquad \text{hashkeys for proposals}$$

$$\mathsf{propWits}\ (pup,\ \_)\ genDelegs =$$
$$\{kh \mid gkh \mapsto (kh,\ \_) \in (\mathrm{dom}\ pup \lhd genDelegs)\}$$

$$\mathsf{certWitsNeeded}\ \mathsf{Tx} \to \mathbb{P}\ \mathsf{Credential} \qquad \text{certificates with witnesses}$$

$$\mathsf{certWitsNeeded}\ tx =$$
$$\bigcup \{\mathsf{cwitness}\ c \mid c \in \mathsf{txcerts}\ tx \setminus (\mathsf{DCert}_{\mathsf{regkey}} \cup \mathsf{DCert}_{\mathsf{mir}})\}$$

$$\mathsf{witsVKeyNeeded} \in \mathsf{UTxO} \to \mathsf{Tx} \to (\mathsf{KeyHash}_{\mathsf{G}} \mapsto \mathsf{VKey}) \to \mathbb{P}\ \mathsf{KeyHash} \qquad \text{required key hashes}$$

$$\begin{aligned}
&\mathsf{witsVKeyNeeded}\ utxo\ tx\ genDelegs = \\
&\quad \{\mathsf{paymentHK}\ a \mid i \mapsto (a, _) \in utxo,\ i \in \mathsf{txinsVKey}\ tx\} \\
&\cup\ \{\mathsf{stakeCred_r}\ a \mid a \mapsto _ \in \mathsf{Addr}^{\mathsf{vkey}}_{\mathsf{rwd}} \lhd \mathsf{txwdrls}\ tx\} \\
&\cup\ (\mathsf{Addr}^{\mathsf{vkey}} \cap \mathsf{certWitsNeeded}\ tx) \\
&\cup\ \mathsf{propWits}\ (\mathsf{txup}\ tx)\ genDelegs \\
&\cup \bigcup_{\substack{c \in \mathsf{txcerts}\ tx \\ c \in \mathsf{DCert}_{\mathsf{regpool}}}} \mathsf{poolOwners}\ c
\end{aligned}$$

$$\mathsf{scriptsNeeded} \in \mathsf{UTxO} \to \mathsf{Tx} \to \mathbb{P}\ \mathsf{ScriptHash} \qquad \text{required script hashes}$$

$$\begin{aligned}
&\mathsf{scriptsNeeded}\ utxo\ tx = \\
&\quad \{\mathsf{validatorHash}\ a \mid i \mapsto (a, _) \in utxo, \\
&\qquad i \in \mathsf{txinsScript}\ (\mathsf{txins}\ tx)\ utxo\} \\
&\cup\ \{\mathsf{stakeCred_r}\ a \mid a \in \mathrm{dom}(\mathsf{Addr}^{\mathsf{script}}_{\mathsf{rwd}} \lhd \mathsf{txwdrls}\ tx)\} \\
&\cup\ (\mathsf{Addr}^{\mathsf{script}} \cap \mathsf{certWitsNeeded}\ tx)
\end{aligned}$$

Figure 18: Functions used in witness rule

- In the case the transaction contains metadata, but the transaction body does not contain a metadata hash, there is a *MissingTxBodyMetadataHash* failure.
- In the case that the transaction body contains a metadata hash, but there is no metadata outside the body, there is a *MissingTxMetadata* failure.
- In the case that the metadata hash in the transaction body is not equal to the hash of the metadata outside the body, there is a *ConflictingMetadataHash* failure.

UTxO with witness transitions

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

Figure 19: UTxO with witness transition-system types

$$(utxo, \_, \_, \_) := utxoSt$$
$$witsKeyHashes := \{\mathsf{hashKey}\ vk | vk \in \mathrm{dom}(\mathsf{txwitsVKey}\ tx)\}$$

$$\forall hs \mapsto validator \in \mathsf{txwitsScript}\ tx,$$
$$\mathsf{hashScript}\ validator = hs \wedge \mathsf{validateScript}\ validator\ tx$$

$$\mathsf{scriptsNeeded}\ utxo\ tx = \mathrm{dom}(\mathsf{txwitsScript}\ tx)$$

$$txbodyHash := \mathsf{hash}\ (\mathsf{txbody}\ tx)$$
$$\forall vk \mapsto \sigma \in \mathsf{txwitsVKey}\ tx, \mathcal{V}_{vk}[\![txbodyHash]\!]_\sigma$$
$$\mathsf{witsVKeyNeeded}\ utxo\ tx\ genDelegs \subseteq witsKeyHashes$$

$$genSig := \{genDelegate \mid (genDelegate, \_) \in \mathrm{range}\ genDelegs\} \cap witsKeyHashes$$
$$\{c \in \mathsf{txcerts}\ tx \cap \mathsf{DCert_{mir}}\} \neq \emptyset \implies |genSig| \geq \mathsf{Quorum}$$

$$mdh := \mathsf{txMDhash}\ tx \qquad md := \mathsf{txMD}\ tx$$
$$(mdh = \diamond \wedge md = \diamond) \vee (mdh = \mathsf{hashMD}\ md)$$

$$\text{UTxO-wit}\ \frac{\begin{array}{r} slot \\ pp \\ poolParams \\ genDelegs \end{array} \vdash utxoSt \xrightarrow[\text{UTXO}]{tx} utxoSt'}{\begin{array}{r} slot \\ pp \\ poolParams \\ genDelegs \end{array} \vdash utxoSt \xrightarrow[\text{UTXOW}]{tx} \boldsymbol{utxoSt'}} \quad (5)$$

Figure 20: UTxO with witnesses inference rules

9 Delegation

We briefly describe the motivation and context for delegation. The full context is contained in [SL-D1].

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

Stake pool retirement is handled a bit differently than stake deregistration. Stake credentials 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 credentials, the delegation map from registered stake credentials to stake pools, pointers associated with stake credentials, 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 credentials to rewards.

Finally, there are two types of delegation certificates available only to the genesis keys. The genesis keys will still be used for update proposals at the beginning of the Shelley era, and so there must be a way to maintain the delegation of these keys to their cold keys. This mapping is also maintained by the delegation state. There is also a mechanism to transfer rewards directly from either the reserves pot or the treasury pot to a reward address. While technically everybody can post such a certificate, the transaction that contains it must be signed by Quorum-many genesis key delegates.

9.1 Delegation Definitions

In Figure 21 we give the delegation primitives. Here we introduce the following primitive datatypes used in delegation:

- $\mathsf{DCert}_{\mathsf{regkey}}$: a stake credential registration certificate.
- $\mathsf{DCert}_{\mathsf{deregkey}}$: a stake credential de-registration certificate.
- $\mathsf{DCert}_{\mathsf{delegate}}$: a stake credential delegation certificate.
- $\mathsf{DCert}_{\mathsf{regpool}}$: a stake pool registration certificate.
- $\mathsf{DCert}_{\mathsf{retirepool}}$: a stake pool retirement certificate.
- $\mathsf{DCert}_{\mathsf{genesis}}$: a genesis key delegation certificate.
- $\mathsf{DCert}_{\mathsf{mir}}$: a move instantaneous rewards certificate.
- DCert: any one of of the seven certificate types above.

The following derived types are introduced:

- 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 hash of the VRF verification key.
  - the pool relays.
  - optional pool metadata (a url and a hash).

  The idea of pool owners is explained in Section 4.4.4 of [SL-D1]. 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 [SL-D1]. The pool reward account is where all pool rewards go. The pool relays and metadata url are explained in Sections 3.4.4 and 4.2 of [SL-D1].

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 is not defined as stake key registrations do not require a witness.
- 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.
- For a $\mathsf{DCert}_{\mathsf{genesis}}$ certificate, cwitness returns the hashkey of the genesis key.
- For a $\mathsf{DCert}_{\mathsf{mir}}$ certificate, cwitness is not defined as there is no single core node or genesis key that posts the certificate.

Abstract types

$$\begin{array}{rcll} url & \in & \mathsf{Url} & \text{a url} \\ mp & \in & \mathsf{MIRPot} & \text{either ReservesMIR or TreasuryMIR} \end{array}$$

Delegation Certificate types

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

Derived types

$$\begin{array}{rcll} \mathsf{PoolMD} & = & \mathsf{Url} \times \mathsf{PoolMDHash} & \text{stake pool metadata} \\ \mathsf{PoolParam} & = & \mathbb{P}\ \mathsf{KeyHash} \times \mathsf{Coin} \times [0,\ 1] \times \mathsf{Coin} & \text{stake pool parameters} \\ & & \times \mathsf{Addr}_{\mathsf{rwd}} \times \mathsf{KeyHash}_{vrf} & \\ & & \mathsf{Url}^* \times \mathsf{PoolMD}^? & \end{array}$$

Certificate Accessor functions

$$\begin{array}{rcll} \mathsf{cwitness} & \in & \mathsf{DCert} \setminus (\mathsf{DCert}_{\mathsf{regkey}} \cup \mathsf{DCert}_{\mathsf{mir}}) \to \mathsf{Credential} & \text{certificate witness} \\ \mathsf{regCred} & \in & \mathsf{DCert}_{\mathsf{regkey}} \to \mathsf{Credential} & \text{registered credential} \\ \mathsf{dpool} & \in & \mathsf{DCert}_{\mathsf{delegate}} \to \mathsf{KeyHash} & \text{pool being delegated to} \\ \mathsf{poolParam} & \in & \mathsf{DCert}_{\mathsf{regpool}} \to \mathsf{PoolParam} & \text{stake pool} \\ \mathsf{retire} & \in & \mathsf{DCert}_{\mathsf{retirepool}} \to \mathsf{Epoch} & \text{epoch of pool retirement} \\ \mathsf{genesisDeleg} & \in & \mathsf{DCert}_{\mathsf{genesis}} \to (\mathsf{KeyHash}_{\mathsf{G}},\ \mathsf{KeyHash},\ \mathsf{KeyHash}_{vrf}) & \text{genesis delegation} \\ \mathsf{credCoinMap} & \in & \mathsf{DCert}_{\mathsf{mir}} \to (\mathsf{Credential}_{stake} \mapsto \mathsf{Coin}) & \text{moved inst. rewards} \\ \mathsf{mirPot} & \in & \mathsf{DCert}_{\mathsf{mir}} \to \mathsf{MIRPot} & \text{pot for inst. rewards} \end{array}$$

Pool Parameter Accessor functions

$$\begin{array}{rcll} \mathsf{poolOwners} & \in & \mathsf{PoolParam} \to \mathbb{P}\ \mathsf{KeyHash} & \text{stake pool owners} \\ \mathsf{poolCost} & \in & \mathsf{PoolParam} \to \mathsf{Coin} & \text{stake pool cost} \\ \mathsf{poolMargin} & \in & \mathsf{PoolParam} \to [0,\ 1] & \text{stake pool margin} \\ \mathsf{poolPledge} & \in & \mathsf{PoolParam} \to \mathsf{Coin} & \text{stake pool pledge} \\ \mathsf{poolRAcnt} & \in & \mathsf{PoolParam} \to \mathsf{Addr}_{\mathsf{rwd}} & \text{stake pool reward account} \\ \mathsf{poolVRF} & \in & \mathsf{PoolParam} \to \mathsf{KeyHash}_{vrf} & \text{stake pool VRF key hash} \end{array}$$

Figure 21: Delegation Definitions

9.2 Delegation Transitions

In Figure 22 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:
  - *rewards* stores the rewards accumulated by stake credentials. These are represented by a finite map from reward addresses to the accumulated rewards.
  - *delegations* stores the delegation relation, mapping stake credentials to the pool to which is delegates.
  - *ptrs* maps stake credentials to the position of the registration certificate in the blockchain. This is needed to lookup the stake hashkey of a pointer address.
  - *fGenDelegs* are the future genesis keys delegations. This variable is needed because genesis keys can only update their delegation with a delay of StabilityWindow slots after submitting the certificate (this is necessary for header validation, see Section 12)
  - *genDelegs* maps genesis key hashes to hashes of the cold key delegates.
  - i_{rwd} stores two maps of stake credentials to Coin, which is used for moving instantaneous rewards at the epoch boundary. One map corresponds to rewards taken from the reserves, and the other corresponds to rewards taken from the treasury.
- PState keeps track of the stake pool information:
  - *poolParams* tracks the parameters associated with each stake pool, such as their costs and margin.
  - When changes are made to the pool parameters late in an epoch, they are staged in *fPoolParams*. These parameters will be updated by another transition (namely EPOCH) when the next epoch starts.
  - *retiring* tracks stake pool retirements, using a map from hashkeys to the epoch in which it will retire.

The operational certificates counters *cs* in the stake pool state are a tool to ensure that blocks containing outdated certificates are rejected. These certificates are part of the block header. For a discussion of why this additional mechanism is needed, see the document [SL-D1], and for the relevant rules, see Section 12.8.

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

Delegation Types

$$\begin{array}{rcll} stakeCred & \in & \mathsf{Credential}_{stake} & = & (\mathsf{KeyHash}_{stake} \uplus \mathsf{ScriptHash}) \\ fGenDelegs & \in & \mathsf{FutGenesisDelegation} & = & (\mathsf{Slot} \times \mathsf{KeyHash}_{\mathsf{G}}) \mapsto (\mathsf{KeyHash} \times \mathsf{KeyHash}_{vrf}) \\ ir & \in & \mathsf{InstantaneousRewards} & = & (\mathsf{Credential}_{stake} \mapsto \mathsf{Coin}) \\ & & & & \quad \times (\mathsf{Credential}_{stake} \mapsto \mathsf{Coin}) \end{array}$$

Delegation States

$$\mathsf{DState} = \left(\begin{array}{rcll} rewards & \in & \mathsf{Credential}_{stake} \mapsto \mathsf{Coin} & \text{rewards} \\ delegations & \in & \mathsf{Credential}_{stake} \mapsto \mathsf{KeyHash}_{pool} & \text{delegations} \\ ptrs & \in & \mathsf{Ptr} \mapsto \mathsf{Credential}_{stake} & \text{pointer to stake credential} \\ fGenDelegs & \in & \mathsf{FutGenesisDelegation} & \text{future genesis key delegations} \\ genDelegs & \in & \mathsf{GenesisDelegation} & \text{genesis key delegations} \\ i_{rwd} & \in & \mathsf{InstantaneousRewards} & \text{instantaneous rewards} \end{array} \right)$$

$$\mathsf{PState} = \left(\begin{array}{rcll} poolParams & \in & \mathsf{KeyHash}_{pool} \mapsto \mathsf{PoolParam} & \text{registered pools to pool parameters} \\ fPoolParams & \in & \mathsf{KeyHash}_{pool} \mapsto \mathsf{PoolParam} & \text{future pool parameters} \\ retiring & \in & \mathsf{KeyHash}_{pool} \mapsto \mathsf{Epoch} & \text{retiring stake pools} \end{array} \right)$$

Delegation Environment

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

Pool Environment

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

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 22: Delegation Transitions

9.3 Delegation Rules

The rules for registering and delegating stake credentials are given in Figure 23. Note that section 5.2 of [SL-D1] describes how a wallet would help a user choose a stake pool, though these concerns are independent of the ledger rules.

- Stake credential registration is handled by Equation (6), 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 reward accounts (which is the source of truth for which stake credentials are registered).

  Registration causes the following state transformation:

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

- Stake credential deregistration is handled by Equation (7). There is a precondition that the credential 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 credential delegation is handled by Equation (8). There is a precondition that the key has been registered. Delegation causes the following state transformation:

  - The delegation relation is updated so that the stake credential 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.

- Genesis key delegation is handled by Equation (9). There is a precondition that the genesis key is already in the mapping *genDelegs*. Genesis delegation causes the following state transformation:

  - The future genesis delegation relation is updated with the new delegate to be adopted in StabilityWindow-many slots.

- Moving instantaneous rewards is handled by **??** and **??**. There is a precondition that the current slot is early enough in the current epoch and that the available reserves or treasury are sufficient to pay for the instantaneous rewards.

The DELEG rule has ten possible predicate failures:

- In the case of a key registration certificate, if the staking credential is already registered, there is a *StakeKeyAlreadyRegistered* failure.
- In the case of a key deregistration certificate, if the key is not registered, there is a *StakeKeyNotRegistered* failure.
- In the case of a key deregistration certificate, if the associated reward account is non-zero, there is a *StakeKeyNonZeroAccountBalance* failure.

- In the case of a non-existing stake pool key in a delegation certificate, there is a *StakeDelegationImpossible* failure.
- In the case of a pool delegation certificate, there is a *WrongCertificateType* failure.
- In the case of a genesis key delegation certificate, if the genesis key is not in the domain of the genesis delegation mapping, there is a *GenesisKeyNotInMapping* failure.
- In the case of a genesis key delegation certificate, if the delegate key is in the range of the genesis delegation mapping, there is a *DuplicateGenesisDelegate* failure.
- In the case of insufficient reserves to pay the instantaneous rewards, there is a *InsufficientForInstantaneousRewards* failure.
- In the case that a MIR certificate is issued during the last StabilityWindow-many slots of the epoch, there is a *MIRCertificateTooLateinEpoch* failure.
- In the case of a genesis key delegation certificate, if the VRF key is in the range of the genesis delegation mapping, there is a *DuplicateGenesisVRF* failure.

$$\text{Deleg-Reg}\ \frac{c \in \mathsf{DCert}_{\mathsf{regkey}} \qquad hk := \mathsf{regCred}\ c \qquad hk \notin \mathrm{dom}\ rewards}{\begin{pmatrix} slot \\ ptr \\ acnt \end{pmatrix} \vdash \begin{pmatrix} rewards \\ delegations \\ ptrs \\ fGenDelegs \\ genDelegs \\ i_{rwd} \end{pmatrix} \xrightarrow[\text{DELEG}]{c} \begin{pmatrix} rewards \cup \{hk \mapsto 0\} \\ delegations \\ ptrs \cup \{ptr \mapsto hk\} \\ fGenDelegs \\ genDelegs \\ i_{rwd} \end{pmatrix}} \tag{6}$$

$$\text{Deleg-Dereg}\ \frac{c \in \mathsf{DCert}_{\mathsf{deregkey}} \qquad hk := \mathsf{cwitness}\ c \qquad hk \mapsto 0 \in rewards}{\begin{pmatrix} slot \\ ptr \\ acnt \end{pmatrix} \vdash \begin{pmatrix} rewards \\ delegations \\ ptrs \\ fGenDelegs \\ genDelegs \\ i_{rwd} \end{pmatrix} \xrightarrow[\text{DELEG}]{c} \begin{pmatrix} \{hk\} \lhd rewards \\ \{hk\} \lhd delegations \\ ptrs \rhd \{hk\} \\ fGenDelegs \\ genDelegs \\ i_{rwd} \end{pmatrix}} \tag{7}$$

$$\text{Deleg-Deleg}\ \frac{c \in \mathsf{DCert}_{\mathsf{delegate}} \qquad hk := \mathsf{cwitness}\ c \qquad hk \in \mathrm{dom}\ rewards}{\begin{pmatrix} slot \\ ptr \\ acnt \end{pmatrix} \vdash \begin{pmatrix} rewards \\ delegations \\ ptrs \\ fGenDelegs \\ genDelegs \\ i_{rwd} \end{pmatrix} \xrightarrow[\text{DELEG}]{c} \begin{pmatrix} rewards \\ delegations \cup\!\!\rightarrow \{hk \mapsto \mathsf{dpool}\ c\} \\ ptrs \\ fGenDelegs \\ genDelegs \\ i_{rwd} \end{pmatrix}} \tag{8}$$

$$\text{Deleg-Gen}\ \frac{\begin{array}{c} c \in \mathsf{DCert}_{\mathsf{genesis}} \qquad (gkh,\ vkh,\ vrf) := \mathsf{genesisDeleg}\ c \\ s' := slot + \mathsf{StabilityWindow} \qquad gkh \in \mathrm{dom}\ genDelegs \\ \\ cod := \{(k,\ v) \mid (\ g\ \mapsto (k,\ v)) \in genDelegs\ ,\ g \neq gkh\} \\ fod := \{(k,\ v) \mid ((\_,\ g) \mapsto (k,\ v)) \in fGenDelegs,\ g \neq gkh\} \\ currentOtherColdKeyHashes := \{k \mid \_ \mapsto (k,\ \_) \in cod\} \\ currentOtherVrfKeyHashes := \{v \mid \_ \mapsto (\_,\ v) \in cod\} \\ futureOtherColdKeyHashes := \{k \mid \_ \mapsto (k,\ \_) \in fod\} \\ futureOtherVrfKeyHashes := \{v \mid \_ \mapsto (\_,\ v) \in fod\} \\ vkh \notin currentOtherColdKeyHashes \cup futureOtherColdKeyHashes \\ vrf \notin currentOtherVrfKeyHashes \cup futureOtherVrfKeyHashes \\ fdeleg := \{(s',\ gkh) \mapsto (vkh,\ vrf)\} \end{array}}{\begin{pmatrix} slot \\ ptr \\ acnt \end{pmatrix} \vdash \begin{pmatrix} rewards \\ delegations \\ ptrs \\ fGenDelegs \\ genDelegs \\ i_{rwd} \end{pmatrix} \xrightarrow[\text{DELEG}]{c} \begin{pmatrix} rewards \\ delegations \\ ptrs \\ fGenDelegs \cup\!\!\rightarrow fdeleg \\ genDelegs \\ i_{rwd} \end{pmatrix}} \tag{9}$$

Figure 23: Delegation Inference Rules

$$
\text{Deleg-Mir}\ \frac{
\begin{array}{c}
c \in \mathsf{DCert}_{\mathsf{mir}} \\
slot < \mathsf{firstSlot}\ ((\mathsf{epoch}\ slot) + 1) - \mathsf{StabilityWindow} \\
(irR,\ irT) := i_{rwd} \qquad (treasury,\ reserves) := acnt \\
(pot,\ irPot) := \begin{cases} (reserves,\ irR) & \mathsf{mirPot}\ c = \mathsf{ReservesMIR} \\ (treasury,\ irT) & \mathsf{mirPot}\ c = \mathsf{TreasuryMIR} \end{cases} \\
combined := (\mathsf{mirTarget}\ c) \underset{\leftarrow}{\cup} irPot \\
\sum_{\_ \mapsto val \in combined} val \leq pot \\
\forall r \in \mathsf{range}\ irPot,\ r \geq 0 \\
i'_{rwd} := \begin{cases} (combined,\ irT) & \mathsf{mirPot}\ c = \mathsf{ReservesMIR} \\ (irR,\ combined) & \mathsf{mirPot}\ c = \mathsf{TreasuryMIR} \end{cases}
\end{array}
}{
\begin{array}{l} slot \\ ptr \\ acnt \end{array} \vdash \begin{pmatrix} rewards \\ delegations \\ ptrs \\ fGenDelegs \\ genDelegs \\ i_{rwd} \end{pmatrix} \xrightarrow[\text{DELEG}]{c} \begin{pmatrix} rewards \\ delegations \\ ptrs \\ fGenDelegs \\ genDelegs \\ i'_{rwd} \end{pmatrix}
} \tag{10}
$$

Figure 24: Move Instantaneous Rewards Inference Rule

9.4 Stake Pool Rules

The rules for updating the part of the ledger state defining the current stake pools are given in Figure 25. The calculation of stake distribution is described in Section 11.4.

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 (11). 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 (12). This rule, which also matches on the certificate type DCertRegPool, is distinguished from Equation (11) by the requirement that the pool be registered.

  Unlike the initial stake pool registrations, the pool parameters will not change until the next epoch, after stake distribution snapshots are taken. This gives delegators an entire epoch to respond to changes in stake pool parameters. The staging is achieved by adding updates to the mapping *fPoolParams*, which will override *poolParam* with new values in the EPOCH transition (see Figure Figure 45).

  This rule also ends stake pool retirements. Note that *poolParams* is **not** updated. The registration creation slot does does not change.
- Stake pool retirements are handled by Equation (13). 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) and that it is no more than 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 11.

  Retirement 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.

The POOL rule has four predicate failures:

- In the case of a pool registration or re-registration certificate, if specified pool cost parameter is smaller than the value of the protocol parameter minPoolCost, there is a *StakePoolCostTooLow* failure.
- In the case of a pool retirement certificate, if the pool key is not in the domain of the stake pools mapping, there is a *StakePoolNotRegisteredOnKey* failure.
- In the case of a pool retirement certificate, if the retirement epoch is not between the current epoch and the relative maximal epoch from the current epoch, there is a *StakePoolRetirementWrongEpoch* failure.
- If the delegation certificate is not of one of the pool types, there is a *WrongCertificateType* failure.

$$\text{Pool-Reg}\ \frac{\begin{array}{c} c \in \mathsf{DCert}_{\mathsf{regpool}} \qquad hk := \mathsf{cwitness}\ c \qquad pool := \mathsf{poolParam}\ c \\ hk \notin \mathrm{dom}\ poolParams \qquad \mathsf{poolCost}\ pool \geq \mathsf{minPoolCost}\ pp \end{array}}{slot,\ pp \vdash \begin{pmatrix} poolParams \\ fPoolParams \\ retiring \end{pmatrix} \xrightarrow[\text{POOL}]{c} \begin{pmatrix} \boldsymbol{poolParams} & \boldsymbol{\cup} & \boldsymbol{\{hk \mapsto pool\}} \\ fPoolParams & & \\ retiring & & \end{pmatrix}} \tag{11}$$

$$\text{Pool-reReg}\ \frac{\begin{array}{c} c \in \mathsf{DCert}_{\mathsf{regpool}} \qquad hk := \mathsf{cwitness}\ c \qquad pool := \mathsf{poolParam}\ c \\ hk \in \mathrm{dom}\ poolParams \qquad \mathsf{poolCost}\ pool \geq \mathsf{minPoolCost}\ pp \end{array}}{slot,\ pp \vdash \begin{pmatrix} poolParams \\ fPoolParams \\ retiring \end{pmatrix} \xrightarrow[\text{POOL}]{c} \begin{pmatrix} poolParams & & \\ \boldsymbol{fPoolParams} & \boldsymbol{\underset{\rightarrow}{\cup}} & \boldsymbol{\{hk \mapsto pool\}} \\ \boldsymbol{\{hk\}} & \boldsymbol{\lhd} & \boldsymbol{retiring} \end{pmatrix}} \tag{12}$$

$$\text{Pool-Retire}\ \frac{\begin{array}{c} c \in \mathsf{DCert}_{\mathsf{retirepool}} \qquad hk := \mathsf{cwitness}\ c \qquad hk \in \mathrm{dom}\ poolParams \\ e := \mathsf{retire}\ c \qquad cepoch := \mathsf{epoch}\ slot \qquad cepoch < e \leq cepoch + (\mathsf{emax}\ pp) \end{array}}{slot,\ pp \vdash \begin{pmatrix} poolParams \\ fPoolParams \\ retiring \end{pmatrix} \xrightarrow[\text{POOL}]{c} \begin{pmatrix} poolParams & & \\ fPoolParams & & \\ \boldsymbol{retiring} & \boldsymbol{\underset{\rightarrow}{\cup}} & \boldsymbol{\{hk \mapsto e\}} \end{pmatrix}} \tag{13}$$

Figure 25: Pool Inference Rule

9.5 Delegation and Pool Combined Rules

We now combine the delegation and pool transition systems. Figure 26 gives the state, environment and transition type for the combined transition.

Delegation and Pool Combined Environment

$$\mathsf{DPEnv} = \left(\begin{array}{rcll} \mathit{slot} & \in & \mathsf{Slot} & \text{slot} \\ \mathit{ptr} & \in & \mathsf{Ptr} & \text{certificate pointer} \\ \mathit{pp} & \in & \mathsf{PParams} & \text{protocol parameters} \\ \mathit{acnt} & \in & \mathsf{Acnt} & \text{accounting state} \end{array} \right)$$

Delegation and Pool Combined State

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

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 26: Delegation and Pool Combined Transition Type

Figure 27, gives the rules for the combined transition. Note that for any given certificate, at most one of the two rules (Equation (14) and Equation (15)) will be successful, since the pool certificates are disjoint from the delegation certificates.

Delegation and Pool Combined Rules

$$\text{Delpl-Deleg}\ \frac{\begin{matrix}slot\\ptr\\acnt\end{matrix} \vdash dstate \xrightarrow[\text{DELEG}]{c} dstate'}{\begin{matrix}slot\\ptr\\pp\\acnt\end{matrix} \vdash \begin{pmatrix} dstate \\ pstate \end{pmatrix} \xrightarrow[\text{DELPL}]{c} \begin{pmatrix} \mathbf{dstate'} \\ pstate \end{pmatrix}} \tag{14}$$

$$\text{Delpl-Pool}\ \frac{\begin{matrix}slot\\pp\end{matrix} \vdash pstate \xrightarrow[\text{POOL}]{c} pstate'}{\begin{matrix}slot\\ptr\\pp\\acnt\end{matrix} \vdash \begin{pmatrix} dstate \\ pstate \end{pmatrix} \xrightarrow[\text{DELPL}]{c} \begin{pmatrix} dstate \\ \mathbf{pstate'} \end{pmatrix}} \tag{15}$$

Figure 27: 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 27 above.

Figure 28 defines the types for the delegation certificate sequence transition.

Certificate Sequence Environment

$$\mathsf{DPSEnv} = \begin{pmatrix} slot & \in & \mathsf{Slot} & \text{slot} \\ txIx & \in & \mathsf{Ix} & \text{transaction index} \\ pp & \in & \mathsf{PParams} & \text{protocol parameters} \\ tx & \in & \mathsf{Tx} & \text{transaction} \\ acnt & \in & \mathsf{Acnt} & \text{accounting state} \end{pmatrix}$$

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

Figure 28: Delegation sequence transition type

Figure 29 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 (16). 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 credential 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 (17). 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.

$$\text{Seq-delg-base}\ \frac{\begin{array}{c} wdrls := \mathsf{txwdrls}\ (\mathsf{txbody}\ tx) \qquad wdrls \subseteq rewards \\ rewards' := rewards \underset{\rightarrow}{\cup} \{(w, 0) \mid w \in \mathrm{dom}\ wdrls\} \end{array}}{\begin{array}{c} slot \\ txIx \\ pp \\ tx \\ acnt \end{array} \vdash \left(\begin{array}{c} \left(\begin{array}{r} rewards \\ delegations \\ ptrs \\ fGenDelegs \\ genDelegs \\ i_{rwd} \end{array} \right) \\ \left(\begin{array}{c} poolParams \\ fPoolParams \\ retiring \end{array} \right) \end{array} \right) \xrightarrow[\text{DELEGS}]{\epsilon} \left(\begin{array}{c} \left(\begin{array}{r} \boldsymbol{rewards'} \\ delegations \\ ptrs \\ fGenDelegs \\ genDelegs \\ i_{rwd} \end{array} \right) \\ \left(\begin{array}{c} poolParams \\ fPoolParams \\ retiring \end{array} \right) \end{array} \right)} \tag{16}$$

$$\text{Seq-delg-ind}\ \frac{\begin{array}{c} \begin{array}{c} slot \\ txIx \\ pp \\ tx \\ acnt \end{array} \vdash dpstate \xrightarrow[\text{DELEGS}]{\Gamma} dpstate' \\ c \in \mathsf{DCert}_{\mathsf{delegate}} \Rightarrow \mathsf{dpool}\ c \in \mathrm{dom}\ poolParams \\ ptr := (slot,\ txIx,\ \mathsf{len}\ \Gamma) \\ \begin{array}{c} slot \\ ptr \\ pp \\ acnt \end{array} \vdash dpstate' \xrightarrow[\text{DELPL}]{c} dpstate'' \end{array}}{\begin{array}{c} slot \\ txIx \\ pp \\ tx \\ acnt \end{array} \vdash dpstate \xrightarrow[\text{DELEGS}]{\Gamma ; c} \boldsymbol{dpstate''}} \tag{17}$$

Figure 29: Delegation sequence rules

The DELEGS rule has two predicate failures:

- In the case of a key delegation certificate, if the pool key is not registered, there is a *DelegateeNotRegistered* failure.
- If the withdrawals mapping of the transaction is not a subset of the rewards mapping of the delegation state, there is a *WithdrawalsNotInRewards* failure.

10 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 30 defines the types for this transition. The environment for this rule consists of:

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

The ledger state consists of:

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

Ledger environment

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

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 30: Ledger transition-system types

Figure 30 defines the ledger state transition. It has a single rule, which first calls the UTXOW transition, then calls the DELEGS transition.

$$\frac{\begin{array}{l}\begin{array}{l}slot\\txIx\\pp\\tx\\acnt\end{array} \vdash dpstate \xrightarrow[\text{DELEGS}]{\mathsf{txcerts}\ (\mathsf{txbody}\ tx)} dpstate' \\ (dstate, pstate) := dpstate \\ (\_,\_,\_,\_,genDelegs,\_) := dstate \\ (poolParams,\_,\_) := pstate \\ \begin{array}{l}slot\\pp\\poolParams\\genDelegs\end{array} \vdash utxoSt \xrightarrow[\text{UTXOW}]{tx} utxoSt'\end{array}}{\text{ledger}\ \begin{array}{l}slot\\txIx\\pp\\acnt\end{array} \vdash \left(\begin{array}{l}utxoSt\\dpstate\end{array}\right) \xrightarrow[\text{LEDGER}]{tx} \left(\begin{array}{l}\boldsymbol{utxoSt'}\\\boldsymbol{dpstate'}\end{array}\right)} \tag{18}$$

Figure 31: Ledger inference rule

The transition system LEDGER in Figure 31 is iterated in LEDGERS in order to process a list of transactions.

Ledger Sequence transitions

$$\_ \vdash \_ \xrightarrow[\text{LEDGERS}]{} \_ \subseteq \mathbb{P}\left((\mathsf{Slot} \times \mathsf{PParams} \times \mathsf{Coin}) \times \mathsf{LState} \times \mathsf{Tx}^* \times \mathsf{LState}\right)$$

Figure 32: Ledger Sequence transition-system types

$$\text{Seq-ledger-base}\ \frac{}{\begin{matrix} slot \\ pp \\ acnt \end{matrix} \vdash ls \xrightarrow[\text{LEDGERS}]{\epsilon} ls} \tag{19}$$

$$\text{Seq-ledger-ind}\ \frac{\begin{matrix} slot \\ pp \\ acnt \end{matrix} \vdash ls \xrightarrow[\text{LEDGERS}]{\Gamma} ls' \qquad \begin{matrix} slot \\ \mathsf{len}\ \Gamma \\ pp \\ acnt \end{matrix} \vdash ls' \xrightarrow[\text{LEDGER}]{tx} ls''}{\begin{matrix} slot \\ pp \\ acnt \end{matrix} \vdash ls \xrightarrow[\text{LEDGERS}]{\Gamma;\, tx} \mathbf{ls''}} \tag{20}$$

Figure 33: Ledger sequence rules

11 Rewards and the Epoch Boundary

This chapter introduces the epoch boundary transition system and the related reward calculation.

The transition system is defined in Section 11.8, and involves taking stake distribution snapshots (Sections 11.4 and 11.5), retiring stake pools (Section 11.6), and performing protocol updates (Section 11.7). The reward calculation, defined in Sections 11.9 and 11.10, distributes the leader election rewards.

11.1 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}

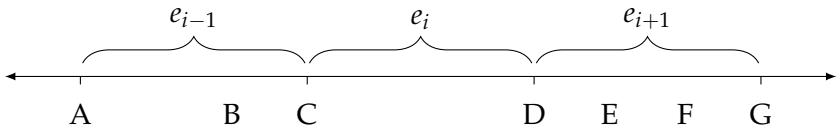
(C) Epoch e_i begins.

(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.

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

(F) The reward calculation is finished and an update to the ledger state is ready to be applied.

(G) 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.

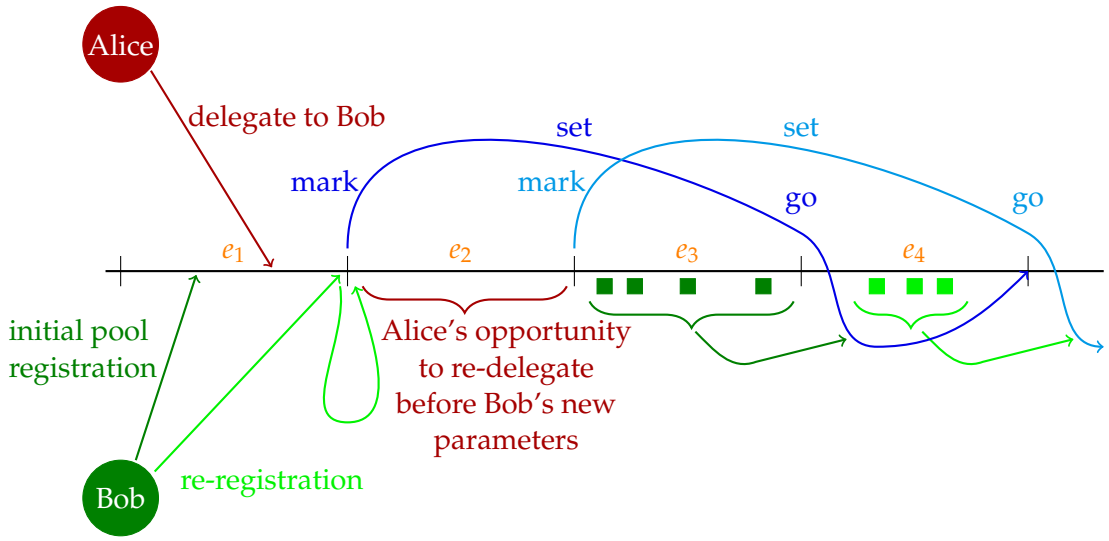
The two main transition systems in this section are:

- The transition system named EPOCH, which is defined in Section 11.8, covers what happens at the epoch boundary, such as at (A), (C), (D) and (G).
- The transition named RUPD, which is defined in Section 12.6, covers the reward calculation that happens between (E) and (F).

NOTE

Between time D and E we are concerned with chain growth and stability. Therefore this duration can be stated as 2k blocks (to state it in slots requires details about the particular version of the Ouroboros protocol). The duration between F and G is also 2k blocks. Between E and F a single honest block is enough to ensure a random nonce.

11.2 Example Illustration of the Reward Cycle



Bob registers his stake pool in epoch e_1. Alice delegates to Bob's stake pool in epoch e_1. Just before the end of epoch e_1, Bob submits a stake pool re-registration, changing his pool parameters. The change in parameters is not immediate, as shown by the curved arrow around the epoch boundary.

A snapshot is taken on the e_1/e_2 boundary. It is labeled "mark" initially. This snapshot includes Alice's delegation to Bob's pool, and Bob's pool parameters and listed in the initial pool registration certificate.

If Alice changes her delegation choice any time during epoch e_2, she will never be effected by Bob's change of parameters.

A new snapshot is taken on the e_2/e_3 boundary. The previous (darker blue) snapshot is now labeled "set", and the new one labeled "mark". The "set" snapshot is used for leader election in epoch e_3.

On the e_3/e_4 boundary, the darker blue snapshot is labeled "go" and the lighter blue snapshot is labeled "set". Bob's stake pool performance during epoch e_3 (he produced 4 blocks) will be used with the darker blue snapshot for the rewards which will be handed out at the beginning of epoch e_5.

11.3 Helper Functions and Accounting Fields

Figure 34 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.
- The function poolStake filters the stake distribution to one stake pool.

The Figure 35 lists the accounting fields, denoted by Acnt, which will be used throughout this section. It 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.

More will be said about the general accounting system in Section 11.10.

Total possible refunds

$$\mathsf{obligation} \in \mathsf{PParams} \to (\mathsf{Credential}_{stake} \mapsto \mathsf{Coin}) \to (\mathsf{KeyHash}_{pool} \mapsto \mathsf{PoolParam}) \to \mathsf{Coin}$$
$$\mathsf{obligation}\ pp\ rewards\ poolParams =$$
$$(\mathsf{keyDeposit}\ pp) \cdot |rewards| + (\mathsf{poolDeposit}\ pp) \cdot |poolParams|$$

Filter Stake to one Pool

$$\mathsf{poolStake} \in \mathsf{KeyHash}_{pool} \to (\mathsf{KeyHash}_{stake} \mapsto \mathsf{KeyHash}_{pool}) \to \mathsf{Stake} \to \mathsf{Stake}$$
$$\mathsf{poolStake}\ hk\ delegs\ stake = \mathrm{dom}\ (delegs \rhd \{hk\}) \lhd stake$$

Figure 34: Helper Functions used in Rewards and Epoch Boundary

Accounting Fields

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

Figure 35: Accounting fields

11.4 Stake Distribution Calculation

This section defines the stake distribution calculations. Figure 36 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.

Derived types

$$\begin{array}{rlcll} blocks & \in \mathsf{BlocksMade} & = & \mathsf{KeyHash}_{pool} \mapsto \mathbb{N} & \text{blocks made by stake pools} \\ stake & \in \mathsf{Stake} & = & \mathsf{Credential} \mapsto \mathsf{Coin} & \text{stake} \end{array}$$

Figure 36: Epoch definitions

The stake distribution calculation is given in Figure 37.

- $\mathsf{aggregate}_+$ takes a relation on $A \times B$, where B is any monoid $(B, +, e)$ and returns a map from each $a \in A$ to the "sum" (using the monoidal $+$ operation) of all $b \in B$ such that $(a, b) \in A \times B$.
- 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:
  - $\mathsf{stakeCred_b}^{-1}$, relating credentials to (base) addresses
  - $(\mathsf{addrPtr} \circ ptr)^{-1}$, relating credentials to (pointer) addresses
  - range $utxo$, relating addresses to coins
  - $\mathsf{stakeCred_r}^{-1} \circ rewards$, 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)

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

$$\mathsf{stakeDistr}\ utxo\ dstate\ pstate =$$

$$\left((\operatorname{dom} activeDelegs) \lhd (\mathsf{aggregate}_+\ stakeRelation)\right,\ delegations,\ poolParams\right)$$

where

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

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

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

$$activeDelegs = (\operatorname{dom} rewards) \lhd delegations \rhd (\operatorname{dom} poolParams)$$

Figure 37: Stake Distribution Function

11.5 Snapshot Transition

The state transition types for stake distribution snapshots are given in Figure 38. Each snapshot consists of:

- *stake*, a stake distribution, which is defined in Figure 36 as a mapping of credentials to coin.
- *delegations*, a delegation map, mapping credentials to stake pools.
- *poolParameters*, storing the pool parameters of each stake pool.

The type Snapshots contains the information needing to be saved on the epoch boundary:

- $pstake_{mark}$, $pstake_{set}$ and $pstake_{go}$ are the three snapshots as explained in Section 11.1.
- *feeSS* stores the fees which are added to the reward pot during the next reward update calculation, which is then subtracted from the fee pot on the epoch boundary.

Snapshots

$$\mathsf{Snapshot} = \left(\begin{array}{rcll} \mathit{stake} & \in & \mathsf{Stake} & \text{stake distribution} \\ \mathit{delegations} & \in & \mathsf{Credential} \mapsto \mathsf{KeyHash}_{pool} & \text{stake delegations} \\ \mathit{poolParameters} & \in & \mathsf{KeyHash}_{pool} \mapsto \mathsf{PoolParam} & \text{pool parameters} \end{array}\right)$$

$$\mathsf{Snapshots} = \left(\begin{array}{rcll} pstake_{mark} & \in & \mathsf{Snapshot} & \text{newest stake} \\ pstake_{set} & \in & \mathsf{Snapshot} & \text{middle stake} \\ pstake_{go} & \in & \mathsf{Snapshot} & \text{oldest stake} \\ \mathit{feeSS} & \in & \mathsf{Coin} & \text{fee snapshot} \end{array}\right)$$

Snapshot transitions

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

Figure 38: Snapshot transition-system types

The snapshot transition rule is given in Figure 39. 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 current fees pot is stored in *feeSS*. Note that this value will not change during the epoch, unlike the *fees* value in the UTxO state.

$$\text{Snapshot}\ \frac{\begin{array}{rcl}((utxo,\ \_, fees,\ \_),\ (dstate,\ pstate)) & := & lstate \\ stake & := & \mathsf{stakeDistr}\ utxo\ dstate\ pstate\end{array}}{lstate \vdash \begin{pmatrix} pstake_{mark} \\ pstake_{set} \\ pstake_{go} \\ feeSS \end{pmatrix} \xrightarrow[\text{SNAP}]{} \begin{pmatrix} \boldsymbol{stake} \\ \boldsymbol{pstake_{mark}} \\ \boldsymbol{pstake_{set}} \\ \boldsymbol{fees} \end{pmatrix}} \tag{21}$$

Figure 39: Snapshot Inference Rule

11.6 Pool Reaping Transition

Figure 40 defines the types for the pool reap transition, which is responsible for removing pools slated for retirement in the given epoch.

Pool Reap State

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

Pool Reap transitions

$$\_ \vdash \_ \xrightarrow[\text{POOLREAP}]{} \_ \in \mathbb{P}\ (\mathsf{PParams} \times \mathsf{PIReapState} \times \mathsf{Epoch} \times \mathsf{PIReapState})$$

Figure 40: Pool Reap Transition

The pool-reap transition rule is given in Figure 41. 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.
- The deposit pool is reduced by the amount of claimed and unclaimed refunds.
- Any delegation to a retiring pool is removed.
- Each retiring pool is removed from all four maps in the pool state.

$$
\begin{aligned}
\mathit{retired} &:= \mathrm{dom}\,(\mathit{retiring}^{-1}\,e)\\
\mathit{pr} &:= \{\mathit{hk} \mapsto (\mathsf{poolDeposit}\;\mathit{pp}) \mid \mathit{hk} \in \mathit{retired}\}\\
\mathit{rewardAcnts} &:= \{\mathit{hk} \mapsto \mathsf{poolRAcnt}\;\mathit{pool} \mid \mathit{hk} \mapsto \mathit{pool} \in \mathit{retired} \lhd \mathit{poolParams}\}\\
\mathit{rewardAcnts}' &:= \left\{a \mapsto \sum \mathit{pr}(\mathit{rewardAcnts}^{-1}(a)) \;\middle|\; a \in \mathrm{range}\,\mathit{rewardAcnts}\right\}\\
\mathit{refunds} &:= \mathrm{dom}\,\mathit{rewards} \lhd \mathit{rewardAcnts}'\\
\mathit{mRefunds} &:= \mathrm{dom}\,\mathit{rewards} \not\lhd \mathit{rewardAcnts}'\\
\mathit{refunded} &:= \sum_{\_ \mapsto c \in \mathit{refunds}} c\\
\mathit{unclaimed} &:= \sum_{\_ \mapsto c \in \mathit{mRefunds}} c
\end{aligned}
$$

Pool-Reap

$$
\mathit{pp} \vdash \begin{pmatrix} \mathit{utxo}\\ \mathit{deposited}\\ \mathit{fees}\\ \mathit{ppup}\\ \\ \mathit{treasury}\\ \mathit{reserves}\\ \\ \mathit{rewards}\\ \mathit{delegations}\\ \mathit{ptrs}\\ \mathit{genDelegs}\\ \mathit{fGenDelegs}\\ i_{rwd}\\ \\ \mathit{poolParams}\\ \mathit{fPoolParams}\\ \mathit{retiring} \end{pmatrix} \xrightarrow[\text{POOLREAP}]{e} \begin{pmatrix} \mathit{utxo} & & \\ \mathit{deposited} & - & (\mathit{unclaimed} + \mathit{refunded})\\ \mathit{fees} & & \\ \mathit{ppup} & & \\ & & \\ \mathit{treasury} & + & \mathit{unclaimed}\\ \mathit{reserves} & & \\ & & \\ \mathit{rewards} & \cup_+ & \mathit{refunds}\\ \mathit{delegations} & \rhd & \mathit{retired}\\ \mathit{ptrs} & & \\ \mathit{genDelegs} & & \\ \mathit{fGenDelegs} & & \\ i_{rwd} & & \\ & & \\ \mathit{retired} & \not\lhd & \mathit{poolParams}\\ \mathit{retired} & \not\lhd & \mathit{fPoolParams}\\ \mathit{retired} & \not\lhd & \mathit{retiring} \end{pmatrix} \tag{22}
$$

Figure 41: Pool Reap Inference Rule

11.7 Protocol Parameters Update Transition

Finally, reaching the epoch boundary may trigger a change in the protocol parameters. The protocol parameters environment consists of the delegation and pool states, and the signal is an optional new collection of protocol parameters 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 42.

New Proto Param environment

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

New Proto Param States

$$\mathsf{NewPParamState} = \begin{pmatrix} \mathit{utxoSt} & \in & \mathsf{UTxOState} & \text{utxo state} \\ \mathit{acnt} & \in & \mathsf{Acnt} & \text{accounting} \\ \mathit{pp} & \in & \mathsf{PParams} & \text{current protocol parameters} \end{pmatrix}$$

New Proto Param transitions

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

Helper Functions

$$\mathsf{updatePpup} \in \mathsf{UTxOState} \to \mathsf{PParams} \to \mathsf{UTxOState}$$

$$\mathsf{updatePpup}\ \mathit{utxoSt}\ \mathit{pp} = \begin{cases} (\mathit{utxo}, \mathit{deposited}, \mathit{fees}, (\mathit{fpup}, \emptyset)) & \mathit{canFollow} \\ (\mathit{utxo}, \mathit{deposited}, \mathit{fees}, (\emptyset, \emptyset)) & \text{otherwise} \end{cases}$$

where

$$\mathit{canFollow} = \forall \mathit{ps} \in \mathsf{range}\ \mathit{pup},\ \mathit{pv} \mapsto v \in \mathit{ps} \implies \mathsf{pvCanFollow}\ (\mathsf{pv}\ \mathit{pp})\ v$$

$$(\mathit{utxo}, \mathit{deposited}, \mathit{fees}, (\mathit{pup}, \mathit{fpup})) = \mathit{utxoSt}$$

Figure 42: New Proto Param transition-system types

Figure 43 defines the new protocol parameter transition. The transition has two rules, depending on whether or not the new protocol parameters meet some requirements. In particular, we require that the new parameters would not incur a debt of the system that can not be covered by the reserves, and that the max block size is greater than the sum of the max transaction size and the max header size. If the requirements are met, the new protocol parameters are accepted, the proposal is reset, and the reserves are adjusted to account for changes in the deposits. Otherwise, the only change is that the proposal is reset.

The NEWPP rule also cleans up the protocol parameter update proposals, by calling updatePpup on the UTxO state. The updatePpup sets the protocol parameter updates to the future protocol parameter updates provided the protocol versions all can follow from the version given in the protocol parameters, or the emptyset otherwise. In any case, the future protocol parameters update proposals are set to the empty set. If new protocol parameters are being adopted, then these is the value given to updatePpup, otherwise the old parameters are given.

Regarding adjusting the reserves for changes in the deposits, one of three things happens:

- If the new protocol parameters mean that **fewer** funds are required in the deposit pot to cover all possible refunds, then the excess is moved to the reserves.
- 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 funds are moved from the reserve pot to cover the difference.

- 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 24 meets the precondition and the only change of state is that the update proposals are reset.

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.

$$pp_{new} \neq \diamond$$

$$\begin{array}{rcl}
(utxo,\ deposited,\ fees,\ ppup) & := & utxoSt \\
(rewards,\ _,\ _,\ _,\ _,\ i_{rwd}) & := & dstate \\
(poolParams,\ _,\ _) & := & pstate \\
oblg_{cur} & := & \mathsf{obligation}\ pp\ rewards\ poolParams \\
oblg_{new} & := & \mathsf{obligation}\ pp_{new}\ rewards\ poolParams \\
diff & := & oblg_{cur} - oblg_{new}
\end{array}$$

$$oblg_{cur} = deposited$$
$$reserves + diff \geq \sum_{\_ \mapsto val \in i_{rwd}} val$$
$$\mathsf{maxTxSize}\ pp_{new} + \mathsf{maxHeaderSize}\ pp_{new} < \mathsf{maxBlockSize}\ pp_{new}$$

$$utxoSt' := (utxo,\ \mathbf{oblg_{new}},\ fees,\ ppup)$$
$$utxoSt'' := \mathsf{updatePpup}\ utxoSt'\ pp_{new}$$

$$(treasury,\ reserves) := acnt$$
$$acnt' := (treasury,\ \mathbf{reserves + diff})$$

New-Proto-Param-Accept

$$\begin{array}{r} dstate \\ pstate \end{array} \vdash \begin{pmatrix} utxoSt \\ acnt \\ pp \end{pmatrix} \xrightarrow[\text{NEWPP}]{pp_{new}} \begin{pmatrix} \mathbf{utxoSt''} \\ \mathbf{acnt'} \\ \mathbf{pp_{new}} \end{pmatrix} \tag{23}$$

$$\left(\begin{array}{c}
pp_{new} = \diamond \\
\vee \\
reserves + diff < \sum_{_ \mapsto val \in i_{rwd}} val \\
\vee \\
\mathsf{maxTxSize}\ pp_{new} + \mathsf{maxHeaderSize}\ pp_{new} \geq \mathsf{maxBlockSize}\ pp_{new}
\end{array} \right)$$

$$\begin{array}{rcl}
(rewards,\ _,\ _,\ _,\ _,\ i_{rwd}) & := & dstate \\
(poolParams,\ _,\ _) & := & pstate \\
oblg_{cur} & := & \mathsf{obligation}\ pp\ rewards\ poolParams \\
oblg_{new} & := & \mathsf{obligation}\ pp_{new}\ rewards\ poolParams \\
diff & := & oblg_{cur} - oblg_{new}
\end{array}$$

$$utxoSt' := \mathsf{updatePpup}\ utxoSt\ pp$$

New-Proto-Param-Denied

$$\begin{array}{r} dstate \\ pstate \end{array} \vdash \begin{pmatrix} utxoSt \\ acnt \\ pp \end{pmatrix} \xrightarrow[\text{NEWPP}]{pp_{new}} \begin{pmatrix} \mathbf{utxoSt'} \\ acnt \\ pp \end{pmatrix} \tag{24}$$

Figure 43: New Proto Param Inference Rule

11.8 Complete Epoch Boundary Transition

Finally, it is possible to define the complete epoch boundary transition type, which is defined in Figure 44. The transition has no evironment. The state is made up of the the accounting state, the snapshots, the ledger state and the protocol parameters. The transition uses a helper function votedValue which returns the consensus value of update proposals in the event that consensus is met. **Note that** votedValue **is only well-defined if** *quorum* **is greater than half the number of core nodes, i.e.** Quorum $>$ $|genDelegs|/2$ **.**

Epoch States

$$\mathsf{EpochState} = \left(\begin{array}{rcll} acnt & \in & \mathsf{Acnt} & \text{accounting} \\ ss & \in & \mathsf{Snapshots} & \text{snapshots} \\ ls & \in & \mathsf{LState} & \text{ledger state} \\ prevPp & \in & \mathsf{PParams} & \text{previous protocol parameters} \\ pp & \in & \mathsf{PParams} & \text{protocol parameters} \end{array}\right)$$

Epoch transitions

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

Accessor Functions

$$\mathsf{getIR} \in \mathsf{EpochState} \to (\mathsf{Credential}_{stake} \mapsto \mathsf{Coin}) \quad \text{get instantaneous rewards}$$

Helper Functions

$$\mathsf{votedValue} \in (\mathsf{KeyHash}_{\mathsf{G}} \mapsto \mathsf{PParamsUpdate}) \to \mathsf{PParams} \to \mathbb{N} \to \mathsf{PParamsUpdate}^?$$

$$\mathsf{votedValue}\ pup\ pp\ quorum = \begin{cases} pp \underset{\to}{\cup} p & \exists! p \in \text{range}\ pup\ (|pup \rhd p| \geq quorum) \\ \diamond & \text{otherwise} \end{cases}$$

Figure 44: Epoch transition-system types

The epoch transition rule calls SNAP, POOLREAP and NEWPP in sequence. It also stores the previous protocol parameters in *prevPp*. The previous protocol parameters will be used for the reward calculation in the upcoming epoch, note that they correspond to the epoch for which the rewards are being calculated. Additionally, this transition also adopts the pool parameters *fPoolParams* corresponding to the pool re-registration certificates which we submitted late in the ending epoch. The ordering of these rules is important. The stake pools which will be updated by *fPoolParams* or reaped during the POOLREAP transition must still be a part of the new snapshot, and so SNAP must occur before these two actions. Moreover, SNAP sets the deposit pot equal to current obligation, which is a property that is preserved by POOLREAP and which is necessary for the preservation of Ada property in the NEWPP transition.

$$lstate \vdash ss \xrightarrow[\text{SNAP}]{} ss'$$

$$\begin{gathered}(utxoSt,\ (dstate,\ pstate)) := ls \\ (poolParams,\ fPoolParams,\ retiring) := pstate \\ pstate' := (poolParams \underrightarrow{\cup} fPoolParams,\ \emptyset,\ retiring)\end{gathered}$$

$$pp \vdash \begin{pmatrix} utxoSt \\ acnt \\ dstate \\ pstate' \end{pmatrix} \xrightarrow[\text{POOLREAP}]{e} \begin{pmatrix} utxoSt' \\ acnt' \\ dstate' \\ pstate'' \end{pmatrix}$$

$$\begin{gathered}(\_,\ \_,\ \_,\ (pup, \_)) := utxoSt' \\ pp_{new} := \mathsf{votedValue}\ pup\ pp\ \mathsf{Quorum} \\ dstate' \\ pstate'' \end{gathered} \vdash \begin{pmatrix} utxoSt' \\ acnt' \\ pp \end{pmatrix} \xrightarrow[\text{NEWPP}]{pp_{new}} \begin{pmatrix} utxoSt'' \\ acnt'' \\ pp' \end{pmatrix}$$

$$\text{Epoch}\ \frac{ls' := (utxoSt'',\ (dstate',\ pstate''))}{\vdash \begin{pmatrix} acnt \\ ss \\ ls \\ prevPp \\ pp \end{pmatrix} \xrightarrow[\text{EPOCH}]{e} \begin{pmatrix} \mathbf{acnt''} \\ \mathbf{ss'} \\ \mathbf{ls'} \\ \mathbf{pp} \\ \mathbf{pp'} \end{pmatrix}} \tag{25}$$

Figure 45: Epoch Inference Rule

11.9 Rewards Distribution Calculation

This section defines the reward calculation for the proof of stake leader election. Figure 46 defines the pool reward as described in section 5.5.2 of [SL-D1].

- 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 mkApparentPerformance computes the apparent performance of a stake pool. It depends on the protocol parameter d, the relative stake σ, the number n of blocks the pool added to the chain and the total number $\overline{N}$ of blocks added to the chain in the last epoch.

Maximal Reward Function, called $f(s, \sigma)$ in section 5.5.2 of [SL-D1]

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

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

where

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

Apparent Performance, called $\hat{p}$ in section 5.5.2 of [SL-D1]

$$\mathsf{mkApparentPerformance} \in [0,\ 1] \to [0,\ 1] \to \mathbb{N} \to \mathbb{N} \to \mathbb{Q}$$

$$\mathsf{mkApparentPerformance}\ d\ \sigma\ n\ \overline{N} = \begin{cases} \frac{\beta}{\sigma} & \text{if } d < 0.8 \\ 1 & \text{otherwise} \end{cases}$$

where

$$\beta = \frac{n}{\max(1, \overline{N})}$$

Figure 46: Functions used in the Reward Calculation

Figure 47 gives the calculation for splitting the pool rewards with its members, as described in 6.5.2 of [SL-D1]. The portion of rewards allocated to the pool operator and owners is different than that of the members.

- The $\mathsf{r}_{\mathsf{operator}}$ function calculates the leader reward, based on the pool cost, margin and the 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 5.5.3 of [SL-D1]

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

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

where

$$c = \mathsf{poolCost}\ \mathit{pool}$$

$$m = \mathsf{poolMargin}\ \mathit{pool}$$

Pool member reward, from section 5.5.3 of [SL-D1]

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

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

where

$$c = \mathsf{poolCost}\ \mathit{pool}$$

$$m = \mathsf{poolMargin}\ \mathit{pool}$$

Figure 47: Functions used in the Reward Splitting

Finally, the full reward calculation is presented in Figure 48. 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 credential of the pool operator. The function returns the rewards, calculated as follows:
  - *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 accounts 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.
- The reward function applies rewardOnePool to each registered stake pool.

Calculation to reward a single stake pool

$$\begin{aligned}
&\mathsf{rewardOnePool} \in \mathsf{PParams} \to \mathsf{Coin} \to \mathbb{N} \to \mathbb{N} \to \mathsf{PoolParam} \\
&\quad \to \mathsf{Stake} \to \mathbb{Q} \to \mathbb{Q} \to \mathsf{Coin} \to \mathbb{P}\ \mathsf{Addr}_{\mathsf{rwd}} \to (\mathsf{Addr}_{\mathsf{rwd}} \mapsto \mathsf{Coin}) \\
&\mathsf{rewardOnePool}\ pp\ R\ n\ \overline{N}\ pool\ stake\ \sigma\ \sigma_a\ tot\ addrs_{rew} = rewards \\
&\quad \textbf{where} \\
&\quad ostake = \sum_{\substack{hk_i \to c \in stake \\ hk \in (\mathsf{poolOwners}\ pool)}} c \\
&\quad pledge = \mathsf{poolPledge}\ pool \\
&\quad p_r = pledge / tot \\
&\quad maxP = \begin{cases} \mathsf{maxPool}\ pp\ R\ \sigma\ p_r & pledge \leq ostake \\ 0 & \text{otherwise.} \end{cases} \\
&\quad appPerf = \mathsf{mkApparentPerformance}\ (\mathsf{d}\ pp)\ \sigma_a\ n\ \overline{N} \\
&\quad poolR = \lfloor appPerf \cdot maxP \rfloor \\
&\quad mRewards = \\
&\qquad \left\{ \mathsf{addr}_{\mathsf{rwd}}\ \ hk \mapsto \mathsf{r}_{\mathsf{member}}\ poolR\ pool\ \frac{c}{tot}\ \sigma \;\middle|\; hk \mapsto c \in stake,\ \ hk \notin (\mathsf{poolOwners}\ pool) \right\} \\
&\quad lReward = \mathsf{r}_{\mathsf{operator}}\ poolR\ pool\ \frac{ostake}{tot}\ \sigma \\
&\quad potentialRewards = mRewards \cup \{(\mathsf{poolRAcnt}\ pool) \mapsto lReward\} \\
&\quad rewards = addrs_{rew} \lhd potentialRewards
\end{aligned}$$

Calculation to reward all stake pools

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

Figure 48: The Reward Calculation

11.10 Reward Update Calculation

This section defines the calculation of a reward update. A reward update is the information needed to account for the movement of lovelace in the system due to paying out rewards.

Figure 49 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 "reward pot", a temporary variable used during the reward update calculation in Figure 51. The blue arrows represent the movement of funds that pass through the "reward pot".

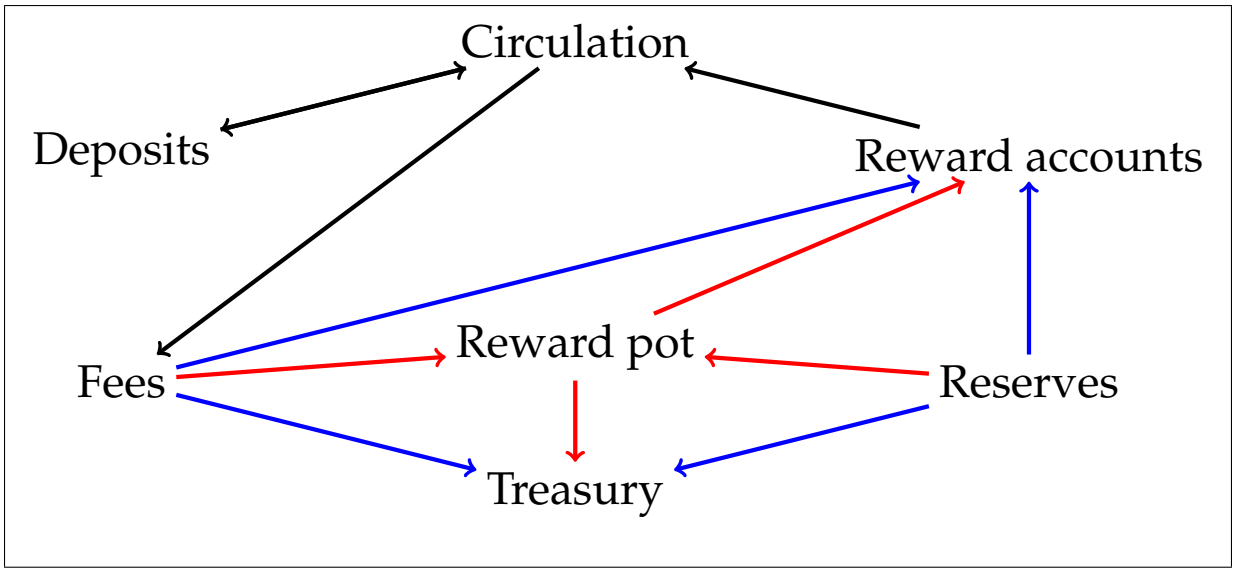


Figure 49: Preservation of Value

Figure 50 defines a reward update. It consists of four pots:

- The change to the treasury. This will be a positive value.
- The change to the reserves. This will be a negative value.
- The map of new individual rewards (to be added to the existing rewards).
- The change to the fee pot. This will be a negative value. rewards.

Reward Update

$$\mathsf{RewardUpdate} = \left(\begin{array}{rcll} \Delta t & \in & \mathsf{Coin} & \text{change to the treasury} \\ \Delta r & \in & \mathsf{Coin} & \text{change to the reserves} \\ rs & \in & \mathsf{Addr}_{\mathsf{rwd}} \mapsto \mathsf{Coin} & \text{new individual rewards} \\ \Delta f & \in & \mathsf{Coin} & \text{change to the fee pot} \end{array} \right)$$

Figure 50: Rewards Update type

Figure 51 defines two functions, createRUpd to create a reward update and applyRUpd to apply a reward update to an instance of EpochState.

The createRUpd function does the following:

- Note that for all the calculations below, we use the previous protocol parameters *prevPp*, which corresponds to the parameters during the epoch for which we are creating rewards.
- First we calculate the change to the reserves, as determined by the ρ protocol parameter.
- Next we calculate *rewardPot*, the total amount of coin available for rewards this epoch, as described in section 6.4 of [SL-D1]. It consists of:
  - The fee pot, containing the transaction fees from the epoch.
  - The amount of monetary expansion from the reserves, calculated above.

  Note that the fee pot is taken from the snapshot taken at the epoch boundary. (See Figure39).
- Next we calculate the proportion of the reward pot that will move to the treasury, as determined by the τ protocol parameter. The remaining pot is called the R, just as in section 6.5 of [SL-D1].
- 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 added to the amount moved to the treasury.
- The fee pot will be 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.

The applyRUpd function does the following:

- Adjust the treasury, reserves and fee pots by the appropriate amounts.
- Add each individual reward to the global reward mapping. We must be careful, though, not to give out rewards to accounts that have been deregistered after the reward update was created.
  - Rewards for accounts that are still registered are added to the reward mappings.
  - The sum of the unregistered rewards are added to the reserves.

These two functions will be used in the blockchain transition systems in Section 12. In particular, createRUpd will be used in Equation 34, and applyRUpd will be used in Equation 27.

Calculation to create a reward update

$$\mathsf{createRUpd} \in \mathbb{N} \to \mathsf{BlocksMade} \to \mathsf{EpochState} \to \mathsf{Coin} \to \mathsf{RewardUpdate}$$

$$\mathsf{createRUpd}\ \mathit{slotsPerEpoch}\ b\ \mathit{es}\ \mathit{total} = (\Delta t_1, -\Delta r_1 + \Delta r_2,\ \mathit{rs},\ -\mathit{feeSS})$$

where

$$(\mathit{acnt},\ \mathit{ss},\ \mathit{ls},\ \mathit{prevPp},\ \_) = \mathit{es}$$

$$(\_,\ \_,\ \mathit{pstake}_{go},\ \mathit{feeSS}) = \mathit{ss}$$

$$(\mathit{stake},\ \mathit{delegs},\ \mathit{poolParams}) = \mathit{pstate}_{go}$$

$$(\_,\ \mathit{reserves}) = \mathit{acnt}$$

$$(\_,\ ((\mathit{rewards},\ \_,\ \_,\ \_,\ \_,\ \_)\ ,\ \_)) = \mathit{ls}$$

$$\Delta r_1 = \lfloor \min(1, \eta) \cdot (\mathsf{rho}\ \mathit{prevPp}) \cdot \mathit{reserves} \rfloor$$

$$\eta = \begin{cases} 1 & (\mathsf{d}\ \mathit{prevPp}) \geq 0.8 \\ \frac{\mathit{blocksMade}}{\lceil (1 - \mathsf{d}\ \mathit{prevPp}) \cdot \mathit{slotsPerEpoch} \cdot \mathsf{ActiveSlotCoeff} \rceil} & \text{otherwise} \end{cases}$$

$$\mathit{rewardPot} = \mathit{feeSS} + \Delta r_1$$

$$\Delta t_1 = \lfloor (\mathsf{tau}\ \mathit{prevPp}) \cdot \mathit{rewardPot} \rfloor$$

$$R = \mathit{rewardPot} - \Delta t_1$$

$$\mathit{circulation} = \mathit{total} - \mathit{reserves}$$

$$\mathit{rs} = \mathsf{reward}\ \mathit{prevPp}\ b\ R\ (\mathrm{dom}\ \mathit{rewards})\ \mathit{poolParams}\ \mathit{stake}\ \mathit{delegs}\ \mathit{circulation}$$

$$\Delta r_2 = R - \left(\sum_{\_ \mapsto c \in \mathit{rs}} c \right)$$

$$\mathit{blocksMade} = \sum_{\_ \mapsto m \in b} m$$

Figure 51: Reward Update Creation

Applying a reward update

$$\mathsf{applyRUpd} \in \mathsf{RewardUpdate} \to \mathsf{EpochState} \to \mathsf{EpochState}$$

$$\mathsf{applyRUpd}\begin{pmatrix}\Delta t\\ \Delta r\\ rs\\ \Delta f\end{pmatrix}\begin{pmatrix}treasury\\ reserves\\ \\ rewards\\ delegations\\ ptrs\\ genDelegs\\ fGenDelegs\\ i_{rwd}\\ \\ poolParams\\ fPoolParams\\ retiring\\ \\ utxo\\ deposited\\ fees\\ up\\ \\ prevPp\\ pp\end{pmatrix} = \begin{pmatrix}treasury + \Delta t + unregRU'\\ reserves + \Delta r\\ \\ rewards \cup_{+} regRU\\ delegations\\ ptrs\\ genDelegs\\ fGenDelegs\\ i_{rwd}\\ \\ poolParams\\ fPoolParams\\ retiring\\ \\ utxo\\ deposited\\ fees + \Delta f\\ up\\ \\ prevPp\\ pp\end{pmatrix}$$

where

$$regRU = (\mathrm{dom}\, rewards) \lhd rs$$

$$unregRU = (\mathrm{dom}\, rewards) \not\lhd rs$$

$$unregRU' = \sum_{\_ \mapsto c \in unregRU} c$$

Figure 52: Reward Update Application

12 Blockchain layer

This chapter introduces the view of the blockchain layer as required for the ledger. This includes in particular the information required for the epoch boundary and its rewards calculation as described in Section 11. It also covers the transitions that keep track of produced blocks in order to calculate rewards and penalties for stake pools.

The main transition rule is CHAIN which calls the subrules NEWEPOCH and UPDN, VRF and BBODY.

12.1 Block Definitions

Abstract types

$$\begin{array}{rcll} h & \in & \mathsf{HashHeader} & \text{hash of a block header} \\ hb & \in & \mathsf{HashBBody} & \text{hash of a block body} \\ bn & \in & \mathsf{BlockNo} & \text{block number} \end{array}$$

Operational Certificate

$$\mathsf{OCert} = \left(\begin{array}{rcll} vk_{hot} & \in & \mathsf{VKey_{ev}} & \text{operational (hot) key} \\ n & \in & \mathbb{N} & \text{certificate issue number} \\ c_0 & \in & \mathsf{KESPeriod} & \text{start KES period} \\ \sigma & \in & \mathsf{Sig} & \text{cold key signature} \end{array}\right)$$

Block Header Body

$$\mathsf{BHBody} = \left(\begin{array}{rcll} prev & \in & \mathsf{HashHeader}^? & \text{hash of previous block header} \\ vk & \in & \mathsf{VKey} & \text{block issuer} \\ vrfVk & \in & \mathsf{VKey} & \text{VRF verification key} \\ blockno & \in & \mathsf{BlockNo} & \text{block number} \\ slot & \in & \mathsf{Slot} & \text{block slot} \\ \eta & \in & \mathsf{Seed} & \text{nonce} \\ prf_\eta & \in & \mathsf{Proof} & \text{nonce proof} \\ \ell & \in & [0,\,1] & \text{leader election value} \\ prf_\ell & \in & \mathsf{Proof} & \text{leader election proof} \\ bsize & \in & \mathbb{N} & \text{size of the block body} \\ bhash & \in & \mathsf{HashBBody} & \text{block body hash} \\ oc & \in & \mathsf{OCert} & \text{operational certificate} \\ pv & \in & \mathsf{ProtVer} & \text{protocol version} \end{array}\right)$$

Block Types

$$\begin{array}{rcll} bh & \in & \mathsf{BHeader} & = & \mathsf{BHBody} \times \mathsf{Sig} \\ b & \in & \mathsf{Block} & = & \mathsf{BHeader} \times \mathsf{Tx}^* \end{array}$$

Abstract functions

$$\begin{array}{rcll} \mathsf{bhHash} & \in & \mathsf{BHeader} \to \mathsf{HashHeader} & \text{hash of a block header} \\ \mathsf{bHeaderSize} & \in & \mathsf{BHeader} \to \mathbb{N} & \text{size of a block header} \\ \mathsf{bBodySize} & \in & \mathsf{Tx}^* \to \mathbb{N} & \text{size of a block body} \\ \mathsf{slotToSeed} & \in & \mathsf{Slot} \to \mathsf{Seed} & \text{convert a slot to a seed} \\ \mathsf{prevHashToNonce} & \in & \mathsf{HashHeader}^? \to \mathsf{Seed} & \text{convert an optional header hash to a seed} \\ \mathsf{bbodyhash} & \in & \mathsf{Tx}^* \to \mathsf{HashBBody} & \end{array}$$

Accessor Functions

$$\begin{array}{rcl@{\qquad}rcl} \mathsf{bheader} & \in & \mathsf{Block} \to \mathsf{BHeader} & \mathsf{bhbody} & \in & \mathsf{BHeader} \to \mathsf{BHBody} \\ \mathsf{hsig} & \in & \mathsf{BHeader} \to \mathsf{Sig} & \mathsf{bbody} & \in & \mathsf{Block} \to \mathsf{Tx}^* \\ \mathsf{bvkcold} & \in & \mathsf{BHBody} \to \mathsf{VKey} & \mathsf{bvkvrf} & \in & \mathsf{BHBody} \to \mathsf{VKey} \\ \mathsf{bprev} & \in & \mathsf{BHBody} \to \mathsf{HashHeader}^? & \mathsf{bslot} & \in & \mathsf{BHBody} \to \mathsf{Slot} \\ \mathsf{bblockno} & \in & \mathsf{BHBody} \to \mathsf{BlockNo} & \mathsf{bnonce} & \in & \mathsf{BHBody} \to \mathsf{Seed} \\ \mathsf{bprf_n} & \in & \mathsf{BHBody} \to \mathsf{Proof} & \mathsf{bleader} & \in & \mathsf{BHBody} \to \mathbb{N} \\ \mathsf{bprf}_\ell & \in & \mathsf{BHBody} \to \mathsf{Proof} & \mathsf{hBbsize} & \in & \mathsf{BHBody} \to \mathbb{N} \\ \mathsf{bhash} & \in & \mathsf{BHBody} \to \mathsf{HashBBody} & \mathsf{bocert} & \in & \mathsf{BHBody} \to \mathsf{OCert} \end{array}$$

Figure 53: Block Definitions

MIR Transitions

$$\vdash \_ \xrightarrow[\text{MIR}]{} \_ \subseteq \mathbb{P}\ (\mathsf{EpochState} \times \mathsf{EpochState})$$

Figure 54: MIR transition-system types

12.2 MIR Transition

The transition which moves the instantaneous rewards is MIR. Figure 54 defines the types for the transition. It has no environment or signal, and the state is EpochState.

Figure 55 defines the MIR state transition.

If the reserve and treasury pots are large enough to cover the sum of the corresponding instantaneous rewards, the reward accounts are increased by the appropriate amount and the two pots are decreased appropriately. In either case, if the pots are large enough or not, we reset both of the instantaneous reward mappings back to the empty mapping.

$$\frac{\begin{array}{c}(rewards,\ delegations,\ ptrs,\ fGenDelegs,\ genDelegs,\ i_{rwd}) := ds \\ (treasury,\ reserves) := acnt \qquad (irReserves,\ irTreasury) := i_{rwd} \\ \\ irwdR := \{\mathsf{addr}_{\mathsf{rwd}}\ hk \mapsto val \mid hk \mapsto val \in (\mathrm{dom}\ rewards) \lhd irReserves\} \\ irwdT := \{\mathsf{addr}_{\mathsf{rwd}}\ hk \mapsto val \mid hk \mapsto val \in (\mathrm{dom}\ rewards) \lhd irTreasury\} \\ \\ totR := \sum_{\_ \mapsto v \in irwdR} v \qquad totT := \sum_{\_ \mapsto v \in irwdT} v \\ enough := totR \leq reserves \wedge totT \leq treasury \\ acnt' := \begin{cases} (treasury - totT,\ reserves - totR) & enough \\ acnt & \text{otherwise} \end{cases} \\ rewards' := \begin{cases} rewards \cup_+ irwdR \cup_+ irwdT & enough \\ rewards & \text{otherwise} \end{cases} \\ ds' := (\mathbf{rewards'},\ delegations,\ ptrs,\ fGenDelegs,\ genDelegs,\ (\emptyset,\ \emptyset)) \end{array}}{\vdash \begin{pmatrix} acnt \\ ss \\ (us,\ (ds,\ ps)) \\ prevPP \\ pp \end{pmatrix} \xrightarrow[\text{MIR}]{} \begin{pmatrix} \mathbf{acnt'} \\ ss \\ (us,\ (\mathbf{ds'},\ ps)) \\ prevPP \\ pp \end{pmatrix}}\ \text{MIR} \tag{26}$$

Figure 55: MIR rules

12.3 New Epoch Transition

For the transition to a new epoch (NEWEPOCH), the environment is given in Figure 56, it consists of

- The current slot.
- The set of genesis keys.

The new epoch state is given in Figure 56, it consists of

- The number of the last epoch.
- The information about produced blocks for each stake pool during the previous epoch.

New Epoch states

$$\mathsf{NewEpochState} = \left(\begin{array}{rcll} e_\ell & \in & \mathsf{Epoch} & \text{last epoch} \\ b_{prev} & \in & \mathsf{BlocksMade} & \text{blocks made last epoch} \\ b_{cur} & \in & \mathsf{BlocksMade} & \text{blocks made this epoch} \\ es & \in & \mathsf{EpochState} & \text{epoch state} \\ ru & \in & \mathsf{RewardUpdate}^? & \text{reward update} \\ pd & \in & \mathsf{PoolDistr} & \text{pool stake distribution} \end{array} \right)$$

New Epoch Transitions

$$\vdash \_ \xrightarrow[\text{NEWEPOCH}]{} \_ \subseteq \mathbb{P}\ (\mathsf{NewEpochState} \times \mathsf{Epoch} \times \mathsf{NewEpochState})$$

Helper function

$$\begin{aligned} &\mathsf{calculatePoolDistr} \in \mathsf{Snapshot} \to \mathsf{PoolDistr} \\ &\mathsf{calculatePoolDistr}\ (\mathit{stake},\ \mathit{delegs},\ \mathit{poolParams}) = \\ &\left\{ hk_p \mapsto (\sigma,\ \mathsf{poolVRF}\ p) \;\middle|\; \begin{array}{l} hk_p \mapsto \sigma \in \mathit{sd} \\ hk_p \mapsto p \in \mathit{poolParams} \end{array} \right\} \\ &\quad \textbf{where} \\ &\quad\quad \mathit{total} = \sum_{\_ \mapsto c \in \mathit{stake}} c \\ &\quad\quad \mathit{sd} = \mathsf{aggregate}_+ \left(\mathit{delegs}^{-1} \circ \left\{ \left(hk, \frac{c}{\mathit{total}} \right) \mid (hk, c) \in \mathit{stake} \right\} \right) \end{aligned}$$

Figure 56: NewEpoch transition-system types

- The information about produced blocks for each stake pool during the current epoch.
- The old epoch state.
- An optional rewards update.
- The stake pool distribution of the epoch.

Figure 57 defines the new epoch state transition. It has three rules. The first rule describes the change in the case of e being equal to the next epoch $e_\ell + 1$. It also calls the MIR and EPOCH rules and checks that the reward update is net neutral with respect to the Ada in the system. This should always hold (by the definition of the createRUpd function) and is present only for extra assurance and for help in proving that Ada is preserved by this transition. The second rule deals with the case when the epoch signal e is not one greater than the current epoch e_ℓ. This rule does not change the state. The third rule is nearly the same as the first rule, only there is no reward update to apply. This third rule is defined for completeness, but in practice we hope that it is never used, since it only applies in the case that epoch e processed no blocks after the first StabilityWindow-many slots.

In the first case, the new epoch state is updated as follows:

- The epoch is set to the new epoch e.
- The mapping for the blocks produced by each stake pool for the previous epoch is set to the current such mapping.

- The mapping for the blocks produced by each stake pool for the current epoch is set to the empty map.
- The epoch state is updated with: first applying the rewards update ru, then calling the MIR transition, and finally by calling the EPOCH transition.
- The rewards update is set to $\diamond$.
- The new pool distribution pd' is calculated from the delegation map and stake allocation of the previous epoch.

$$\text{New-Epoch}\ \frac{\begin{array}{c} e = e_\ell + 1 \qquad ru \neq \diamond \qquad (\Delta t,\ \Delta r,\ rs,\ \Delta f) := ru \\ \Delta t + \Delta r + \left(\sum_{\_ \mapsto v \in rs} v\right) + \Delta f = 0 \\ es' := \mathsf{applyRUpd}\ ru\ es \qquad \vdash es' \xrightarrow[\text{MIR}]{} es'' \qquad \vdash es'' \xrightarrow[\text{EPOCH}]{e} es''' \\ \begin{array}{rcl} (acnt,\ ss,\ \_,\ \_,\ \_) & := & es''' \\ (\_,\ pstake_{set},\ \_,\ \_) & := & ss \\ pd' & := & \mathsf{calculatePoolDistr}\ pstake_{set} \end{array} \end{array}}{\vdash \begin{pmatrix} e_\ell \\ b_{prev} \\ b_{cur} \\ es \\ ru \\ pd \end{pmatrix} \xrightarrow[\text{NEWEPOCH}]{e} \begin{pmatrix} e \\ b_{cur} \\ \emptyset \\ es''' \\ \diamond \\ pd' \end{pmatrix}} \tag{27}$$

$$\text{Not-New-Epoch}\ \frac{(e_\ell,\ \_,\ \_,\ \_,\ \_,\ \_,\ \_) := nes \qquad e \neq e_\ell + 1}{\vdash nes \xrightarrow[\text{NEWEPOCH}]{e} nes} \tag{28}$$

$$\text{No-Reward-Update}\ \frac{\begin{array}{c} (e_\ell,\ \_,\ \_,\ \_,\ ru,\ \_,\ \_) := nes \qquad e = e_\ell + 1 \qquad ru = \diamond \\ \vdash es \xrightarrow[\text{MIR}]{} es'' \qquad \vdash es'' \xrightarrow[\text{EPOCH}]{e} es''' \\ \begin{array}{rcl} (acnt,\ ss,\ \_,\ \_,\ \_) & := & es''' \\ (\_,\ pstake_{set},\ \_,\ \_) & := & ss \\ pd' & := & \mathsf{calculatePoolDistr}\ pstake_{set} \end{array} \end{array}}{\vdash \begin{pmatrix} e_\ell \\ b_{prev} \\ b_{cur} \\ es \\ ru \\ pd \end{pmatrix} \xrightarrow[\text{NEWEPOCH}]{e} \begin{pmatrix} e \\ b_{cur} \\ \emptyset \\ es''' \\ ru \\ pd' \end{pmatrix}} \tag{29}$$

Figure 57: New Epoch rules

Tick Nonce environments

$$\mathsf{TickNonceEnv} = \begin{pmatrix} pp & \in & \mathsf{PParams} & \text{protocol parameters} \\ \eta_c & \in & \mathsf{Seed} & \text{candidate nonce} \\ \eta_{ph} & \in & \mathsf{Seed} & \text{previous header hash as nonce} \end{pmatrix}$$

Tick Nonce states

$$\mathsf{TickNonceState} = \begin{pmatrix} \eta_0 & \in & \mathsf{Seed} & \text{epoch nonce} \\ \eta_h & \in & \mathsf{Seed} & \text{seed generated from hash of previous epoch's last block header} \end{pmatrix}$$

12.4 Tick Nonce Transition

The Tick Nonce Transition is responsible for updating the epoch nonce and the previous epoch's hash nonce at the start of an epoch. Its environment is shown in Figure 12.4 and consists of the protocol parameters pp, the candidate nonce η_c and the previous epoch's last block header hash as a nonce. Its state consists of the epoch nonce η_0 and the previous epoch's last block header hash nonce.

The signal to the transition rule TICKN is a marker indicating whether we are in a new epoch. If we are in a new epoch, we update the epoch nonce and the previous hash. Otherwise, we do nothing.

$$\text{Not-New-Epoch}\ \frac{}{\begin{matrix} pp \\ \eta_c \\ \eta_{ph} \end{matrix} \vdash \begin{pmatrix} \eta_0 \\ \eta_h \end{pmatrix} \xrightarrow[\text{TICKN}]{\mathsf{False}} \begin{pmatrix} \eta_0 \\ \eta_h \end{pmatrix}} \tag{30}$$

$$\text{New-Epoch}\ \frac{\eta_e := \mathsf{extraEntropy}\ pp}{\begin{matrix} pp \\ \eta_c \\ \eta_{ph} \end{matrix} \vdash \begin{pmatrix} \eta_0 \\ \eta_h \end{pmatrix} \xrightarrow[\text{TICKN}]{\mathsf{True}} \begin{pmatrix} \eta_c \star \eta_h \star \eta_e \\ \eta_{ph} \end{pmatrix}} \tag{31}$$

Figure 58: Tick Nonce rules

12.5 Update Nonce Transition

The Update Nonce Transition updates the nonces until the randomness gets fixed. The environment is shown in Figure 59 and consists of the block nonce η. The update nonce state is shown in Figure 59 and consists of the candidate nonce η_c and the evolving nonce η_v.

The transition rule UPDN takes the slot s as signal. There are two different cases for UPDN: one where s is not yet StabilityWindow slots from the beginning of the next epoch and one where s is less than StabilityWindow slots until the start of the next epoch.

Note that in 32, the nonce candidate η_c transitions to $\eta_v \star \eta$, not $\eta_c \star \eta$. The reason for this is that even though the nonce candidate is frozen sometime during the epoch, we want the two nonces to again be equal at the start of a new epoch (so that the entropy added near the end of the epoch is not discarded).

Update Nonce environments

$$\mathsf{UpdateNonceEnv} = \left(\begin{array}{lll} \eta & \in \mathsf{Seed} & \text{new nonce} \end{array} \right)$$

Update Nonce states

$$\mathsf{UpdateNonceState} = \left(\begin{array}{lll} \eta_v & \in \mathsf{Seed} & \text{evolving nonce} \\ \eta_c & \in \mathsf{Seed} & \text{candidate nonce} \end{array} \right)$$

Update Nonce Transitions

$$\_ \vdash \_ \xrightarrow[\text{UPDN}]{\_} \_ \subseteq \mathbb{P}\left(\mathsf{UpdateNonceEnv} \times \mathsf{UpdateNonceState} \times \mathsf{Slot} \times \mathsf{UpdateNonceState}\right)$$

Figure 59: UpdNonce transition-system types

$$\text{Update-Both}\ \frac{s < \mathsf{firstSlot}\left((\mathsf{epoch}\ s) + 1\right) - \mathsf{StabilityWindow}}{\eta \ \vdash \begin{pmatrix} \eta_v \\ \eta_c \end{pmatrix} \xrightarrow[\text{UPDN}]{s} \begin{pmatrix} \eta_v \star \eta \\ \eta_v \star \eta \end{pmatrix}} \tag{32}$$

$$\text{Only-Evolve}\ \frac{s \geq \mathsf{firstSlot}\left((\mathsf{epoch}\ s) + 1\right) - \mathsf{StabilityWindow}}{\eta \ \vdash \begin{pmatrix} \eta_v \\ \eta_c \end{pmatrix} \xrightarrow[\text{UPDN}]{s} \begin{pmatrix} \eta_v \star \eta \\ \eta_c \end{pmatrix}} \tag{33}$$

Figure 60: Update Nonce rules

12.6 Reward Update Transition

The Reward Update Transition calculates a new RewardUpdate to apply in a NEWEPOCH transition. The environment is shown in Figure 61, it consists of the produced blocks mapping b and the epoch state es. Its state is an optional reward update.

The transition rules are shown in Figure 62. There are three cases, one which computes a new reward update, one which leaves the rewards update unchanged as it has not yet been applied and finally one that leaves the reward update unchanged as the transition was started too early.

The signal of the transition rule RUPD is the slot s. The execution of the transition role is as follows:

Reward Update environments

$$\mathsf{RUpdEnv} = \left(\begin{array}{lll} b & \in \mathsf{BlocksMade} & \text{blocks made} \\ es & \in \mathsf{EpochState} & \text{epoch state} \end{array} \right)$$

Reward Update Transitions

$$\_ \vdash \_ \xrightarrow[\text{RUPD}]{\_} \_ \subseteq \mathbb{P}\left(\mathsf{RUpdEnv} \times \mathsf{RewardUpdate}^? \times \mathsf{Slot} \times \mathsf{RewardUpdate}^?\right)$$

Figure 61: Reward Update transition-system types

- If the current reward update ru is empty and s is greater than the sum of the first slot of its epoch and the duration RandomnessStabilisationWindow, then a new rewards update is calculated and the state is updated. (Note the errata in Section 17.3.)
- If the current reward update ru is not $\diamond$, i.e., a reward update has already been calculated but not yet applied, then the state is not updated.
- If the current reward update ru is empty and s is less than or equal to the sum of the first slot of its epoch and the duration to start rewards RandomnessStabilisationWindow, then the state is not updated.

$$\text{Create-Reward-Update}\frac{s > \mathsf{firstSlot}\ (\mathsf{epoch}\ s) + \mathsf{RandomnessStabilisationWindow} \qquad ru = \diamond \qquad ru' := \mathsf{createRUpd\ SlotsPerEpoch}\ b\ es\ \mathsf{MaxLovelaceSupply}}{\begin{matrix} b \\ es \end{matrix} \vdash ru \xrightarrow[\text{RUPD}]{s} ru'} \tag{34}$$

$$\text{Reward-Update-Exists}\frac{ru \neq \diamond}{\begin{matrix} b \\ es \end{matrix} \vdash ru \xrightarrow[\text{RUPD}]{s} ru} \tag{35}$$

$$\text{Reward-Too-Early}\frac{ru = \diamond \qquad s \leq \mathsf{firstSlot}\ (\mathsf{epoch}\ s) + \mathsf{RandomnessStabilisationWindow}}{\begin{matrix} b \\ es \end{matrix} \vdash ru \xrightarrow[\text{RUPD}]{s} ru} \tag{36}$$

Figure 62: Reward Update rules

12.7 Chain Tick Transition

The Chain Tick Transition performs some chain level upkeep. The environment consists of a set of genesis keys, and the state is the epoch specific state necessary for the NEWEPOCH transition.

Part of the upkeep is updating the genesis key delegation mapping according to the future delegation mapping. For each genesis key, we adopt the most recent delegation in *fGenDelegs* that is past the current slot, and any future genesis key delegations past the current slot is removed. The helper function adoptGenesisDelegs accomplishes the update.

The TICK transition rule is shown in Figure 64. The signal is a slot s.

Three transitions are done:

- The NEWEPOCH transition performs any state change needed if it is the first block of a new epoch.
- The RUPD creates the reward update if it is late enough in the epoch. **Note** that for every block header, either NEWEPOCH or RUPD will be the identity transition, and so, for instance, it does not matter if RUPD uses nes or nes' to obtain the needed state.

Chain Tick Transitions

$$\vdash \_ \xrightarrow[\text{TICK}]{} \_ \subseteq \mathbb{P}\,(\mathsf{NewEpochState} \times \mathsf{Slot} \times \mathsf{NewEpochState})$$

helper function

$\mathsf{adoptGenesisDelegs} \in \mathsf{EpochState} \to \mathsf{Slot} \to \mathit{EpochState}$

$\mathsf{adoptGenesisDelegs}\ es\ slot = es'$

where

$(acnt,\ ss, (us, (ds, ps)),\ prevPp,\ pp) := es$

$(\ rewards,\ delegations,\ ptrs,\ fGenDelegs,\ genDelegs,\ i_{rwd}) := ds$

$curr := \{(s,\ gkh) \mapsto (vkh,\ vrf) \in fGenDelegs \mid s \leq slot\}$

$fGenDelegs' := fGenDelegs \setminus curr$

$$genDelegs' := \left\{ gkh \mapsto (vkh,\ vrf) \;\middle|\; \begin{array}{l} (s,\ gkh) \mapsto (vkh,\ vrf) \in curr \\ s = \max\{s' \mid (s',\ gkh) \in \mathrm{dom}\, curr\} \end{array} \right\}$$

$ds' := (rewards,\ delegations,\ ptrs,\ fGenDelegs',\ genDelegs \underset{\rightarrow}{\cup} genDelegs',\ i_{rwd})$

$es' := (acnt,\ ss, (us, (ds', ps)),\ prevPp,\ pp)$

Figure 63: Tick transition-system types

$$\vdash nes \xrightarrow[\text{NEWEPOCH}]{\mathsf{epoch}\ slot} nes'$$

$$(\_,\ b_{prev},\ \_,\ es,\ \_,\ \_) := nes$$

$$\begin{matrix} b_{prev} \\ es \end{matrix} \vdash ru' \xrightarrow[\text{RUPD}]{slot} ru''$$

$$\begin{array}{c} (e'_\ell,\ b'_{prev},\ b'_{cur},\ es',\ ru',\ pd') := nes' \\ es'' := \mathsf{adoptGenesisDelegs}\ es'\ slot \\ nes'' := (e'_\ell,\ b'_{prev},\ b'_{cur},\ es'',\ ru'',\ pd') \end{array}$$

$$\text{Tick}\ \frac{}{\vdash nes \xrightarrow[\text{TICK}]{slot} \mathit{nes''}} \tag{37}$$

Figure 64: Tick rules

Operational Certificate environments

$$\mathsf{OCertEnv} = \begin{pmatrix} \mathit{stpools} & \in & \mathbb{P}\ \mathsf{KeyHash} & \text{stake pools} \\ \mathit{genDelegs} & \in & \mathbb{P}\ \mathsf{KeyHash} & \text{genesis key delegates} \end{pmatrix}$$

Operational Certificate Transitions

$$\_ \vdash \_ \xrightarrow[\text{OCERT}]{} \_ \subseteq \mathbb{P}\ (\mathsf{OCertEnv} \times \mathsf{KeyHash}_{pool} \mapsto \mathbb{N} \times \mathsf{BHeader} \times \mathsf{KeyHash}_{pool} \mapsto \mathbb{N})$$

Operational Certificate helper function

$$\mathsf{currentIssueNo} \in \mathsf{OCertEnv} \to (\mathsf{KeyHash}_{pool} \mapsto \mathbb{N}) \to \mathsf{KeyHash}_{pool} \to \mathbb{N}^?$$

$$\mathsf{currentIssueNo}\ (\mathit{stpools}, \mathit{genDelegs})\ \mathit{cs}\ \mathit{hk} = \begin{cases} \mathit{hk} \mapsto n \in \mathit{cs} & n \\ \mathit{hk} \in \mathit{stpools} & 0 \\ \mathit{hk} \in \mathit{genDelegs} & 0 \\ \text{otherwise} & \diamond \end{cases}$$

Figure 65: OCert transition-system types

12.8 Operational Certificate Transition

The Operational Certificate Transition environment consists of the genesis key delegation map *genDelegs* and the set of stake pools *stpools*. Its state is the mapping of operation certificate issue numbers. Its signal is a block header.

The transition rule is shown in Figure 66. From the block header body *bhb* we first extract the following:

- The operational certificate, consisting of the hot key vk_{hot}, the certificate issue number n, the KES period start c_0 and the cold key signature.
- The cold key vk_{cold}.
- The slot s for the block.
- The number of KES periods that have elapsed since the start period on the certificate.

Using this we verify the preconditions of the operational certificate state transition which are the following:

- The KES period of the slot in the block header body must be greater than or equal to the start value c_0 listed in the operational certificate, and less than MaxKESEvo-many KES periods after c_0. The value of MaxKESEvo is the agreed-upon lifetime of an operational certificate, see [SL-D1].
- *hk* exists as key in the mapping of certificate issues numbers to a KES period m and that period is less than or equal to n.
- The signature τ can be verified with the cold verification key vk_{cold}.
- The KES signature σ can be verified with the hot verification key vk_{hot}.

After this, the transition system updates the operational certificate state by updating the mapping of operational certificates where it overwrites the entry of the key *hk* with the KES period n.

The OCERT rule has six predicate failures:

$$(bhb,\ \sigma) := bh \qquad (vk_{hot},\ n,\ c_0,\ \tau) := \mathsf{bocert}\ bhb \qquad vk_{cold} := \mathsf{bvkcold}\ bhb$$
$$hk := \mathsf{hashKey}\ vk_{cold} \qquad s := \mathsf{bslot}\ bhb \qquad t := \mathsf{kesPeriod}\ s - c_0$$

$$c_0 \leq \mathsf{kesPeriod}\ s < c_0 + \mathsf{MaxKESEvo}$$
$$\mathsf{currentIssueNo}\ oce\ cs\ hk = m \qquad m \leq n$$

$$\text{OCert}\ \frac{\mathcal{V}_{vk_{cold}}[\![(vk_{hot},\ n,\ c_0)]\!]_\tau \qquad \mathcal{V}^{\mathsf{KES}}_{vk_{hot}}[\![bhb]\!]^t_\sigma}{oce \vdash cs \xrightarrow[\text{OCERT}]{bh} cs \underset{\cup}{\rightarrow} \{hk \mapsto n\}} \tag{38}$$

Figure 66: OCert rules

- If the KES period is less than the KES period start in the certificate, there is a *KESBeforeStart* failure.
- If the KES period is greater than or equal to the KES period end (start + MaxKESEvo) in the certificate, there is a *KESAfterEnd* failure.
- If the period counter in the original key hash counter mapping is larger than the period number in the certificate, there is a *CounterTooSmall* failure.
- If the signature of the hot key, KES period number and period start is incorrect, there is an *InvalidSignature* failure.
- If the KES signature using the hot key of the block header body is incorrect, there is an *InvalideKesSignature* failure.
- If there is no entry in the key hash to counter mapping for the cold key, there is a *NoCounterForKeyHash* failure.

12.9 Verifiable Random Function

In this section we define a function vrfChecks which performs all the VRF related checks on a given block header body. In addition to the block header body, the function requires the epoch nonce, the stake distribution (aggregated by pool), and the active slots coefficient from the protocol parameters. The function checks:

- The validity of the proofs for the leader value and the new nonce.
- The verification key is associated with relative stake σ in the stake distribution.
- The bleader value of *bhb* indicates a possible leader for this slot. The function checkLeaderVal is defined in 16.

VRF helper function

$$\mathsf{issuerIDfromBHBody} \in \mathsf{BHBody} \to \mathsf{KeyHash}_{pool}$$
$$\mathsf{issuerIDfromBHBody} = \mathsf{hashKey} \circ \mathsf{bvkcold}$$

$$\mathsf{mkSeed} \in \mathsf{Seed} \to \mathsf{Slot} \to \mathsf{Seed} \to \mathsf{Seed}$$
$$\mathsf{mkSeed}\ \mathit{ucNonce}\ \mathit{slot}\ \eta_0 = \mathit{ucNonce}\ \mathsf{XOR}\ (\mathsf{slotToSeed}\ \mathit{slot} \star \eta_0)$$

$$\mathsf{vrfChecks} \in \mathsf{Seed} \to \mathsf{BHBody} \to \mathsf{Bool}$$
$$\mathsf{vrfChecks}\ \eta_0\ \mathit{bhb} =$$
$$\begin{array}{ll} & \mathsf{verifyVrf}_{\mathsf{Seed}}\ \mathit{vrfK}\ (\mathsf{mkSeed}\ \mathsf{Seed}_\eta\ \mathit{slot}\ \eta_0)\ (\mathsf{bprf}_n\ \mathit{bhb},\ \mathsf{bnonce}\ \mathit{bhb}) \\ \wedge & \mathsf{verifyVrf}_{[0,\,1]}\ \mathit{vrfK}\ (\mathsf{mkSeed}\ \mathsf{Seed}_\ell\ \mathit{slot}\ \eta_0)\ (\mathsf{bprf}_\ell\ \mathit{bhb},\ \mathsf{bleader}\ \mathit{bhb}) \end{array}$$
where
$$\mathit{slot} := \mathsf{bslot}\ \mathit{bhb}$$
$$\mathit{vrfK} := \mathsf{bvkvrf}\ \mathit{bhb}$$

$$\mathsf{praosVrfChecks} \in \mathsf{Seed} \to \mathsf{PoolDistr} \to (0,\,1] \to \mathsf{BHBody} \to \mathsf{Bool}$$
$$\mathsf{praosVrfChecks}\ \eta_0\ \mathit{pd}\ f\ \mathit{bhb} =$$
$$\begin{array}{ll} & \mathit{hk} \mapsto (\sigma,\ \mathit{vrfHK}) \in \mathit{pd} \\ \wedge & \mathit{vrfHK} = \mathsf{hashKey}\ \mathit{vrfK} \\ \wedge & \mathsf{vrfChecks}\ \eta_0\ \mathit{bhb} \\ \wedge & \mathsf{checkLeaderVal}\ (\mathsf{bleader}\ \mathit{bhb})\ \sigma\ f \end{array}$$
where
$$\mathit{hk} := \mathsf{issuerIDfromBHBody}\ \mathit{bhb}$$
$$\mathit{vrfK} := \mathsf{bvkvrf}\ \mathit{bhb}$$

$$\mathsf{pbftVrfChecks} \in \mathsf{KeyHash}_{vrf} \to \mathsf{Seed} \to \mathsf{BHBody} \to \mathsf{Bool}$$
$$\mathsf{pbftVrfChecks}\ \mathit{vrfHK}\ \eta_0\ \ \mathit{bhb} =$$
$$\begin{array}{ll} & \mathit{vrfHK} = \mathsf{hashKey}\ (\mathsf{bvkvrf}\ \mathit{bhb}) \\ \wedge & \mathsf{vrfChecks}\ \eta_0\ \mathit{bhb} \end{array}$$

12.10 Overlay Schedule

The transition from the bootstrap era to a fully decentralized network is explained in section 3.9.2 of [SL-D1]. Key to this transition is a protocol parameter d which controls how many slots are governed by the genesis nodes via OBFT, and which slots are open to any registered stake pool. The transition system introduced in this section, OVERLAY, covers this mechanism.

This transition is responsible for validating the protocol for both the OBFT blocks and the Praos blocks, depending on the overlay schedule.

The actual overlay schedule itself it determined by two functions, isOverlaySlot and classifyOverlaySlot, which are defined in Figure 68. The function isOverlaySlot determines if the current slot is reserved for the OBFT nodes. In particular, it looks at the current relative slot to see if the next multiple of d (the decentralization protocol parameter) raises the ceiling. If a slot is indeed in the overlay schedule, the function classifyOverlaySlot determines if the current slot should be a silent slot (no block is allowed) or which core node is responsible for the block. The non-silent blocks are the multiples of $1/\mathsf{ActiveSlotCoeff}$, and the responsible core node are determined by taking turns according the lexicographic ordering of the core node keyhashes. **Note** that $1/\mathsf{ActiveSlotCoeff}$ needs to be natural number, otherwise the multilples of $\lfloor 1/\mathsf{ActiveSlotCoeff} \rfloor$ will yield a different proportion of active slots than the Praos blocks.

The environments for the overlay schedule transition are:

- The decentralization parameter d.
- The epoch nonce η_0.
- The stake pool stake distribution pd.
- The mapping $genDelegs$ of genesis keys to their cold keys and vrf keys.

The states for this transition consist only of the mapping of certificate issue numbers.

This transition establishes that a block producer is in fact authorized. Since there are three key pairs involved (cold keys, VRF keys, and hot KES keys) it is worth examining the interaction closely. First we look at the regular Praos/decentralized setting, which is given by Equation 40.

- First we check the operational certificate with OCERT. This uses the cold verification key given in the block header. We do not yet trust that this key is a registered pool key. If this transition is successful, we know that the cold key in the block header has authorized the block.
- Next, in the vrfChecks predicate, we check that the hash of this cold key is in the mapping pd, and that it maps to $(\sigma,\ hk_{vrf})$, where $(\sigma,\ hk_{vrf})$ is the hash of the VRF key in the header. If praosVrfChecks returns true, then we know that the cold key in the block header was a registered stake pool at the beginning of the previous epoch, and that it is indeed registered with the VRF key listed in the header.
- Finally, we use the VRF verification key in the header, along with the VRF proofs in the header, to check that the operator is allowed to produce the block.

The situation for the overlay schedule, given by Equation 39, is similar. The difference is that we check the overlay schedule to see what core node is supposed to make a block, and then use the genesis delegation mapping to check the correct cold key hash and vrf key hash.

The OVERLAY rule has nine predicate failures:

- If in the decentralized case the VRF key is not in the pool distribution, there is a *VRFKeyUnknown* failure.

Overlay environments

$$\mathsf{OverlayEnv} = \left(\begin{array}{rcll} d & \in & \{0,\ 1/100,\ 2/100,\ \ldots,\ 1\} & \text{decentralization parameter} \\ pd & \in & \mathsf{PoolDistr} & \text{pool stake distribution} \\ genDelegs & \in & \mathsf{GenesisDelegation} & \text{genesis key delegations} \\ \eta_0 & \in & \mathsf{Seed} & \text{epoch nonce} \end{array} \right)$$

Overlay Transitions

$$\_ \vdash \_ \xrightarrow[\text{OVERLAY}]{} \_ \subseteq \mathbb{P}\ (\mathsf{OverlayEnv} \times (\mathsf{KeyHash}_{pool} \mapsto \mathbb{N}) \times \mathsf{BHeader} \times (\mathsf{KeyHash}_{pool} \mapsto \mathbb{N}))$$

Overlay Schedule helper functions

$$\begin{aligned} &\mathsf{isOverlaySlot} \in \mathsf{Slot} \to [0,\ 1] \to \mathsf{Slot} \to \mathsf{Bool} \\ &\mathsf{isOverlaySlot}\ fSlot\ dval\ slot = \lceil s \cdot dval \rceil < \lceil (s+1) \cdot dval \rceil \\ &\quad \textbf{where}\ \ s := slot\ -_s\ fslot \end{aligned}$$

$$\begin{aligned} &\mathsf{classifyOverlaySlot} \in \mathsf{Slot} \to \mathbb{P}\ \mathsf{KeyHash_G} \to [0,\ 1] \to (0,\ 1] \to \mathsf{Slot} \to \mathsf{Bool} \\ &\mathsf{classifyOverlaySlot}\ fSlot\ gkeys\ dval\ asc\ slot = \\ &\quad \begin{cases} \mathsf{elemAt}\ \left(\frac{position}{ascInv} \bmod |gkeys|\right)\ gkeys & \text{if } position \equiv 0 \pmod{ascInv} \\ \diamond & \text{otherwise} \end{cases} \\ &\quad \textbf{where} \\ &\quad ascInv := \lfloor 1/asc \rfloor \\ &\quad position := \lceil (slot\ -_s\ fSlot) \cdot dval \rceil \\ &\quad \mathsf{elemAt}\ i\ set := i'\text{th lexicographic element of } set \end{aligned}$$

Figure 67: Overlay transition-system types

$$\frac{\begin{array}{c} bhb := \mathsf{bheader}\ bh \qquad vk := \mathsf{bvkcold}\ bhb \qquad vkh := \mathsf{hashKey}\ vk \\ slot := \mathsf{bslot}\ bhb \qquad fSlot := \mathsf{firstSlot}\ (\mathsf{epoch}\ slot) \\ gkh \mapsto (vkh,\ vrfh) \in genDelegs \\ \mathsf{isOverlaySlot}\ fSlot\ (\mathrm{dom}\ genDelegs)\ slot \\ \mathsf{classifyOverlaySlot}\ fSlot\ (\mathrm{dom}\ genDelegs)\ d\ \mathsf{ActiveSlotCoeff}\ slot = ghk \\ \\ \mathsf{pbftVrfChecks}\ vrfh\ \eta_0\ bhb \\ \\ \begin{array}{c}\mathrm{dom}\ pd \\ \mathrm{range}\ genDelegs\end{array} \vdash cs \xrightarrow[\text{OCERT}]{bh} cs' \end{array}}{\begin{array}{c} d \\ \eta_0 \\ pd \\ genDelegs \end{array} \vdash cs \xrightarrow[\text{OVERLAY}]{bh} \mathbf{cs'}}\ \text{Active-OBFT} \tag{39}$$

$$\frac{\begin{array}{c} bhb := \mathsf{bheader}\ bh \qquad slot := \mathsf{bslot}\ bhb \qquad fSlot := \mathsf{firstSlot}\ (\mathsf{epoch}\ slot) \\ \neg\mathsf{isOverlaySlot}\ fSlot\ (\mathrm{dom}\ genDelegs)\ slot \\ \\ \vdash cs \xrightarrow[\text{OCERT}]{bh} cs' \\ \\ \mathsf{praosVrfChecks}\ \eta_0\ pd\ \mathsf{ActiveSlotCoeff}\ bhb \end{array}}{\begin{array}{c} d \\ \eta_0 \\ pd \\ genDelegs \end{array} \vdash cs \xrightarrow[\text{OVERLAY}]{bh} \mathbf{cs'}}\ \text{Decentralized} \tag{40}$$

Figure 68: Overlay rules

- If in the decentralized case the VRF key hash does not match the one listed in the block header, there is a *VRFKeyWrongVRFKey* failure.
- If the VRF generated nonce in the block header does not validate against the VRF certificate, there is a *VRFKeyBadNonce* failure.
- If the VRF generated leader value in the block header does not validate against the VRF certificate, there is a *VRFKeyBadLeaderValue* failure.
- If the VRF generated leader value in the block header is too large compared to the relative stake of the pool, there is a *VRFLeaderValueTooBig* failure.
- In the case of the slot being in the OBFT schedule, but without genesis key (i.e., *Nothing*), there is a *NotActiveSlot* failure.
- In the case of the slot being in the OBFT schedule, if there is a specified genesis key which is not the same key as in the bock header body, there is a *WrongGenesisColdKey* failure.
- In the case of the slot being in the OBFT schedule, if the hash of the VRF key in block header does not match the hash in the genesis delegation mapping, there is a *WrongGenesisVRFKey* failure.

- In the case of the slot being in the OBFT schedule, if the genesis delegate keyhash is not in the genesis delegation mapping, there is a *UnknownGenesisKey* failure. This case should never happen, and represents a logic error.

Protocol environments

$$\mathsf{PrtclEnv} = \left(\begin{array}{rcll} d & \in & \{0,\ 1/100,\ 2/100,\ \ldots,\ 1\} & \text{decentralization parameter} \\ pd & \in & \mathsf{PoolDistr} & \text{pool stake distribution} \\ dms & \in & \mathsf{KeyHash}_{\mathsf{G}} \mapsto \mathsf{KeyHash} & \text{genesis key delegations} \\ \eta_0 & \in & \mathsf{Seed} & \text{epoch nonce} \end{array} \right)$$

Protocol states

$$\mathsf{PrtclState} = \left(\begin{array}{rcll} cs & \in & \mathsf{KeyHash}_{pool} \mapsto \mathbb{N} & \text{operational certificate issues numbers} \\ \eta_v & \in & \mathsf{Seed} & \text{evolving nonce} \\ \eta_c & \in & \mathsf{Seed} & \text{candidate nonce} \end{array} \right)$$

Protocol Transitions

$$\_ \vdash \_ \xrightarrow[\text{PRTCL}]{} \_ \subseteq \mathbb{P}\ (\mathbb{P}\ \mathsf{PrtclEnv} \times \mathsf{PrtclState} \times \mathsf{BHeader} \times \mathsf{PrtclState})$$

Figure 69: Protocol transition-system types

12.11 Protocol Transition

The protocol transition covers the common predicates of OBFT and Praos, and then calls OVERLAY for the particular transitions, followed by the transition to update the evolving and candidate nonces.

The environments for this transition are:

- The decentralization parameter d.
- The stake pool stake distribution pd.
- The mapping dms of genesis keys to their cold keys.
- The epoch nonce η_0.

The states for this transition consists of:

- The operational certificate issue number mapping.
- The evolving nonce.
- The canditate nonce for the next epoch.

The PRTCL rule has no predicate failures, besides those of the two sub-transitons.

$$\eta := \mathsf{bnonce}\ (\mathsf{bhbody}\ bhb)$$

$$\text{PRTCL}\ \frac{\eta \vdash \begin{pmatrix} \eta_v \\ \eta_c \end{pmatrix} \xrightarrow[\text{UPDN}]{slot} \begin{pmatrix} \eta_v' \\ \eta_c' \end{pmatrix} \qquad \begin{matrix} d \\ pd \\ dms \\ \eta_0 \end{matrix} \vdash cs \xrightarrow[\text{OVERLAY}]{bh} cs'}{\begin{matrix} d \\ pd \\ dms \\ \eta_0 \end{matrix} \vdash \begin{pmatrix} cs \\ \eta_v \\ \eta_c \end{pmatrix} \xrightarrow[\text{PRTCL}]{bh} \begin{pmatrix} \boldsymbol{cs'} \\ \boldsymbol{\eta_v'} \\ \boldsymbol{\eta_c'} \end{pmatrix}} \tag{41}$$

Figure 70: Protocol rules

BBody environments

$$\mathsf{BBodyEnv} = \begin{pmatrix} pp & \in & \mathsf{PParams} & \text{protocol parameters} \\ acnt & \in & \mathsf{Acnt} & \text{accounting state} \end{pmatrix}$$

BBody states

$$\mathsf{BBodyState} = \begin{pmatrix} ls & \in & \mathsf{LState} & \text{ledger state} \\ b & \in & \mathsf{BlocksMade} & \text{blocks made} \end{pmatrix}$$

BBody Transitions

$$\_ \vdash \_ \xrightarrow[\text{BBODY}]{} \_ \subseteq \mathbb{P}\,(\mathsf{BBodyEnv} \times \mathsf{BBodyState} \times \mathsf{Block} \times \mathsf{BBodyState})$$

BBody helper function

$$\mathsf{incrBlocks} \in \mathsf{Bool} \to \mathsf{KeyHash}_{pool} \to \mathsf{BlocksMade} \to \mathsf{BlocksMade}$$

$$\mathsf{incrBlocks}\ isOverlay\ hk\ b = \begin{cases} b & \text{if } isOverlay \\ b \cup \{hk \mapsto 1\} & \text{if } hk \notin \operatorname{dom} b \\ b \underset{\cup}{\rightarrow} \{hk \mapsto n+1\} & \text{if } hk \mapsto n \in b \end{cases}$$

Figure 71: BBody transition-system types

12.12 Block Body Transition

The Block Body Transition updates the block body state which comprises the ledger state and the map describing the produced blocks. The environment of the BBODY transition are the protocol parameters and the accounting state. The environments and states are defined in Figure 71, along with a helper function incrBlocks, which counts the number of non-overlay blocks produced by each stake pool.

The BBODY transition rule is shown in Figure 72, its sub-rule is LEDGERS which does the update of the ledger state. The signal is a block from which we extract:

- The sequence of transactions *txs* of the block.
- The block header body *bhb*.
- The verification key *vk* of the issuer of the *block* and its hash *hk*.

The transition is executed if the following preconditions are met:

- The size of the block body matches the value given in the block header body.
- The hash of the block body matches the value given in the block header body.
- The LEDGERS transition succeeds.

After this, the transition system updates the mapping of the hashed stake pool keys to the incremented value of produced blocks ($n + 1$), provided the current slot is not an overlay slot.

The BBODY rule has two predicate failures:

- if the size of the block body in the header is not equal to the real size of the block body, there is a *WrongBlockBodySize* failure.
- if the hash of the block body is not also the hash of transactions, there is an *InvalidBodyHash* failure.

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

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

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

$$\text{Block-Body}\ \frac{\begin{matrix}\mathsf{bslot}\ bhb \\ pp \\ acnt\end{matrix} \vdash ls \xrightarrow[\text{LEDGERS}]{txs} ls'}{\begin{matrix}pp \\ acnt\end{matrix} \vdash \begin{pmatrix} ls \\ b \end{pmatrix} \xrightarrow[\text{BBODY}]{block} \begin{pmatrix} ls' \\ \mathsf{incrBlocks}\ (\mathsf{isOverlaySlot}\ fSlot\ (\mathsf{d}\ pp)\ slot)\ hk\ b \end{pmatrix}} \tag{42}$$

Figure 72: BBody rules

12.13 Chain Transition

The CHAIN transition rule is the main rule of the blockchain layer part of the STS. It calls BHEAD, PRTCL, and BBODY as sub-rules.

The chain rule has no environment.

The transition checks six things (via chainChecks and prtlSeqChecks from Figure 74):

- The slot in the block header body is larger than the last slot recorded.
- The block number increases by exactly one.
- The previous hash listed in the block header matches the previous block header hash which was recorded.
- The size of *bh* is less than or equal to the maximal size that the protocol parameters allow for block headers.
- The size of the block body, as claimed by the block header, is less than or equal to the maximal size that the protocol parameters allow for block bodies. It will later be verified that the size of the block body matches the size claimed in the header (see Figure 72).
- The node is not obsolete, meaning that the major component of the protocol version in the protocol parameters is not bigger than the constant MaxMajorPV.

The chain state is shown in Figure 73, it consists of the following:

- The epoch specific state *nes*.
- The operational certificate issue number map *cs*.
- The epoch nonce η_0.
- The evolving nonce η_v.
- The candidate nonce η_c.
- The previous epoch hash nonce η_h.
- The last header hash h.
- The last slot s_ℓ.
- The last block number b_ℓ.

The CHAIN transition rule is shown in Figure 75. Its signal is a *block*. The transition uses a few helper functions defined in Figure 74.

The CHAIN rule has six predicate failures:

- If the slot of the block header body is not larger than the last slot or greater than the current slot, there is a *WrongSlotInterval* failure.
- If the block number does not increase by exactly one, there is a *WrongBlockNo* failure.
- If the hash of the previous header of the block header body is not equal to the hash given in the environment, there is a *WrongBlockSequence* failure.
- If the size of the block header is larger than the maximally allowed size, there is a *HeaderSizeTooLarge* failure.

Chain states

$$\mathsf{LastAppliedBlock} = \left(\begin{array}{rcll} b_\ell & \in & \mathsf{Slot} & \text{last block number} \\ s_\ell & \in & \mathsf{Slot} & \text{last slot} \\ h & \in & \mathsf{HashHeader} & \text{latest header hash} \end{array}\right)$$

$$\mathsf{ChainState} = \left(\begin{array}{rcll} nes & \in & \mathsf{NewEpochState} & \text{epoch specific state} \\ cs & \in & \mathsf{KeyHash}_{pool} \mapsto \mathbb{N} & \text{operational certificate issue numbers} \\ \eta_0 & \in & \mathsf{Seed} & \text{epoch nonce} \\ \eta_v & \in & \mathsf{Seed} & \text{evolving nonce} \\ \eta_c & \in & \mathsf{Seed} & \text{candidate nonce} \\ \eta_h & \in & \mathsf{Seed} & \text{seed generated from hash of previous epoch's last block header} \\ lab & \in & \mathsf{LastAppliedBlock}^? & \text{latest applied block} \end{array}\right)$$

Chain Transitions

$$\vdash \_ \xrightarrow[\text{CHAIN}]{} \_ \subseteq \mathbb{P}\left(\mathsf{ChainState} \times \mathsf{Block} \times \mathsf{ChainState}\right)$$

Figure 73: Chain transition-system types

- If the size of the block body is larger than the maximally allowed size, there is a *BlockSizeTooLarge* failure.
- If the major component of the protocol version is larger than MaxMajorPV, there is a *ObsoleteNode* failure.

Chain Transition Helper Functions

$$\begin{aligned}
&\mathsf{getGKeys} \in \mathsf{NewEpochState} \to \mathbb{P}\ \mathsf{KeyHash_G}\\
&\mathsf{getGKeys}\ nes = \mathrm{dom}\ genDelegs\\
&\quad \textbf{where} \qquad (_,\ _,\ _,\ es,\ _,\ _) = nes\\
&\qquad\qquad\quad (_,\ _,\ ls,\ _,\ _) = es\\
&\qquad\qquad (_,\ ((_,\ _,\ _,\ _,\ genDelegs,\ _),\ _)) = ls
\end{aligned}$$

$$\begin{aligned}
&\mathsf{updateNES} \in \mathsf{NewEpochState} \to \mathsf{BlocksMade} \to \mathsf{LState} \to \mathsf{NewEpochState}\\
&\mathsf{updateNES}\ (e_\ell,\ b_{prev},\ _,\ (acnt,\ ss,\ _,\ prevPp,\ pp),\ ru,\ pd)\ b_{cur}\ ls =\\
&\quad (e_\ell,\ b_{prev},\ b_{cur},\ (acnt,\ ss,\ ls,\ prevPp,\ pp),\ ru,\ pd)
\end{aligned}$$

$$\begin{aligned}
&\mathsf{chainChecks} \in \mathbb{N} \to (\mathbb{N}, \mathbb{N}, \mathsf{ProtVer}) \to \mathsf{BHeader} \to \mathsf{Bool}\\
&\mathsf{chainChecks}\ maxpv\ (maxBHSize,\ maxBBSize,\ protocolVersion)\ bh =\\
&\quad m \leq maxpv\\
&\quad \wedge\ \mathsf{bHeaderSize}\ bh \leq maxBHSize\\
&\quad \wedge\ \mathsf{hBbsize}\ (\mathsf{bhbody}\ bh) \leq maxBBSize\\
&\quad \textbf{where}\ \ (m,\ _) := protocolVersion
\end{aligned}$$

$$\begin{aligned}
&\mathsf{lastAppliedHash} \in \mathsf{LastAppliedBlock}^? \to \mathsf{HashHeader}^?\\
&\mathsf{lastAppliedHash}\ lab = \begin{cases} \diamond & lab = \diamond \\ h & lab = (_,\ _,\ h) \end{cases}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{prtlSeqChecks} \to \mathsf{LastAppliedBlock}^? \to \mathsf{BHeader} \to \mathsf{Bool}\\
&\mathsf{prtlSeqChecks}\ lab\ bh = \begin{cases} \mathsf{True} & lab = \diamond \\ s_\ell < slot \wedge b_\ell + 1 = bn \wedge ph = \mathsf{bprev}\ bhb & lab = (b_\ell,\ s_\ell,\ _) \end{cases}\\
&\quad \textbf{where}\\
&\qquad bhb := \mathsf{bhbody}\ bh\\
&\qquad bn := \mathsf{bblockno}\ bhb\\
&\qquad slot := \mathsf{bslot}\ bhb\\
&\qquad ph := \mathsf{lastAppliedHash}\ lab
\end{aligned}$$

Figure 74: Helper Functions used in the CHAIN transition

$$bh := \mathsf{bheader}\ block \qquad bhb := \mathsf{bhbody}\ bh \qquad s := \mathsf{bslot}\ bhb$$

$$\mathsf{prtlSeqChecks}\ lab\ bh$$

$$\vdash nes \xrightarrow[\text{TICK}]{s} nes'$$

$$\begin{gathered}(e_1, \_, \_, \_, \_, \_) := nes \\ (e_2, \_, b_{cur}, es, \_, \_, pd) := nes' \\ (acnt, \_, ls, \_, pp) := es \\ (\_, ((\_, \_, \_, \_, genDelegs, \_), (\_, \_, \_))) := ls \\ ne := e_1 \neq e_2 \\ \eta_{ph} := \mathsf{prevHashToNonce}\ (\mathsf{lastAppliedHash}\ lab)\end{gathered}$$

$$\mathsf{chainChecks}\ \mathsf{MaxMajorPV}\ (\mathsf{maxHeaderSize}\ pp, \mathsf{maxBlockSize}\ pp, \mathsf{pv}\ pp)\ bh$$

$$\begin{matrix} pp \\ \eta_c \\ \eta_{ph} \end{matrix} \vdash \begin{pmatrix} \eta_0 \\ \eta_h \end{pmatrix} \xrightarrow[\text{TICKN}]{ne} \begin{pmatrix} \eta_0' \\ \eta_h' \end{pmatrix}$$

$$\begin{matrix} (\mathsf{d}\ pp) \\ pd \\ genDelegs \\ \eta_0' \end{matrix} \vdash \begin{pmatrix} cs \\ \eta_v \\ \eta_c \end{pmatrix} \xrightarrow[\text{PRTCL}]{bh} \begin{pmatrix} cs' \\ \eta_v' \\ \eta_c' \end{pmatrix}$$

$$\begin{matrix} pp \\ acnt \end{matrix} \vdash \begin{pmatrix} ls \\ b_{cur} \end{pmatrix} \xrightarrow[\text{BBODY}]{block} \begin{pmatrix} ls' \\ b_{cur}' \end{pmatrix}$$

$$\begin{gathered}nes'' := \mathsf{updateNES}\ nes'\ b_{cur}'\ ls' \\ lab' := (\mathsf{bblockno}\ bhb, s, \mathsf{bhash}\ bh)\end{gathered}$$

$$\text{Chain} \frac{}{\vdash \begin{pmatrix} nes \\ cs \\ \eta_0 \\ \eta_v \\ \eta_c \\ \eta_h \\ lab \end{pmatrix} \xrightarrow[\text{CHAIN}]{block} \begin{pmatrix} \boldsymbol{nes''} \\ \boldsymbol{cs'} \\ \boldsymbol{\eta_0'} \\ \boldsymbol{\eta_v'} \\ \boldsymbol{\eta_c'} \\ \boldsymbol{\eta_h'} \\ \boldsymbol{lab'} \end{pmatrix}} \tag{43}$$

Figure 75: Chain rules

12.14 Byron to Shelley Transition

This section defines the valid initial Shelley ledger states and describes how to transition the state held by the Byron ledger to Shelley. The Byron ledger state CEState is defined in [BC-D1]. The valid initial Shelley ledger states are exactly the range of the function initialShelleyState defined in Figure 76. Figure 77 defines the transition function from Byron. Note that we use the hash of the final Byron header as the first evolving and candidate nonces for Shelley.

Shelley Initial States

$$\mathsf{initialShelleyState} \in \mathsf{LastAppliedBlock}^{?} \to \mathsf{Epoch} \to \mathsf{UTxO} \to \mathsf{Coin} \to \mathsf{GenesisDelegation} \to (\mathsf{Slot} \mapsto \mathsf{KeyHash}_{\mathsf{G}}{}^{?}) \to \mathsf{PParams} \to \mathsf{Seed} \to \mathsf{ChainState}$$

$$\mathsf{initialShelleyState}\begin{pmatrix} lab \\ e \\ utxo \\ reserves \\ genDelegs \\ pp \\ initNonce \end{pmatrix} = \begin{pmatrix} \begin{pmatrix} e \\ \emptyset \\ \emptyset \\ \begin{pmatrix} 0 \\ reserves \end{pmatrix} \\ \begin{pmatrix} \begin{pmatrix} (\emptyset,\emptyset) \\ (\emptyset,\emptyset) \\ (\emptyset,\emptyset) \\ \emptyset \\ \emptyset \\ 0 \end{pmatrix} \\ \begin{pmatrix} \begin{pmatrix} utxo \\ 0 \\ 0 \\ (\emptyset,\emptyset) \end{pmatrix} \\ \begin{pmatrix} \begin{pmatrix} \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \\ genDelegs \\ \emptyset \end{pmatrix} \\ \begin{pmatrix} \emptyset \\ \emptyset \\ \emptyset \end{pmatrix} \end{pmatrix} \end{pmatrix} \\ pp \\ pp \end{pmatrix} \\ \diamond \\ \emptyset \\ cs \\ initNonce \\ initNonce \\ initNonce \\ 0_{seed} \\ lab \end{pmatrix}$$

where $cs = \{hk \mapsto 0 \mid (hk, \_) \in \mathrm{range}\, genDelegs\}$

Figure 76: Initial Shelley States

Byron to Shelley Transition

$$\mathsf{toShelley} \in \mathsf{CEState} \to \mathsf{GenesisDelegation} \to \mathsf{BlockNo} \to \mathsf{ChainState}$$

$$\mathsf{toShelley} \begin{pmatrix} s_{last} \\ \_ \\ h \\ (utxo,\ reserves) \\ \_ \\ us \end{pmatrix} gd\ bn = \mathsf{initialShelleyState} \begin{pmatrix} (s_{last}\ bn,\ \mathsf{prevHashToNonce}\ h) \\ e \\ \mathsf{hash}\ h \\ utxo \\ reserves \\ gd \\ \mathsf{overlaySchedule}\ e\ (\mathrm{dom}\ gd)\ pp \\ pp \\ \mathsf{prevHashToNonce}\ h \end{pmatrix}$$

where

$$e = \mathsf{epoch}\ s_{last}$$

$$pp = \mathsf{pps}\ us$$

Figure 77: Byron to Shelley State Transtition

13 Software Updates

Updates to the software will include increasing the protocol version. An increase in the major version indicates a hard fork, and the minor version a soft fork (meaning old software can validate but not produce new blocks).

The current protocol version (ProtVer) is a member of the protocol parameters. It represents a specific version of the *ledger rules*. If *pv* changes, this document may have to be updated with the new rules and types if there is a change in the logic. If there is a change in the transition rules, nodes must have software installed that can implement these rules at the epoch boundary when the protocol parameter adoption occurs. Switching to using these new rules is mandatory in the sense that if the nodes do not have the applications implementing them, this will prevent a user from reading and writing to the ledger.

Applications must sometimes support *several different versions* of ledger rules in order to accommodate the timely switch of the ProtVer at the epoch boundary. In this situation, the newest protocol version a node is ready to use is included in the block header of the blocks it produces, see 12.1. This is either:

- the current version (if there is no protocol version update pending or the node has not updated to an upcoming software version capable of of implementing a newer protocol version), or
- the next protocol version, (if the node has updated its software, but the current protocol version on the ledger has not been updated yet).

Stake pools have some agency in the process of adoption of new protocol versions. They may refuse to download and install updates. Since software updates cannot be *forced* on the users, if the majority of users do not perform an update which allows the switch to the next ProtVer, it cannot happen.

Note that if there is a *new protocol version* implemented by new software, the core nodes can monitor how many nodes are ready to use the new protocol version via the block headers. Once enough nodes are ready for the new protocol version, this may now be updated as well (by the mechanism in described in Section 7).

14 Transition Rule Dependencies

Figure 78 shows all STS rules, the sub-rules they use and possible dependencies. Each node in the graph represents one rule, the top rule being CHAIN. A straight arrow from one node to another one represents a sub-rule relationship. There are two recursive rules, LEDGERS and DELEGS which have self loops.

An arrow with a dotted line from one node to another represents a dependency in the sense that the output of the target rule is an input to the source one, either as part of the source state, the environment or the signal. In most cases these dependencies are between sub-rules of a rule. In the case of recursive rules, the sub-rule can also have a dependency on the super-rule. Those recursively call themselves while traversing the input signal sequence until reaching the base case with an empty input sequence.

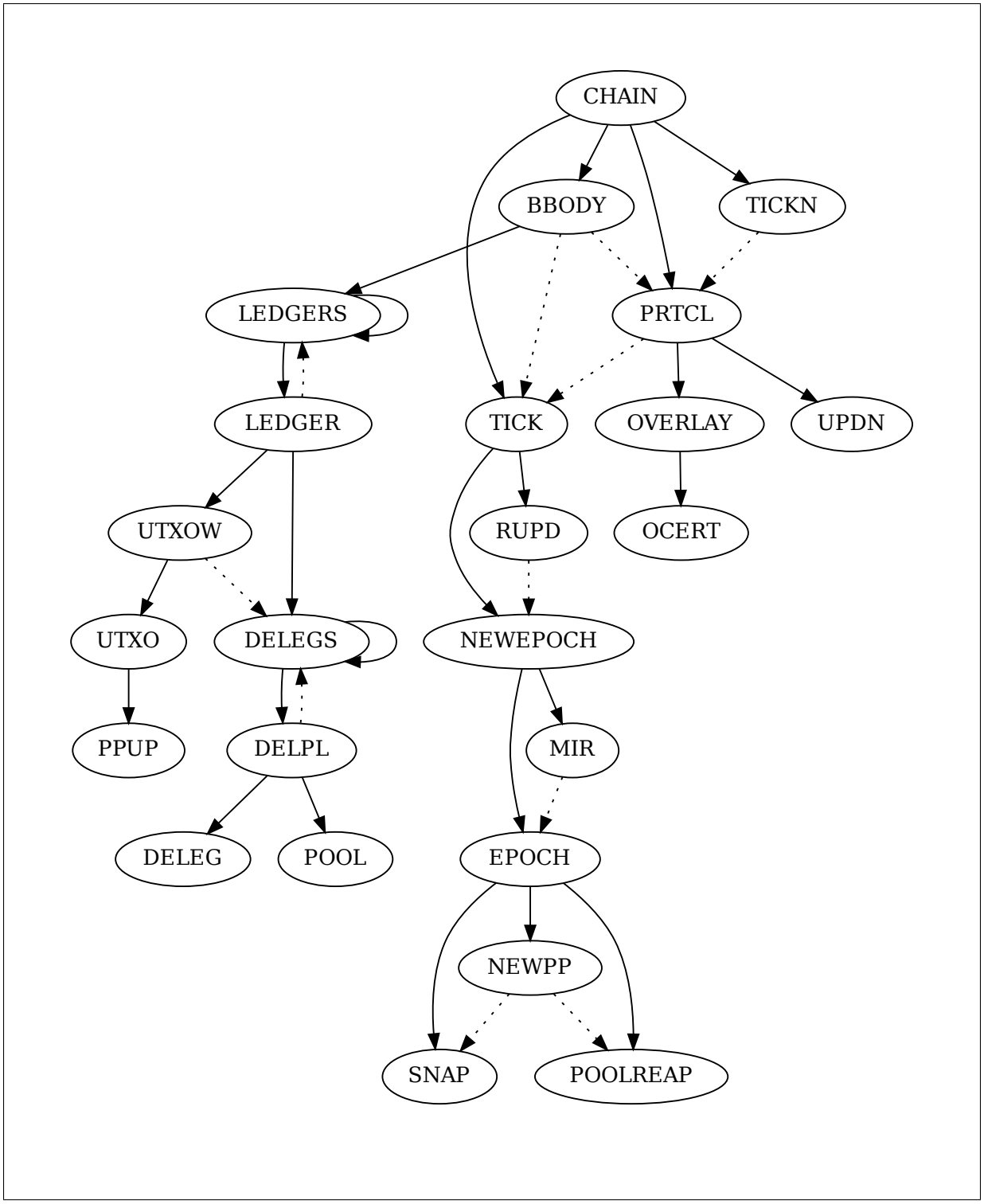


Figure 78: STS Rules, Sub-Rules and Dependencies

15 Properties

This section describes the properties that the ledger should have. The goal is to to include these properties in the executable specification to enable e.g. property-based testing or formal verification.

15.1 Header-Only Validation

The header-only validation properties of the Shelley Ledger are the analogs of those from Section 8.1 of [BC-D1].

In any given chain state, the consensus layer needs to be able to validate the block headers without having to download the block bodies. Property 15.1 states that if an extension of a chain that spans less than StabilityWindow slots is valid, then validating the headers of that extension is also valid. This property is useful for its converse: if the header validation check for a sequence of headers does not pass, then we know that the block validation that corresponds to those headers will not pass either.

First we define the header-only version of the CHAIN transition, which we call CHAINHEAD. It is very similar to CHAIN, the differences being:

- The CHAINHEAD signal is not a block, but a block header (BHeader).
- CHAINHEAD does not call BBODY.
- CHAINHEAD does not call TICK, but instead calls the similar TICKF, which differs by not calling the reward update transition RUPD.
- CHAINHEAD does not store the new epoch state *nes* in its state, but rather contains it in the environment. We will conveniently **abuse the tuple notation** and write $(nes, \tilde{s}) = s$ for splitting the chain state into the new epoch state and the remaiing fields.

$$\text{TickForecast}\ \frac{\begin{array}{c}\vdash nes \xrightarrow[\text{NEWEPOCH}]{\mathsf{epoch}\ slot} nes' \\ (e'_\ell,\ b'_{prev},\ b'_{cur},\ es',\ ru',\ pd') := nes' \\ es'' := \mathsf{adoptGenesisDelegs}\ es'\ slot \\ forecast := (e'_\ell,\ b'_{prev},\ b'_{cur},\ es'',\ ru',\ pd')\end{array}}{\vdash nes \xrightarrow[\text{TICKF}]{slot} \mathit{forecast}} \tag{44}$$

Figure 79: Tick Forecast rules

$$bhb := \mathsf{bhbody}\ bh \qquad s := \mathsf{bslot}\ bhb$$

$$\mathsf{prtlSeqChecks}\ lab\ bh$$

$$\vdash nes \xrightarrow[\text{TICKF}]{s} forecast$$

$$\begin{gathered}(e_1, \_, \_, \_, \_, \_) := nes \\ (e_2, \_, \_, es, \_, \_, pd) := forecast \\ (acnt, \_, ls, \_, pp) := es \\ (\_, ((\_, \_, \_, \_, genDelegs, \_), (\_, \_, \_))) := ls \\ ne := e_1 \neq e_2 \\ \eta_{ph} := \mathsf{prevHashToNonce}\ (\mathsf{lastAppliedHash}\ lab)\end{gathered}$$

$$\mathsf{chainChecks}\ \mathsf{MaxMajorPV}\ (\mathsf{maxHeaderSize}\ pp,\ \mathsf{maxBlockSize}\ pp,\ \mathsf{pv}\ pp)\ bh$$

$$\begin{matrix} pp \\ \eta_c \\ \eta_{ph} \end{matrix} \vdash \begin{pmatrix} \eta_0 \\ \eta_h \end{pmatrix} \xrightarrow[\text{TICKN}]{ne} \begin{pmatrix} \eta_0' \\ \eta_h' \end{pmatrix}$$

$$\begin{matrix} (\mathsf{d}\ pp) \\ pd \\ genDelegs \\ \eta_0' \end{matrix} \vdash \begin{pmatrix} cs \\ \eta_v \\ \eta_c \end{pmatrix} \xrightarrow[\text{PRTCL}]{bh} \begin{pmatrix} cs' \\ \eta_v' \\ \eta_c' \end{pmatrix}$$

$$\text{ChainHead} \frac{lab' := (\mathsf{bblockno}\ bhb,\ s,\ \mathsf{bhash}\ bh)}{nes \vdash \begin{pmatrix} cs \\ \eta_0 \\ \eta_v \\ \eta_c \\ \eta_h \\ lab \end{pmatrix} \xrightarrow[\text{CHAINHEAD}]{bh} \begin{pmatrix} \boldsymbol{cs'} \\ \boldsymbol{\eta_0'} \\ \boldsymbol{\eta_v'} \\ \boldsymbol{\eta_c'} \\ \boldsymbol{\eta_h'} \\ \boldsymbol{lab'} \end{pmatrix}} \tag{45}$$

Figure 80: Chain-Head rules

Property 15.1 (Header only validation)**.** For all states s with slot number t[1], and chain extensions E with corresponding headers H such that:

$$0 \leq t_E - t \leq \mathsf{StabilityWindow}$$

we have:

$$\vdash s \xrightarrow[\text{CHAIN}]{E} {}^* s' \implies nes \vdash \tilde{s} \xrightarrow[\text{CHAINHEAD}]{H} {}^* s''$$

where $s = (nes, \tilde{s})$, t_E is the maximum slot number appearing in the blocks contained in E, and H is obtained from E by applying bheader to each block in E.

Property 15.2 (Body only validation)**.** For all states s with slot number t, and chain extensions $E = [b_0, \ldots, b_n]$ with corresponding headers H such that:

$$0 \leq t_E - t \leq \mathsf{StabilityWindow}$$

[1] i.e. the component s_ℓ of the last applied block of s equals t

we have that for all $i \in [1, n]$:

$$nes \vdash \tilde{s} \xrightarrow[\text{CHAINHEAD}]{H}{}^{*} s_h \wedge \vdash (nes,\ \tilde{s}) \xrightarrow[\text{CHAIN}]{[b_0 \ldots b_{i-1}]}{}^{*} s_{i-1} \implies nes' \vdash \tilde{s}_{i-1} \xrightarrow[\text{CHAINHEAD}]{h_i} s'_h$$

where $s_{i-1} = (nes',\ \tilde{s}_{i-1})$, t_E is the maximum slot number appearing in the blocks contained in E.

Property 15.2 states that if we validate a sequence of headers, we can validate their bodies independently and be sure that the blocks will pass the chain validation rule. To see this, given an environment e and initial state s, assume that a sequence of headers $H = [h_0, \ldots, h_n]$ corresponding to blocks in $E = [b_0, \ldots, b_n]$ is valid according to the chainhead transition system:

$$nes \vdash \tilde{s} \xrightarrow[\text{CHAINHEAD}]{H}{}^{*} \tilde{s'}$$

Assume the bodies of E are valid according to the bbody rules, but E is not valid according to the chain rule. Assume that there is a $b_j \in E$ such that it is **the first block** such that does not pass the chain validation. Then:

$$\vdash (nes,\ \tilde{s}) \xrightarrow[\text{CHAIN}]{[b_0, \ldots b_{j-1}]}{}^{*} s_j$$

But by Property 15.2 we know that

$$nes_j \vdash \tilde{s}_j \xrightarrow[\text{CHAINHEAD}]{h_j} \tilde{s}_{j+1}$$

which means that block b_j has valid headers, and this in turn means that the validation of b_j according to the chain rules must have failed because it contained an invalid block body. But this contradicts our assumption that the block bodies were valid.

Property 15.3 (Existence of roll back function)**.** There exists a function f such that for all chains

$$C = C_0; b; C_1$$

we have that if for all alternative chains C'_1, $|C'_1| \leq \frac{\mathsf{StabilityWindow}}{2}$, with corresponding headers H'_1

$$\vdash s_0 \xrightarrow[\text{CHAIN}]{C_0;b}{}^{*} s_1 \xrightarrow[\text{CHAIN}]{C_1}{}^{*} s_2 \wedge \vdash s_1 \xrightarrow[\text{CHAIN}]{C'_1}{}^{*} s'_1 \implies nes \vdash \tilde{s} \xrightarrow[\text{CHAINHEAD}]{H'_1}{}^{*} s_h$$

where $\mathsf{f}\ (\mathsf{bheader}\ b)\ s_2 = (nes,\ \tilde{s})$.

Property 15.3 expresses the fact the there is a function that allow us to recover the header-only state by rolling back at most k blocks, and use this state to validate the headers of an alternate chain. Note that this property is not inherent to the chain rules and can be trivially satisfied by any function that keeps track of the history of the intermediate chain states up to k blocks back. This property is stated here so that it can be used as a reference for the tests in the consensus layer, which uses the rules presented in this document.

15.2 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 (ledger in the genesis state) and only executing LEDGER state transition rules as specified in Section 10 which changes either the UTxO or the delegation state.

In Figure 81 $Genesis_{Id}$ 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. The function getUTxO extracts the UTxO from a UTxO state.

$$
\begin{array}{rr}
Genesis_{Id} \in & \mathsf{TxId} \\
Genesis_{Out} \in & \mathsf{TxOut} \\
Genesis_{UTxO} := & (Genesis_{Id}, 0) \mapsto Genesis_{Out} \\
\mathsf{ledgerState} \in & \left(\begin{array}{c} \mathsf{UTxOState} \\ \mathsf{DPState} \end{array} \right) \\
 & \\
\mathsf{getUTxO} \in & \mathsf{UTxOState} \to \mathsf{UTxO} \\
\mathsf{getUTxO} := & (utxo, \_, \_, \_) \mapsto utxo
\end{array}
$$

Figure 81: Definitions and Functions for Valid Ledger State

Definition 15.1 (Valid Ledger State).

$$
\forall l_0, \ldots, l_n \in \mathsf{LState}, lenv_0, \ldots, lenv_n \in \mathsf{LEnv}, l_0 = \left(\begin{array}{c} (Genesis_{UTxO}, \_, \_, \_) \\ \left(\begin{array}{c} \emptyset \\ \emptyset \end{array} \right) \end{array} \right)
$$
$$
\implies \forall 0 < i \leq n, (\exists tx_i \in \mathsf{Tx}, lenv_{i-1} \vdash l_{i-1} \xrightarrow[\text{LEDGER}]{tx_i} l_i) \implies \mathsf{validLedgerState}\ l_n
$$

Definition 15.1 defines a valid ledger state reachable from the genesis state via valid LEDGER STS transitions. This gives a constructive rule how to reach a valid ledger state.

15.3 Ledger Properties

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

Property 15.4 (Preserve Balance).

$$
\forall l, l' \in \mathsf{LState} : \mathsf{validLedgerstate}\ l, l = (u, \_, \_, \_), l' = (u', \_, \_, \_)
$$
$$
\implies \forall tx \in \mathsf{Tx}, lenv \in \mathsf{LEnv}, lenv \vdash u \xrightarrow[\text{UTXOW}]{tx} u'
$$
$$
\implies \mathsf{destroyed}\ pc\ utxo\ stkCreds\ rewards\ tx = \mathsf{created}\ pc\ stPools\ tx
$$

Property 15.4 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 15.5 (Preserve Balance Restricted to TxIns in Balance of TxOuts).

$$
\forall l, l' \in \mathsf{ledgerState} : \mathsf{validLedgerstate}\ l, l = (u, \_, \_, \_), l' = (u', \_, \_, \_)
$$
$$
\implies \forall tx \in \mathsf{Tx}, lenv \in \mathsf{LEnv}, lenv \vdash u \xrightarrow[\text{UTXOW}]{tx} u'
$$
$$
\implies \mathsf{ubalance}(\mathsf{txins}\ tx \lhd \mathsf{getUTxO}\ u) = \mathsf{ubalance}(\mathsf{outs}\ tx) + \mathsf{txfee}\ tx + depositChange
$$

Property 15.5 states a slightly 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 and the change in the deposit *depositChange* (cf. Fig. 16).

Property 15.6 (Preserve Outputs of Transaction).

$$\forall l, l' \in \mathsf{ledgerState} : \mathsf{validLedgerstate}\ l, l = (u, \_, \_, \_), l' = (u', \_, \_, \_)$$
$$\implies \forall tx \in \mathsf{Tx}, lenv \in \mathsf{LEnv}, lenv \vdash u \xrightarrow[\text{UTXOW}]{tx} u' \implies \forall out \in \mathsf{outs}\ tx, out \in \mathsf{getUTxO}\ u'$$

Property 15.6 states that for all 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 15.7 (Eliminate Inputs of Transaction).

$$\forall l, l' \in \mathsf{ledgerState} : \mathsf{validLedgerstate}\ l, l = (u, \_, \_, \_), l' = (u', \_, \_, \_)$$
$$\implies \forall tx \in \mathsf{Tx}, lenv \in \mathsf{LEnv}, lenv \vdash u \xrightarrow[\text{UTXOW}]{tx} u' \implies \forall in \in \mathsf{txins}\ tx, in \notin \mathsf{dom}(\mathsf{getUTxO}\ u')$$

Property 15.7 states that for all ledger states l, l' and transaction tx as above, all transaction inputs in of tx are not in the domain of the UTxO of l', i.e., these are no longer available to spend.

Property 15.8 (Completeness and Collision-Freeness of new Transaction Ids).

$$\forall l, l' \in \mathsf{ledgerState} : \mathsf{validLedgerstate}\ l, l = (u, \_, \_, \_), l' = (u', \_, \_, \_)$$
$$\implies \forall tx \in \mathsf{Tx}, lenv \in \mathsf{LEnv}, lenv \vdash u \xrightarrow[\text{UTXOW}]{tx} u'$$
$$\implies \forall ((txId', \_) \mapsto \_) \in \mathsf{outs}\ tx, ((txId, \_) \mapsto \_) \in \mathsf{getUTxO}\ u \implies txId' \neq txId$$

Property 15.8 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 15.9 (Absence of Double-Spend).

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

Property 15.9 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.

$$
\begin{aligned}
\mathsf{getStDelegs} &\in & \mathsf{DState} &\to \mathbb{P}\ \mathsf{Credential} \\
\mathsf{getStDelegs} &:= & (\mathit{stkCreds}, \_, \_, \_, \_, \_) &\to \mathit{stkCreds} \\
\\
\mathsf{getRewards} &\in & \mathsf{DState} &\to (\mathsf{Addr}_{\mathsf{rwd}} \mapsto \mathsf{Coin}) \\
\mathsf{getRewards} &:= & (\_, \mathit{rewards}, \_, \_, \_, \_) &\to \mathit{rewards} \\
\\
\mathsf{getDelegations} &\in & \mathsf{DState} &\to (\mathsf{Credential} \mapsto \mathsf{KeyHash}) \\
\mathsf{getDelegations} &:= & (\_, \_, \mathit{delegations}, \_, \_, \_) &\to \mathit{delegations} \\
\\
\mathsf{getStPools} &\in & \mathsf{LState} &\to (\mathsf{KeyHash} \mapsto \mathsf{DCert}_{\mathsf{regpool}}) \\
\mathsf{getStPools} &:= & (\_, (\_, (\mathit{stpools}, \_, \_, \_))) &\to \mathit{stpools} \\
\\
\mathsf{getRetiring} &\in & \mathsf{LState} &\to (\mathsf{KeyHash} \mapsto \mathsf{Epoch}) \\
\mathsf{getRetiring} &:= & (\_, (\_, (\_, \_, \mathit{retiring}, \_))) &\to \mathit{retiring}
\end{aligned}
$$

Figure 82: Definitions and Functions for Stake Delegation in Ledger States

15.4 Ledger State Properties for Delegation Transitions

Property 15.10 (Registered Staking Credential with Zero Rewards).

$$
\begin{aligned}
&\forall l, l' \in \mathsf{ledgerState} : \mathsf{validLedgerstate}\ l, l = (\_, ((d, \_), \_)), l' = (\_, ((d', \_), \_)), \mathit{dEnv} \in \mathsf{DEnv} \\
&\quad \implies \forall c \in \mathsf{DCert}_{\mathsf{regkey}}, \mathit{dEnv} \vdash d \xrightarrow[\text{DELEG}]{c} d' \implies \mathsf{cwitness}\ c = \mathit{hk} \\
&\implies \mathit{hk} \notin \mathsf{getStDelegs}\ d \implies \mathit{hk} \in \mathsf{getStDelegs}\ d' \wedge (\mathsf{getRewards}\ d')[\mathsf{addr}_{\mathsf{rwd}} \mathit{hk}] = 0
\end{aligned}
$$

Property 15.10 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 credential of l' includes the credential hk associated with the key registration certificate and the associated reward is set to 0 in l'.

Property 15.11 (Deregistered Staking Credential).

$$
\begin{aligned}
&\forall l, l' \in \mathsf{ledgerState} : \mathsf{validLedgerstate}\ l, l = (\_, (d, \_)), l' = (\_, (d', \_)), \mathit{dEnv} \in \mathsf{DEnv} \\
&\quad \implies \forall c \in \mathsf{DCert}_{\mathsf{deregkey}}, \mathit{dEnv} \vdash d \xrightarrow[\text{DELEG}]{c} d' \implies \mathsf{cwitness}\ c = \mathit{hk} \\
&\implies \mathit{hk} \notin \mathsf{getStDelegs}\ d' \wedge \mathit{hk} \notin \left\{\mathsf{stakeCred}_{\mathsf{r}}\ \mathit{sc} \middle| \mathit{sc} \in \mathsf{dom}(\mathsf{getRewards}\ d')\right\} \\
&\qquad\qquad \wedge \mathit{hk} \notin \mathsf{dom}(\mathsf{getDelegations}\ d'))
\end{aligned}
$$

Property 15.11 states that for l, l' as above but with a delegation transition of type $\mathsf{DCert}_{\mathsf{deregkey}}$, the staking credential hk associated with the deregistration certificate is not in the set of staking credentials of l' and is not in the domain of either the rewards or the delegation map of l'.

Property 15.12 (Delegated Stake).

$$\begin{aligned}\forall l, l' \in \mathsf{ledgerState} : \mathsf{validLedgerstate}\ l, l = (\_, (d, \_)), l' = (\_, (d', \_)), dEnv \in \mathsf{DEnv} \\ \implies \forall c \in \mathsf{DCert_{delegate}}, dEnv \vdash d \xrightarrow[\text{DELEG}]{c} d' \implies \mathsf{cwitness}\ c = hk \\ \implies hk \in \mathsf{getStDelegs}\ d' \wedge (\mathsf{getDelegations}\ d')[hk] = \mathsf{dpool}\ c\end{aligned}$$

Property 15.12 states that for l, l' as above but with a delegation transition of type $\mathsf{DCert_{delegate}}$, the staking credential hk associated with the deregistration certificate is in the set of staking credentials of l and delegates to the staking pool associated with the delegation certificate in l'.

Property 15.13 (Genesis Keys are Always All Delegated).

$$\begin{aligned}\forall l, l' \in \mathsf{LState} : \mathsf{validLedgerstate}\ l, \\ \implies \forall \Gamma \in \mathsf{Tx}^*, env \in (\mathsf{Slot} \times \mathsf{PParams}), \\ env \vdash l \xrightarrow[\text{LEDGERS}]{\Gamma} l' \implies |genDelegs| = 7\end{aligned}$$

Property 15.13 states that all seven of the genesis keys are constantly all delegated after applying a list of transactions to a valid ledger state.

15.5 Ledger State Properties for Staking Pool Transitions

Property 15.14 (Registered Staking Pool).

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

Property 15.14 states that for l, l' as above but with a delegation transition of type $\mathsf{DCert_{regpool}}$, 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 15.15 (Start Staking Pool Retirement).

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

Property 15.15 states that for l, l' as above but with a delegation transition of type $\mathsf{DCert_{retirepool}}$, the key hk is associated with the author of the pool registration certificate in *stpools* of l' and that hk is in the map of retiring staking pools of l' with retirement epoch e, as well as that hk is in the map of stake pools in l and l'.

Property 15.16 (Stake Pool Reaping).

$$\begin{aligned}
&\forall l, l' \in \mathsf{ledgerState}, e \in \mathsf{Epoch} : \mathsf{validLedgerstate}\ l, \\
&\qquad l = (_, (d, p)), l' = (_, (d', p')), pp \in \mathsf{PParams}, acnt, acnt' \in \mathsf{Acnt} \\
&\implies pp \vdash (acnt, d, p) \xrightarrow[\text{POOLREAP}]{e} (acnt, d', p') \implies \forall retire \in (\mathsf{getRetiring}\ p)^{-1}[e], retire \neq \emptyset \\
&\qquad \wedge\ retire \subseteq \mathsf{dom}(\mathsf{getStPool}\ p) \wedge retire \cap \mathsf{dom}(\mathsf{getStPool}\ p') = \emptyset \\
&\qquad\qquad \wedge\ retire \cap \mathsf{dom}(\mathsf{getRetiring}\ p') = \emptyset
\end{aligned}$$

Property 15.16 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 e. In l' all those stake pools are removed from the maps $stpools$ and $retiring$.

15.6 Properties of Numerical Calculations

The numerical calculations for refunds and rewards in (see Section 11) 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 usable in properties-based testing for validation in the executable spec.

Property 15.17 (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 15.18 (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 $\overline{p}$ being the relation $\frac{n}{\max(1, \overline{N})}$ of the number of produced blocks n of one pool to the total number $\overline{N}$ of produced blocks in an epoch and $maxP$ being the maximal reward for the stake pool. This gives us:

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

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

Property 15.19 (Reward Splitting). The reward splitting is done via $\mathsf{r}_{\mathsf{operator}}$ 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:

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

Values associated with the leader value calculations

$$\begin{array}{rcll} \mathit{certNat} & \in & \{n | n \in \mathbb{N}, n \in [0, 2^{512})\} & \text{Certified natural value from VRF} \\ f & \in & [0,1] & \text{Active slot coefficient} \\ \sigma & \in & [0,1] & \text{Stake proportion} \end{array}$$

16 Leader Value Calculation

This section details how we determine whether a node is entitled to lead (under the Praos protocol) given the output of its verifiable random function calculation.

16.1 Computing the leader value

The verifiable random function gives us a 64-byte random output. We interpret this as a natural number *certNat* in the range $[0, 2^{512})$.

16.2 Node eligibility

As per [DGKR17], a node is eligible to lead when its leader value $p < 1 - (1-f)^{\sigma}$. We have

$$\begin{aligned} & p < 1 - (1-f)^{\sigma} \\ \iff & \left(\frac{1}{1-p}\right) < \exp\left(-\sigma \cdot \ln\left(1-f\right)\right) \end{aligned}$$

The latter inequality can be efficiently computed through use of its Taylor expansion and error estimation to stop computing terms once we are certain that the result will be either above or below the target value.

We carry out all computations using fixed precision arithmetic (specifically, we use 34 decimal bits of precision, since this is enough to represent the fraction of a single lovelace.)

As such, we define the following:

$$\begin{aligned} p &= \frac{\mathit{certNat}}{2^{512}} \\ q &= 1 - p \\ c &= \ln\left(1-f\right) \end{aligned}$$

and define the function *checkLeaderVal* as follows:

$$\mathsf{checkLeaderVal}\ \mathit{certNat}\ \sigma\ f = \begin{cases} \mathsf{True}, & f = 1 \\ \frac{1}{q} < \exp\left(-\sigma \cdot c\right), & \text{otherwise} \end{cases}$$

17 Errata

We list issues found within the Shelley ledger which will be corrected in future eras.

17.1 Total stake calculation

As described in [SL-D1, 3.4.3], stake is sometimes considered relative to all the delegated stake, and sometimes relative to the totally supply less the amount held by the reserves. The former is called active stake, the latter total stake.

The createRUpd function from Figure 62 uses the current value of the reserve pot, named *reserves*, to calculate the total stake. It should, however, use the value of the reserve pot as it was during the previous epoch, since createRUpd is creating the rewards earned during the previous epoch (see the overview in Section 11.1).

17.2 Active stake registrations and the Reward Calculation

The reward calculation takes the set of active reward accounts as a parameter (the forth parameter of reward in Figure 48). It is only used at the end of the calculation for filtering out unregistered accounts. Therefore, if a stake credential is deregistered just before the reward calculation starts, it cannot reregister before the end of the epoch in order to get the rewards. The time within the epoch when the reward calculation starts should not make any difference. Note that this does not affect the earnings that stake pools make from their margin, since each pool's total stake is summed before filtering. The solution is to not filter by the current active reward accounts, since rewards for unregistered accounts go to the treasury already anyway.

17.3 Stability Windows

The constant *RandomnessStabilisationWindow* was intended to be used only for freezing the candidate nonce in the UPDN rule from Figure 60. This was chosen as a good value for both Ouroboros Praos and Ouroboros Genesis. The implementation mistakenly used the constant in the RUPD rule from Figure 62, and used StabilityWindow in for the UPDN transition.

The constant StabilityWindow is a good choice for the UPDN transition while we remain using Ouroboros Praos, but it will be changed before adopting Ouroboros Genesis in the consensus layer.

There is no problem with using *RandomnessStabilisationWindow* for the RUPD transition, since anything after StabilityWindow would work, but there is no reason to do so.

17.4 Reward aggregation

On any given epoch, a reward account can earn rewards by being a member of a stake pool, and also by being the registered reward account for a stake pool (to receive leader rewards). A reward account can be registered to receive leader rewards from multiple stake pools. It was intended that reward accounts receive the sum of all such rewards, but a mistake caused reward accounts to receive at most one of them.

In Figure 48, the value of *potentialRewards* in the rewardOnePool function should be computed using an aggregating union $\cup_+$, so that member rewards are not overridden by leader rewards. Similarly, the value of *rewards* in the reward function should be computed using an aggregating union so that leader rewards from multiple sources are aggregated.

This was corrected at the Allegra hard fork. There were sixty-four stake addresses that were affected, each of which was reimbursed for the exact amount lost using a MIR certificate. Four of the stake address were unregistered and had to be re-registered. The unregistered addresses were reimbursed in transaction

$a01b9fe136e5703668c9c7776a76cc8541b84824635d237e270670a8ca56a392$, and the registered addresses were reimbursed in transaction
$8cab8049e3b8c4802d7d11277b21afc74056223a596c39ef00e002c2d1f507ad$.

17.5 Byron redeem addresses

The bootstrap addresses from Figure 6 were not intended to include the Byron era redeem addresses (those with addrtype 2, see the Byron CDDL spec). These addresses were, however, not spendable in the Shelley era. At the Allegra hard fork they were removed from the UTxO and the Ada contained in them was returned to the reserves.

17.6 Block hash used in the epoch nonce

In the CHAIN rule in Figure 75, the TICKN rule uses the last block hash of the previous epoch to create the new epoch nonce. In the implemenation, however, the *penultimate* block hash of the previous epoch is used.

References

[AP10] B. Akbarpour and L. 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.

[BC-D1] IOHK Formal Methods Team. Byron Blockchain Specification, IOHK Deliverable BC-D1, 2019. URL https://github.com/input-output-hk/cardano-ledger/tree/master/docs/.

[BDN18] D. Boneh, M. Drijvers, and G. Neven. Compact multi-signatures for smaller blockchains. In T. Peyrin and S. D. Galbraith, editors, *Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part II*, volume 11273 of *Lecture Notes in Computer Science*, pages 435–464. Springer, 2018. doi:10.1007/978-3-030-03329-3_15.

[Bir19] H. Birkholz. Concise Data Definition Language (CDDL). RFC 8610, RFC Editor, 6 2019. URL https://tools.ietf.org/html/rfc8610.

[BL-D1] IOHK Formal Methods Team. Byron Ledger Specification, IOHK Deliverable BL-D1, 2019. URL https://github.com/input-output-hk/cardano-ledger/tree/master/docs/.

[Bor13] C. Bormann. Concise Binary Object Representation (CBOR). RFC 7049, RFC Editor, 10 2013. URL https://tools.ietf.org/html/rfc7049.

[DGKR17] B. M. David, P. Gazi, A. Kiayias, and A. Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake protocol. *IACR Cryptology ePrint Archive*, 2017:573, 2017.

[DGNW20] M. Drijvers, S. Gorbunov, G. Neven, and H. Wee. Pixel: Multi-signatures for consensus. In S. Capkun and F. Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 2093–2110. USENIX Association, 2020. URL https://www.usenix.org/conference/usenixsecurity20/presentation/drijvers.

[FM-TR-2018-01] IOHK Formal Methods Team. Small Step Semantics for Cardano, IOHK Technical Report FM-TR-2018-01, 2018. URL https://github.com/input-output-hk/cardano-chain/blob/master/specs/semantics/latex/small-step-semantics.tex.

[Gol20] S. Goldberg. Verifiable Random Functions (VRFs), draft-irtf-cfrg-vrf-06. RFC draft, RFC Editor, 2 2020. URL https://tools.ietf.org/html/draft-irtf-cfrg-vrf-06.

[Jos17] S. Josefsson. Edwards-Curve Digital Signature Algorithm (EdDSA). RFC 8032, RFC Editor, 1 2017. URL https://tools.ietf.org/html/rfc8032.

[MMM01] T. Malkin, D. Micciancio, and S. Miner. Composition and efficiency tradeoffs for forward-secure digital signatures. Cryptology ePrint Archive, Report 2001/034, 2001. https://eprint.iacr.org/2001/034.

[MOR01] S. Micali, K. Ohta, and L. Reyzin. Accountable-subgroup multisignatures: Extended abstract. In *Proceedings of the 8th ACM Conference on Computer and Communications Security*, CCS '01, pages 245–254, New York, NY, USA, 2001. Association for Computing Machinery. doi:10.1145/501983.502017.

[MPSW19] G. Maxwell, A. Poelstra, Y. Seurin, and P. Wuille. Simple schnorr multi-signatures with applications to bitcoin. *Des. Codes Cryptogr.*, 87(9):2139–2164, 2019. doi:10.1007/s10623-019-00608-x.

[NRS20] J. Nick, T. Ruffing, and Y. Seurin. Musig2: Simple two-round schnorr multi-signatures. *IACR Cryptol. ePrint Arch.*, 2020:1261, 2020. URL https://eprint.iacr.org/2020/1261.

[Saa15] M.-J. Saarinen. The BLAKE2 cryptographic hash and message authentication code (MAC). RFC 7693, RFC Editor, 11 2015. URL https://tools.ietf.org/html/rfc7693.

[SL-D1] IOHK Formal Methods Team. Design Specification for Delegation and Incentives in Cardano, IOHK Deliverable SL-D1, 2018. URL https://hydra.iohk.io/job/Cardano/cardano-ledger/delegationDesignSpec/latest/download-by-type/doc-pdf/delegation_design_spec.

[Wui12] P. Wuille. Hierarchical deterministic wallets, February 2012. URL https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki. BIP-32.

[Zah18] J. 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.

A Cryptographic Details

A.1 Warning About Re-serialization

It is a bad security practice to check signatures or hashes on re-serialized data, the original bytes must be preserved.

A.2 Hashing

We present the (informal) properties of cryptographically safe hash functions:

Preimage resistance It should be hard to find a message with a given hash value.

Second preimage resistance Given one message, it should be hard to find another message with the same hash value.

Collision resistance It should be hard to find two messages with the same hash value.

In practice, several cryptographic protocols use hash functions as random oracles. However, the properties of our scheme do not depend on it.

The hashing algorithm we use for all verification keys and multi-signature scripts is BLAKE2b-224[2]. Explicitly, this is the payment and stake credentials (Figure 6), the genesis keys and their delegates (Figure 12), stake pool verification keys (Figure 22), and VRF verification keys (Figure 21). The only property we require of this hash function is that of Second Preimage resistance. Given the hash of verification key or multi-signature script, it should be hard to find a different key or script with the same hash.

Everywhere else we use BLAKE2b-256. In this case we do require second preimage resistance and collision resistance. We use the output of these hash functions to produce signatures (see Figure 20), meaning that if an adversary can find two messages with the same hash, it could break the intended EUF-CMA (Existentially UnForgeable against a Chosen Plaintext Attack) property of the underlying signature scheme.

In the CDDL specification in Appendix D, hash28 refers to BLAKE2b-224 and and hash32 refers to BLAKE2b-256. BLAKE2 is specified in RFC 7693 [Saa15].

A.3 Addresses

The sign and verify functions from Figure 2 use Ed25519. See [Jos17].

A.4 KES

The $\mathsf{sign}_{\mathsf{ev}}$ and $\mathsf{verify}_{\mathsf{ev}}$ functions from Figure 3 use the iterated sum construction from Section 3.1 of [MMM01]. We allow up to 2^7 key evolutions, which is larger than the maximum number of evolutions allow by the spec, MaxKESEvo, which will be set to 90. See Figure 66.

A.5 VRF

The verifyVrf function from Figure 5 uses ECVRF-ED25519-SHA512-Elligator2 as described in the draft IETF specification [Gol20].

More generally, we expect the following properties from a Verifyable Random Function:

Full uniqueness For any fixed public VRF key and for *any* input seed, there is a unique VRF output that can be proved valid.

[2]Note that for the signature and verification algorithms, as well as the proving and verification algorithms of VRFs, we use the hash function as defined in the IETF standard (see Section A.3 and Section A.5).

Full collision resistance It should be computationally infeasible to find two distinct VRF inputs that have the same VRF output, even if the adversary knows the private key.

Pseudorandomness Assuming the public-private key pair has been generated honestly, the VRF hash output on any adversarially-chosen VRF input looks indistinguishable from random for a computationally bounded adversary who does not know the private key.

A.6 Abstract functions

- The transaction ID function txid from Figure 10 is implemented as the BLAKE2b-256 hash of the CBOR serialization of the transaction body. **Note** that we do **not** enforce canonical CBOR and that there are multiple valid serializations for any given transaction body. The transaction ID must be computed using the original bytes that were submitted to the network.
- The seed operation $x \star y$ from Figure 5 is implemented as the BLAKE2b-256 hash of the concatenation of x and y.
- The function slotToSeed from Figure 53 is implemented as the big-endian encoding of the slot number in 8 bytes.

A.7 Multi-Signatures

As presented in Figure 4, Shelley realizes multi-signatures in a native way, via a script-like DSL. One defines the conditions required to validate a multi-signature, and the script takes care of verifying the correctness of the request. It does so in a naïve way, i.e. checking every signature individually. For instance, if the requirement is to have n valid signatures out of some set $\mathcal{V}$ of public keys, the naïve script-based solution checks if: (i) the number of submitted signatures is greater or equal to n, (ii) the distinct verification keys are part of the set $\mathcal{V}$, and (iii) at least n signatures are valid. However, there are more efficient ways to achieve this using more advanced multi-signature schemes, that allow for aggregating both the signatures and the verification procedure. This results in less data to be stored on-chain, and a cheaper verification procedure. Several schemes provide these properties [BDN18, MPSW19, NRS20, DGNW20], and we are currently investigating which would be the best fit for the Cardano ecosystem. We formally introduce multi-signature schemes.

A multi-signature scheme [MOR01] is defined as a tuple of algorithms $\mathsf{MS} = (\mathsf{MS\text{-}Setup}, \mathsf{MS\text{-}KG}, \mathsf{MS\text{-}AVK}, \mathsf{MS\text{-}Sign}, \mathsf{MS\text{-}ASign}, \mathsf{MS\text{-}Verify})$ such that $\Pi \leftarrow \mathsf{MS\text{-}Setup}(1^k)$ generates public parameters—where k is the security parameter. Given the public parameters, one can generate a verification-signing key pair calling, $(\mathsf{vk}, \mathsf{sk}) \leftarrow \mathsf{MS\text{-}KG}(\Pi)$. A multi-signature scheme provides a signing and an aggregate functionality. Mainly

- $\sigma \leftarrow \mathsf{MS\text{-}Sign}(\Pi, \mathsf{sk}, m)$, produces a signature, σ, over a message m using signing key sk, and
- $\tilde{\sigma} \leftarrow \mathsf{MS\text{-}ASig}(\Pi, m, \mathcal{V}, \mathcal{S})$, where given a message m, a set of signatures, $\mathcal{S}$, over the message m, and the corresponding set of verification keys, $\mathcal{V}$, aggregates all signatures into a single, aggregate signature $\tilde{\sigma}$.

To generate an aggregate verification key, the verifier calls the function $\mathsf{avk} \leftarrow \mathsf{MS\text{-}AVK}(\Pi, \mathcal{V})$, which given input a set of verification keys, $\mathcal{V}$, returns an aggregate verification key that can be used for the verification of the aggregate siganture: $\mathsf{MS\text{-}Verify}(\Pi, \mathsf{avk}, m, \tilde{\sigma}) \in \{\mathsf{true}, \mathsf{false}\}$.

Note the distinction between multi-signature schemes (as described above) and multi-signatures as defined in Figure 4. In Figure 4 we allow scripts to consider valid the RequireAllOf, RequireAnyOf or RequireMOfN typed signatures. In the definition above, given a set of public

keys $\mathcal{V}$, a signature is considered valid if *all* key owners participate. However, such multi-signature schemes together with a simple script-like DSL can achieve the properties defined in Figure 4 while still providing the benefits of a simple verification procedure, and a smaller signature size.

B Binary Address Format

Excepting bootstrap addresses, the binary format for every address has a 1-byte header, followed by a payload (bootstrap addresses use the binary encoding from the Byron era). The header is composed of two nibbles (two 4-bit segments), indicating the address type and the network id.

$$\begin{aligned} \mathsf{address} &= \mathsf{header\_byte} | \mathsf{payload} \\ &= \mathsf{addr\_type\_nibble} | \mathsf{network\_nibble} | \mathsf{payload} \end{aligned}$$

B.1 Header, first nibble

The first nibble of the header specifies the type of the address. Bootstrap addresses can also be identified by the first nibble.

address type	payment credential	stake credential	header, first nibble
base address	keyhash	keyhash	0000
base address	scripthash	keyhash	0001
base address	keyhash	scripthash	0010
base address	scripthash	scripthash	0011
pointer address	keyhash	ptr	0100
pointer address	scripthash	ptr	0101
enterprise address	keyhash	-	0110
enterprise address	scripthash	-	0111
bootstrap address	keyhash	-	1000
stake address	-	keyhash	1110
stake address	-	scripthash	1111
future formats	?	?	1001-1101

B.2 Header, second nibble

Excepting bootstrap addresses, the second nibble of the header specifies the network.

network	header, second nibble
testnets	0000
mainnet	0001
future networks	0010-1111

B.3 Header, examples

On a **testnet**, the header for a **pointer** address whose payment credential is a **keyhash** is:

00000100

On **mainnet**, the header for a **pointer** address whose payment credential is a **keyhash** is:

00010100

On **mainnet**, the header for a **pointer** address whose payment credential is a **scripthash** is:

00010101

B.4 Payloads

The payload for the different address types are as follows:

address type	payload
base address	two 28-byte bytestrings
pointer address	one 28-byte bytestring, and three variable-length unsigned integers
enterprise address	one 28-byte bytestrings
stake address	one 28-byte bytestrings

The variable-length encoding used in pointers addresses is the base-128 representation of the number, with the most significant bit of each byte indicating continuation (this is sometimes called a variable-length quantity, or VLQ). If the significant bit is 1, then another byte follows.

C Bootstrap Witnesses

C.1 Bootstrap Witnesses CBOR Specification

In the Byron era of Cardano, public keys are transmitted on chain as extended public keys as specified in [Wui12]. The Shelley era of Cardano does not use extended public keys, but does use the same cryptographic signing algorithm, namely Ed25519.

The extended public key consists of 64 bytes, the first 32 of which corresponds to the public key, the second 32 of which correpsonds to something called the chain code:

$$\mathsf{extended\_pubkey} = \mathsf{pubkey} | \mathsf{chain\_code}$$

The chaincode is required for constructing bootstrap witnesses.

The CBOR specification of bootstrap witnesses, named bootstrap_witness, is given in https://github.com/input-output-hk/cardano-ledger/tree/master/eras/shelley/impl/cddl-files.

In the Shelley era, only pubkey address are supported, and are named bootstrap addresses.

The bootstrap witness consists of the public key, the signature, the chain code, and the address attributes. As explained above, the public key and the signature format are the same as for all other witnesses in the Shelley era. The address attributes has the same format as from the Byron era address, as specified by addrattr in https://github.com/input-output-hk/cardano-ledger/blob/master/eras/byron/cddl-spec/byron.cddl.

C.2 Bootstrap Address ID Recovery

The bootstrap address ID, named addressid in the Byron CDDL specification above, can be computed as follows.

First construct the following byte string:

$$\mathsf{b} = \mathsf{0x830082005840} | \mathsf{pubkey\_bytes} | \mathsf{chain\_code\_bytes} | \mathsf{attributes\_bytes}$$

The address ID is then obtained by first applying sha3 256 to b and then applying blake2b44 to the result.

$$\mathsf{address\_id} = \mathsf{blake2b44}(\mathsf{sha3\_256\ b})$$

D CBOR Serialization Specification

Our CBOR (RFC 7049 [Bor13]) serialization scheme is specified using CDDL (RFC 8610 [Bir19]).

The CDDL specification is located at https://github.com/input-output-hk/cardano-ledger/tree/master/eras/shelley/test-suite/cddl-files.

Note that the ledger does **not support a canonical representation**. This is an intentional decision to discourage people from checking signatures and hashes on re-serialized data.

E Implementation of txSize

The minimum fee calculation in Figure 9 depends on an abstract txSize function. We have implemented txSize as the number of bytes in the CBOR serialization of the transaction, as defined in Appendix D.

F Proofs

For the proofs we use the automated theorem prover MetiTarski [AP10] which is specialized for proofs over real arithmetic, including elementary functions.

Proof. The property (**??**) (p. **??**) 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. For the property (15.19) (p. 105) for reward splitting 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 [SL-D1]. Using this, we show for $\hat{f} > c$

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

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))).
```

□