## Contents

1. **Introduction**  
   1.1 A note on Agda ............................................. 3  
   1.2 Separation of concerns ....................................... 3  
   1.3 Reflexive-transitive closure .................................. 3  
   1.4 Computational .............................................. 4  
   1.5 Sets & maps .................................................. 5  
   1.6 Superscripts and other special notations ...................... 5

2. **Notation** .................................................. 6

3. **Cryptographic primitives** .................................. 7

4. **Base types** .................................................. 8

5. **Token algebras** ............................................. 9

6. **Addresses** .................................................. 10

7. **Scripts** ..................................................... 11

8. **Protocol parameters** ....................................... 12

9. **Governance actions** ........................................ 15  
   9.1 Hash protection ............................................... 16  
   9.2 Votes and proposals .......................................... 16

10. **Transactions** ............................................... 19

11. **UTxO** ...................................................... 22  
    11.1 Accounting .................................................. 22  
    11.2 Witnessing .................................................. 27

12. **Governance** ................................................ 29

13. **Delegation** ................................................ 33

14. **Ledger State Transition** ................................... 37

15. **Enactment** ................................................ 39

16. **Ratification** ............................................... 41  
    16.1 Ratification requirements .................................... 41  
    16.2 Protocol parameters and governance actions ................. 41  
    16.3 Ratification restrictions .................................... 42

17. **Epoch boundary** .......................................... 48

18. **Blockchain layer** ......................................... 51

19. **Properties** ................................................ 52  
    19.1 UTxO .......................................................... 52
<table>
<thead>
<tr>
<th>Appendix: Agda essentials</th>
<th>53</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1 Record types</td>
<td>53</td>
</tr>
<tr>
<td>B Bootstrapping EnactState</td>
<td>53</td>
</tr>
<tr>
<td>C Bootstrapping the Governance System</td>
<td>54</td>
</tr>
</tbody>
</table>


1 Introduction

Repository: https://github.com/IntersectMBO/formal-ledger-specifications

This is the work-in-progress specification of the Cardano ledger. The current status of each individual era is described in Table 1.

<table>
<thead>
<tr>
<th>Era</th>
<th>Figures</th>
<th>Prose</th>
<th>Cleanup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shelley</td>
<td>Partial</td>
<td>Partial</td>
<td>Not started</td>
</tr>
<tr>
<td>Shelley-MA</td>
<td>Partial</td>
<td>Partial</td>
<td>Not started</td>
</tr>
<tr>
<td>Alonzo</td>
<td>Partial</td>
<td>Partial</td>
<td>Not started</td>
</tr>
<tr>
<td>Babbage</td>
<td>Not started</td>
<td>Not started</td>
<td>Not started</td>
</tr>
<tr>
<td>Conway [2]</td>
<td>Complete</td>
<td>Partial</td>
<td>Partial</td>
</tr>
</tbody>
</table>

Table 1: Specification progress

1.1 A note on Agda

This specification is written using the Agda programming language and proof assistant [1]. We have spent a lot of time on making this document readable for people unfamiliar with Agda (or other proof assistants, functional programming languages, etc.). However, by the nature of working in a formal language we have to play by its rules, meaning that some instances of uncommon notation are very difficult or impossible to replace. Some are explained in Section 2, but there is no guarantee that this section is complete. Anyone who is confused by the meaning of an expression, please feel free to open an issue in our repository with the ‘notation’ label.

1.2 Separation of concerns

The Cardano Node consists of three pieces:

- Networking layer, which deals with sending messages across the internet
- Consensus layer, which establishes a common order of valid blocks
- Ledger layer, which decides whether a sequence of blocks is valid

Because of this separation, the ledger gets to be a state machine:

\[ s \xrightarrow{b} X s' \]

More generally, we will consider state machines with an environment:

\[ \Gamma \vdash s \xrightarrow{b} X s' \]

These are modelled as 4-ary relations between the environment \( \Gamma \), an initial state \( s \), a signal \( b \) and a final state \( s' \). The ledger consists of 25-ish (depending on the version) such relations that depend on each other, forming a directed graph that is almost a tree.

1.3 Reflexive-transitive closure

Some STS (state transition system) relations need to be applied as many times as they can to arrive at a final state. Since we use this pattern multiple times, we define a closure operation which takes a STS relation and applies it as many times as possible.

The closure \( \leadsto \* \) of a relation \( \leadsto \) is defined in Figure 1. In the remainder of the text, the closure operation is called ReflexiveTransitiveClosure.
Closure type

\[ \vdash [-][_] : C \to S \to \text{List} \to \text{Sig} \to S \to \text{Set} \]

Closure rules

RTC-base:
\[ \Gamma \vdash s \Rightarrow \emptyset \Rightarrow s \]

RTC-ind:
\[ \begin{align*}
\bullet \Gamma \vdash s \Rightarrow \text{[sig, s']} \\
\bullet \Gamma \vdash s' \Rightarrow \text{[sigs, s'']} \\
\hline \\
\Gamma \vdash s \Rightarrow \text{[sig :: sigs, s'']} 
\end{align*} \]

Figure 1: Reflexive transitive closure

1.4 Computational

Since all such state machines need to be evaluated by the node and all nodes should compute the same states, the relations specified by them should be computable by functions. This can be captured by the definition in Figure 2 which is parametrized over the step relation.

```
record Computational (\vdash [-][,] : C \to S \to \text{Sig} \to S \to \text{Set}) : Set where
  field compute : C \to S \to \text{Sig} \to \text{Maybe S}
  \equiv\text{-just} ⇔ \text{STS}:
    compute \Gamma s b \equiv \text{just} s' \equiv \Gamma \vdash s \Rightarrow b \Rightarrow s' \\
  nothing\equiv\text{STS}:
    compute \Gamma s b \equiv \text{nothing} \equiv \forall s' \Rightarrow \neg \Gamma \vdash s \Rightarrow b \Rightarrow s'
```

Figure 2: Computational relations

Unpacking this, we have a `compute` function that computes a final state from a given environment, state and signal. The second piece is correctness: `compute` succeeds with some final state if and only if that final state is in relation to the inputs.

This has two further implications:

- Since `compute` is a function, the step relation is necessarily right-unique, i.e. there is at most one possible final state for each input data. Otherwise, we could prove that `compute` could evaluates to two different states on the same inputs, which is impossible since it is a function.

- The actual definition of `compute` is irrelevant - any two implementations of `compute` have to produce the same result on any input. This is because we can simply chain the equivalences for two different `compute` functions together.

What this all means in the end is that if we give a `Computational` instance for every relation defined in the ledger, we also have an executable version of the rules which is guaranteed to be correct. This is indeed something we have done, and the same source code that generates this document also generates a Haskell library that lets anyone run this code.
1.5 Sets & maps

The ledger heavily uses set theory. For various reasons it was necessary to implement our own set theory (there will be a paper on this some time in the future). Crucially, the set theory is completely abstract (in a technical sense - Agda has an abstract keyword) meaning that implementation details of the set theory are irrelevant. Additionally, all sets in this specification are finite.

We use this set theory to define maps as seen below, which are used in many places. We usually think of maps as partial functions (i.e. functions not defined everywhere), but importantly they are not Agda functions. We denote the powerset operation by \( \mathcal{P} \), which we use here to form a type of sets with elements in a given type.

\[
\begin{align*}
\subseteq &: \{A : \text{Set}\} \to \mathcal{P} A \to \mathcal{P} A \to \text{Set} \\
X \subseteq Y &= \forall \{x\} \rightarrow x \in X \rightarrow x \in Y \\
\equiv^e &: \{A : \text{Set}\} \to \mathcal{P} A \to \mathcal{P} A \to \text{Set} \\
X \equiv^e Y &= X \subseteq Y \times Y \subseteq X \\
Rel &: \text{Set} \to \text{Set} \to \text{Set} \\
Rel A B &= \mathcal{P} (A \times B) \\
\text{left-unique} &: \{A B : \text{Set}\} \to \text{Rel} A B \to \text{Set} \\
\text{left-unique} R &= \forall \{a b b'\} \rightarrow (a, b) \in R \rightarrow (a, b') \in R \rightarrow b \equiv b' \\
\Rightarrow &: \text{Set} \to \text{Set} \to \text{Set} \\
A \Rightarrow B &= r \in \text{Rel} A B \land \text{left-unique} r
\end{align*}
\]

1.6 Superscripts and other special notations

In the current version of this specification, superscript letters are heavily used for things such as disambiguations or type conversions. These are essentially meaningless, only present for technical reasons and can safely be ignored. However there are the two exceptions:

- \(^l\) for left-biased union
- \(^c\) in the context of set restrictions, where it indicates the complement

Also, non-letter superscripts do carry meaning.

At some point in the future we hope to be able to remove all those non-essential superscripts. Since we prefer doing this by changing the Agda source code instead of via hiding them in this document, this is a non-trivial problem that will take some time to address.

Additionally, there are some ? and ¿ operations. These relate to decision procedures and can also safely be ignored. We also plan on refactoring the code in such a way that they should disappear from this document.
2 Notation

In this section, we introduce some notations used in this document.

**Propositions, sets and types** This document loosely treats sets and types as the same thing. When we need to convert a list to its set of elements, we write `fromList`.

**Sums and products** The sum (or disjoint union, coproduct, etc.) of $A$ and $B$ is denoted by $A ⊔ B$, and their product is denoted by $A × B$. The projection functions from products are denoted $\text{proj}_1$ and $\text{proj}_2$, and the injections are denoted $\text{inj}_1$ and $\text{inj}_2$ respectively. The properties whether an element of a coproduct is in the left or right component are called $\text{isInj}_1$ and $\text{isInj}_2$.

**Record types** Record types are explained in Appendix A.

**Postfix projections** Projections can be written using postfix notation. For example, we may write $x \cdot \text{proj}_1$ instead of $\text{proj}_1 x$.

**Restriction, corestriction and complements** The restriction of a function or map $f$ to some domain $A$ is denoted by $f \mid A$, and the restriction to the complement of $A$ is written $f \mid A^c$. Corestriction or range restriction is denoted the same, except that $\mid$ is replaced by $\mid$.

**Inverse image** The expression $m^{-1} B$ denotes the inverse image of the set $B$ under the map $m$.

**Left-biased union** For maps $m$ and $m'$, we write $m ∪^l m'$ for their left-biased union. This means that key-value pairs in $m$ are guaranteed to be in the union, while key-value pairs in $m'$ will be in the union if and only if the keys don’t collide.

**Map addition** For maps $m$ and $m'$, we write $m ∪^* m'$ for their union, where keys that appear in both maps have their corresponding values added.
3 Cryptographic primitives

We rely on a public key signing scheme for verification of spending.

<table>
<thead>
<tr>
<th>Types &amp; functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>SKey VKey Sig Ser : Set</td>
</tr>
<tr>
<td>isKeyPair : SKey → VKey → Set</td>
</tr>
<tr>
<td>isSigned : VKey → Ser → Sig → Set</td>
</tr>
<tr>
<td>sign : SKey → Ser → Sig</td>
</tr>
</tbody>
</table>

KeyPair = Σ[ sk ∈ SKey ] Σ[ vk ∈ VKey ] isKeyPair sk vk

Property of signatures

((sk , vk , _) : KeyPair) (d : Ser) (σ : Sig) → sign sk d ≡ σ → isSigned vk d σ

Figure 3: Definitions for the public key signature scheme
4 Base types

<table>
<thead>
<tr>
<th>Coin</th>
<th>= N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slot</td>
<td>= N</td>
</tr>
<tr>
<td>Epoch</td>
<td>= N</td>
</tr>
</tbody>
</table>

**Figure 4**: Some basic types used in many places in the ledger
5 Token algebras

Abstract types

PolicyId

Derived types

record TokenAlgebra : Set₁ where
  field Value : Set
    ⟨ Value-IsCommutativeMonoid' ⟩ : IsCommutativeMonoid' ₀ ₀ Value

MemoryEstimate : Set
MemoryEstimate = ℕ

  field coin : Value → Coin
  inject : Coin → Value
  policies : Value → ℙ PolicyId
  size : Value → MemoryEstimate
  ⊥₁ₗ : Value → Value → Set
  AssetName : Set
  specialAsset : AssetName
  property : coin ∘ inject ≠ id -- FIXME: rename!
  coinIsMonoidHomomorphism : IsMonoidHomomorphism coin

Helper functions

  sumᵛ : List Value → Value
  sumᵛ [] = inject 0
  sumᵛ (x :: l) = x + sumᵛ l

Figure 5: Token algebras, used for multi-assets
6 Addresses

We define credentials and various types of addresses here. A credential contains a hash, either of a verifying (public) key ($isVKey$) or of a ($isScript$).

**Abstract types**

- **Network**
- **KeyHash**
- **ScriptHash**

**Derived types**

- **Credential** = **KeyHash** $\sqcup$ **ScriptHash**

```plaintext
record BaseAddr : Set where
  field net  : Network
  pay    : Credential
  stake  : Credential

record BootstrapAddr : Set where
  field net  : Network
  pay    : Credential
  attrsSize : ℕ

record RwdAddr : Set where
  field net  : Network
  stake  : Credential

VKeyBaseAddr     = Σ[ addr ∈ BaseAddr ] isVKey (addr . pay)
VKeyBootstrapAddr = Σ[ addr ∈ BootstrapAddr ] isVKey (addr . pay)
ScriptBaseAddr   = Σ[ addr ∈ BaseAddr ] isScript (addr . pay)
ScriptBootstrapAddr = Σ[ addr ∈ BootstrapAddr ] isScript (addr . pay)

Addr    = BaseAddr $\sqcup$ BootstrapAddr
VKeyAddr = VKeyBaseAddr $\sqcup$ VKeyBootstrapAddr
ScriptAddr = ScriptBaseAddr $\sqcup$ ScriptBootstrapAddr
```

**Helper functions**

- **payCred** : Addr $\to$ Credential
- **netId** : Addr $\to$ Network
- **isVKeyAddr** : Addr $\to$ Set
- **isScriptAddr** : Addr $\to$ Set

```plaintext
isVKeyAddr     = isVKey $\circ$ payCred
isScriptAddr   = isScript $\circ$ payCred
isScriptRwdAddr = isScript $\circ$ RwdAddr.stake
```

**Figure 6**: Definitions used in Addresses
7 Scripts

We define Timelock scripts here. They can verify the presence of keys and whether a transaction happens in a certain slot interval. These scripts are executed as part of the regular witnessing.

```
data Timelock : Set where
    RequireAllOf : List Timelock → Timelock
    RequireAnyOf : List Timelock → Timelock
    RequireMOf : ℕ → List Timelock → Timelock
    RequireSig : KeyHash → Timelock
    RequireTimeStart : Slot → Timelock
    RequireTimeExpire : Slot → Timelock

evalTimelock (khs : P KeyHash) (I : Maybe Slot × Maybe Slot) : Timelock → Set where
  evalAll : All (evalTimelock khs I) ss
          → (evalTimelock khs I) (RequireAllOf ss)
  evalAny : Any (evalTimelock khs I) ss
          → (evalTimelock khs I) (RequireAnyOf ss)
  evalMOf : MOf m (evalTimelock khs I) ss
          → (evalTimelock khs I) (RequireMOf m ss)
  evalSig : x ∈ khs
          → (evalTimelock khs I) (RequireSig x)
  evalTSt : M.Any (a ≤_ a) (I .proj₁)
          → (evalTimelock khs I) (RequireTimeStart a)
  evalTEx : M.Any (_≤ a) (I .proj₂)
          → (evalTimelock khs I) (RequireTimeExpire a)
```

Figure 7: Timelock scripts and their evaluation
8 Protocol parameters

This section defines the adjustable protocol parameters of the Cardano ledger. These parameters are used in block validation and can affect various features of the system, such as minimum fees, maximum and minimum sizes of certain components, and more.

The `Acnt` record has two fields, `treasury` and `reserves`, so the `acnt` field in `NewEpochState` keeps track of the total assets that remain in treasury and reserves.

```markdown
record Acnt : Set where
    field treasury reserves : Coin

ProtVer : Set
ProtVer = ℕ × ℕ

data pvCanFollow : ProtVer → ProtVer → Set where
    canFollowMajor : pvCanFollow (m , n) (m + 1 , 0)
    canFollowMinor : pvCanFollow (m , n) (m , n + 1)
```

Figure 8: Definitions related to protocol parameters

`PParams` contains parameters used in the Cardano ledger, which we group according to the general purpose that each parameter serves.

- **NetworkGroup**: parameters related to the network settings;
- **EconomicGroup**: parameters related to the economic aspects of the ledger;
- **TechnicalGroup**: parameters related to technical settings;
- **GovernanceGroup**: parameters related to governance settings;
- **SecurityGroup**: parameters that can impact the security of the system.

The first four groups have the property that every protocol parameter is associated to precisely one of these groups. The `SecurityGroup` is special: a protocol parameter may or may not be in the `SecurityGroup`. So, each protocol parameter belongs to at least one and at most two groups. Note that in [2] there is no `SecurityGroup`, but there is the concept of security-relevant protocol parameters. The difference between these notions is only social, so we implement security-relevant protocol parameters as a group.

The purpose of the groups is to determine voting thresholds for proposals aiming to change parameters. The thresholds depend on the groups of the parameters contained in such a proposal.

These new parameters are declared in Figure 9 and denote the following concepts.

- **drepThresholds**: governance thresholds for DReps; these are rational numbers named `P1`, `P2a`, `P2b`, `P3`, `P4`, `P5a`, `P5b`, `P5c`, `P5d`, and `P6`;
- **poolThresholds**: pool-related governance thresholds; these are rational numbers named `Q1`, `Q2a`, `Q2b`, `Q4` and `Q5e`;
- **ccMinSize**: minimum constitutional committee size;
- **ccMaxTermLength**: maximum term limit (in epochs) of constitutional committee members;
- **govActionLifetime**: governance action expiration;

12
data PParamGroup : Set where

record DrepThresholds : Set where
  field P1 P2a P2b P3 P4 P5a P5b P5c P5d P6 : ℚ

record PoolThresholds : Set where
  field Q1 Q2a Q2b Q4 Q5e : ℚ

record PParams : Set where
  field
  Network group
    maxBlockSize maxTxSize : ℕ
    maxHeaderSize maxValSize : ℕ
    maxCollateralInputs maxTxExUnits : ExUnits
  Economic group
    a b : ℕ
    keyDeposit : Coin
    poolDeposit : Coin
    coinsPerUTxOByte : Coin
    minFeeRefScriptCoinsPerByte : ℚ
    prices : Prices
  Technical group
    a0 : ℚ
    Emax : Epoch
    nopt : ℕ
    collateralPercentage : ℕ
    -- costmdls : Language →/⇀ CostModel (Does not work with DecEq)
    costmdls : CostModel
  Governance group
    drepThresholds : DrepThresholds
    poolThresholds : PoolThresholds
    govActionLifetime : ℕ
    govActionDeposit drepDeposit : Coin
    drepActivity : Epoch
    ccMinSize ccMaxTermLength : ℕ

paramsWellFormed : PParams → Set
paramsWellFormed pp =
  ⊥  ∉ fromList ( maxBlockSize :: maxTxSize :: maxHeaderSize :: maxValSize
                 :: minUTxOValue :: poolDeposit :: collateralPercentage :: ccMaxTermLength
                 :: govActionLifetime :: govActionDeposit :: drepDeposit :: [] )
  where open PParams pp

Figure 9: Protocol parameter declarations

- govActionDeposit: governance action deposit;
- drepDeposit: DRep deposit amount;
- `drepActivity`: DRep activity period;
- `minimumAVS`: the minimum active voting threshold.

Figure 9 also defines the function `paramsWellFormed`. It performs some sanity checks on protocol parameters.

Finally, to update parameters we introduce an abstract type. An update can be applied and it has a set of groups associated with it. An update is well formed if it has at least one group (i.e. if it updates something) and if it preserves well-formedness.

### Abstract types & functions

```
abstract types & functions

`UpdateT` : Set
`applyUpdate` : `PParams` → `UpdateT` → `PParams`
`updateGroups` : `UpdateT` → P `PParamGroup`

Well-formedness condition

```cpp
ppdWellFormed : `UpdateT` → Set
ppdWellFormed u = `updateGroups` u ≠ ∅
∀ pp → `paramsWellFormed` pp → `paramsWellFormed` (`applyUpdate` pp u)
```

**Figure 10:** Abstract type for parameter updates
9 Governance actions

We introduce three distinct bodies that have specific functions in the new governance framework:

1. a constitutional committee (henceforth called \texttt{CC})
2. a group of delegate representatives (henceforth called \texttt{DReps})
3. the stake pool operators (henceforth called \texttt{SPOs})

In the following figure, \texttt{DocHash} is abstract but in the implementation it will be instantiated with a 32-bit hash type (like e.g. \texttt{ScriptHash}). We keep it separate because it is used for a different purpose.

```
data GovRole : Set where
  CC DRep SPO : GovRole
Voter = GovRole × Credential
GovActionID = TxId × ℕ

data VDeleg : Set where
  credVoter : GovRole → Credential → VDeleg
  abstainRep = VDeleg
  noConfidenceRep : VDeleg

record Anchor : Set where
  field url : String
  hash : DocHash

data GovAction : Set where
  NoConfidence : GovAction
  NewCommittee : (Credential → Epoch) → P Credential → Q → GovAction
  NewConstitution : DocHash → Maybe ScriptHash → GovAction
  TriggerHF : ProtVer → GovAction
  ChangePParams : PParamsUpdate → GovAction
  TreasuryWdrl : (RwdAddr → Coin) → GovAction
  Info : GovAction

actionWellFormed : GovAction → Set
actionWellFormed (ChangePParams x) = ppdWellFormed x
actionWellFormed _ = ⊤
```

Figure 11: Governance actions

Figure 11 defines several data types used to represent governance actions including:

- \texttt{GovActionID}—a unique identifier for a governance action, consisting of the \texttt{TxId} of the proposing transaction and an index to identify a proposal within a transaction;

- \texttt{GovRole} (\textit{governance role})—one of three available voter roles defined above (\texttt{CC}, \texttt{DRep}, \texttt{SPO});

- \texttt{VDeleg} (\textit{voter delegation})—one of three ways to delegate votes: by credential, abstention, or no confidence (\texttt{credVoter}, \texttt{abstainRep}, or \texttt{noConfidenceRep});
• **Anchor**—a url and a document hash;

• **GovAction** (*governance action*)—one of seven possible actions (see Figure 12 for definitions).

• **actionWellFormed**—in the case of protocol parameter changes, an action is well-formed if it preserves the well-formedness of parameters. **ppdWellFormed** is effectively the same as **paramsWellFormed**, except that it only applies to the parameters that are being changed.

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NoConfidence</td>
<td>a motion to create a <em>state of no-confidence</em> in the current constitutional committee</td>
</tr>
<tr>
<td>NewCommittee</td>
<td>changes to the members of the constitutional committee and/or to its signature threshold and/or terms</td>
</tr>
<tr>
<td>NewConstitution</td>
<td>a modification to the off-chain Constitution and the proposal policy script</td>
</tr>
<tr>
<td>TriggerHF(^1)</td>
<td>triggers a non-backwards compatible upgrade of the network; requires a prior software upgrade</td>
</tr>
<tr>
<td>ChangePParams</td>
<td>a change to <em>one or more</em> updatable protocol parameters, excluding changes to major protocol versions (“hard forks”)</td>
</tr>
<tr>
<td>TreasuryWdrl</td>
<td>movements from the treasury</td>
</tr>
<tr>
<td>Info</td>
<td>an action that has no effect on-chain, other than an on-chain record</td>
</tr>
</tbody>
</table>

**Figure 12:** Types of governance actions

9.1 Hash protection

For some types of governance actions, enactment requires a second condition on top of the necessary votes, which is that the state after enacting the proposal was intended when the action was submitted. This is achieved by requiring actions to unambiguously link to the state they are modifying via a pointer to the previous modification. A proposal can only be enacted if it contains the **GovActionID** of the previously enacted proposal modifying the same piece of state. **NoConfidence** and **NewCommittee** modify the same state, while every other type of governance action has its own state that isn’t shared with any other action. This means that the enactibility of a proposal can change when other proposals are enacted.

However, not all types of governance actions require this strict protection. For **TreasuryWdrl** and **Info**, enacting them does not change the state in non-commutative ways, so they can always be enacted.

Types related to this hash protection scheme are defined in Figure 13. \(\top\) is the unit type that has exactly one element, which reflects that a **GovActionID** is not necessary.

9.2 Votes and proposals

The data type **Vote** represents the different voting options: *yes*, *no*, or *abstain*. For a **Vote** to be cast, it must be packaged together with further information, such as who votes and for which governance action. This information is combined in the **GovVote** record. An optional **Anchor** can be provided to give context about why a vote was cast in a certain manner.

---

\(^1\)There are many varying definitions of the term “hard fork” in the blockchain industry. Hard forks typically refer to non-backwards compatible updates of a network. In Cardano, we formalize the definition slightly more by calling any upgrade that would lead to *more blocks* being validated a “hard fork” and force nodes to comply with the new protocol version, effectively obsoleting nodes that are unable to handle the upgrade.
To propose a governance action, a GovProposal needs to be submitted. Beside the proposed action, it requires:

- potentially a pointer to the previous action (see Section 9.1),
- potentially a pointer to the proposal policy (if one is required),
- a deposit, which will be returned to returnAddr, and
- an Anchor, providing further information about the proposal.

While the deposit is held, it is added to the deposit pot, similar to stake key deposits. It is also counted towards the stake of the reward address to which it will be returned, so as not to reduce the submitter’s voting power when voting on their own (and competing) actions. For a proposal to be valid, the proposal must be set to the current value of govActionDeposit. The deposit will be returned when the action is removed from the state in any way.

GovActionState is the state of an individual governance action. It contains the individual votes, its lifetime, and information necessary to enact the action and to repay the deposit.
data Vote : Set where
     yes no abstain : Vote

record GovVote : Set where
    field gid      : GovActionID
    voter         : Voter
    vote          : Vote
    anchor        : Maybe Anchor

record GovProposal : Set where
    field action   : GovAction
    prevAction     : NeedsHash action
    policy         : Maybe ScriptHash
    deposit        : Coin
    returnAddr     : RwdAddr
    anchor         : Anchor

record GovActionState : Set where
    field votes    : Voter → Vote
    returnAddr     : RwdAddr
    expiresIn      : Epoch
    action         : GovAction
    prevAction     : NeedsHash action

Figure 14: Vote and proposal types
10 Transactions

Transactions are defined in Figure 15. A transaction is made up of a transaction body, a collection of witnesses and some optional auxiliary data. Some key ingredients in the transaction body are:

- A set of transaction inputs, each of which identifies an output from a previous transaction. A transaction input consists of a transaction id and an index to uniquely identify the output.
- An indexed collection of transaction outputs. The \texttt{TxOut} type is an address paired with a coin value.
- A transaction fee. This value will be added to the fee pot.
- The size and the hash of the serialized form of the transaction that was included in the block.
Abstract types

Ix TxId AuxiliaryData : Set

Derived types

TxIn   = TxId × Ix
TxOut  = Addr × Value × Maybe (Datum ⊎ DataHash) × Maybe Script
UTxO   = TxIn ⊸ TxOut
Wdrl   = RwdAddr ⊸ Coin
RdmrPtr = Tag × Ix

ProposedPPUpdates = KeyHash ⊸ PParamsUpdate
Update        = ProposedPPUpdates × Epoch

Transaction types

record TxBody : Set where
  field txins  : P TxIn
              refInputs  : P TxIn
              txouts    : Ix ⊸ TxOut
              txfee     : Coin
              mint      : Value
              txFldt    : Maybe Slot × Maybe Slot
              txcerts   : List DCert
              txwdrls   : Wdrl
              txvote    : List GovVote
              txprop    : List GovProposal
              txdonation: Coin
              txup      : Maybe Update
              txAEDhash : Maybe ADHash
              netwrk    : Maybe Network
              txsize    : N
              txid      : TxId
              collateral: P TxIn
              reqSigHash: P KeyHash
              scriptIntHash: Maybe ScriptHash

record TxWitnesses : Set where
  field vkSigs  : VKey ⊸ Sig
               scripts : P Script
               txdats  : DataHash ⊸ Datum
               txrdmrs : RdmrPtr ⊸ Redeemer × ExUnits

scriptsP1 : P P1Script
scriptsP1 = mapPartial isInj₁ scripts

record Tx : Set where
  field body    : TxBody
                 wits     : TxWitnesses
                 isValid : Bool
                 txAED    : Maybe AuxiliaryData

Figure 15: Transactions and related types
getValue : TxOut → Value
getValue (_, v, _) = v

TxOutʰ = Addr × Value × Maybe (Datum ⊔ DataHash) × Maybe ScriptHash

txOutHash : TxOut → TxOutʰ
txOutHash (a, v, d, s) = a, (v, (d, M.map hash s))

getValueʰ : TxOutʰ → Value
getValueʰ (_, v, _) = v

txinsVKey : ℙ TxIn → UTxO → ℙ TxIn
txinsVKey txins utxo = txins ∩ dom (utxo ↦ (isVKeyAddr ◦ proj₁))

scriptOuts : UTxO → UTxO
scriptOuts utxo = filter (λ (_, addr, _) → isScriptAddr addr) utxo

txinsScript : ℙ TxIn → UTxO → ℙ TxIn
txinsScript txins utxo = txins ∩ dom (proj₁ (scriptOuts utxo))

refScripts : Tx → UTxO → ℙ Script
refScripts tx utxo =
    mapPartial (proj₂ ◦ proj₂ ◦ proj₂) (range (utxo | (txins ∪ refInputs)))
    where open Tx; open TxBody (tx.body)

txscripts : Tx → UTxO → ℙ Script
txscripts tx utxo = scripts (tx.wits) ∪ refScripts tx utxo
    where open Tx; open TxWitnesses

lookupScriptHash : ScriptHash → Tx → UTxO → Maybe Script
lookupScriptHash sh tx utxo =
    if sh ∈ mapʰ proj₁ (mʰ) then
        just (lookupʰ m sh)
    else
        nothing
    where m = setToHashMap (txscripts tx utxo)

Figure 16: Functions related to transactions
11 UTxO

11.1 Accounting

```plaintext
isTwoPhaseScriptAddress : Tx → UTxO → Addr → Bool
isTwoPhaseScriptAddress tx utxo a =
  if isScriptAddr a then
    (λ {p} → if lookupScriptHash (getScriptHash a p) tx utxo
        then (λ {s} → isP2Script s)
        else false)
  else false

getDataHashes : P TxOut → ℙ DataHash
getDataHashes txo = mapPartial isInj₂ (mapPartial (proj₁ ∘ proj₂ ∘ proj₂) txo)

getInputHashes : Tx → UTxO → ℙ DataHash
getInputHashes tx utxo = getDataHashes (filterˢ (λ (a , _) → isTwoPhaseScriptAddress tx utxo a ≡ true)
  (range (utxo | txins)))
  where open Tx; open TxBody (tx .body)

totExUnits : Tx → ExUnits
totExUnits tx = ∑[ (_, eu) ← tx .wits .txrdmr ] eu
  where open Tx; open TxWitnesses
```

**Figure 17**: Functions supporting UTxO rules

Figures 17, 19, and 20 define functions needed for the UTxO transition system. Note the special multiplication symbol \(*↓\) used in Figure 19: it means multiply and round down the result.

Figure 18 defines the types needed for the UTxO transition system. The UTxO transition system is given in Figure 22.

- The function `outs` creates the unspent outputs generated by a transaction. It maps the transaction id and output index to the output.
- The `balance` function calculates sum total of all the coin in a given UTxO.

The deposits have been reworked since the original Shelley design. We now track the amount of every deposit individually. This fixes an issue in the original design: An increase in deposit amounts would allow an attacker to make lots of deposits before that change and refund them after the change. The additional funds necessary would have been provided by the treasury. Since changes to protocol parameters were (and still are) known publicly and guaranteed before they are enacted, this comes at zero risk for an attacker. This means the deposit amounts could realistically never be increased. This issue is gone with the new design.

Similar to `ScriptPurpose`, `DepositPurpose` carries the information what the deposit is being made for. The deposits are stored in the `deposits` field of `UTxOState`. `updateDeposits` is responsible for updating this map, which is split into `updateCertDeposits` and `updateProposalDeposits`, responsible for certificates and proposals respectively. Both of these functions iterate over the
relevant fields of the transaction body and insert or remove deposits depending on the information seen. Note that some deposits can only be refunded at the epoch boundary and are not removed by these functions.

**Derived types**

```haskell
data DepositPurpose : Set where
    CredentialDeposit : Credential → DepositPurpose
    PoolDeposit : Credential → DepositPurpose
    DRepDeposit : Credential → DepositPurpose
    GovActionDeposit : GovActionID → DepositPurpose

Deposits = DepositPurpose → Coin
```

**UTxO environment**

```haskell
record UTxOEnv : Set where
    field slot : Slot
    pparams : PParams
```

**UTxO states**

```haskell
record UTxOState : Set where
    constructor [_,_,_,_]ᵘ
    field utxo : UTxO
    fees : Coin
    deposits : Deposits
    donations : Coin
```

**UTxO transitions**

```haskell
⟦_,_,_,_⟧⁻⁺ UTxO₀ : UTxOEnv → UTxOState → Tx → UTxOState → Set
```

**Figure 18:** UTxO transition-system types

We redefine `depositRefunds` and `newDeposits` via `depositsChange`, which computes the difference between the total deposits before and after their application. This simplifies their definitions and some correctness proofs. We then add the absolute value of `depositsChange` to `consumed` or `produced` depending on its sign. This is done via `negPart` and `posPart`, which satisfy the key property that their difference is the identity function.
outs : TxBody → UTxO
outs tx = mapKeys (tx .txid ,_) (tx .txouts)

balance : UTxO → Value
balance utxo = ∑[ x ← mapValues txOutHash utxo ] getValueʰ x

cbalance : UTxO → Coin
cbalance utxo = coin (balance utxo)

minfee : PParams → UTxO → Tx → Coin
minfee pp utxo tx =

pp .a * tx .body .txsize + pp .b
+ txscriptfee (pp .prices) (totExUnits tx)
+ pp .minFeeRefScriptCoinsPerByte
×↓ ∑[ x ← mapValues scriptSize (setToHashMap (refScripts tx utxo)) ] x

updateCertDeposits : PParams → List DCert → Deposits → Deposits
updateCertDeposits _ [] deposits = deposits
updateCertDeposits pp (cert :: certs) deposits
= (updateCertDeposits pp certs deposits U* certDeposit cert pp) | certRefund cert ĉ

where
certDeposit : DCert → PParams → Deposits
certDeposit (delegate c _ v) ĉ = { CredentialDeposit c , v ĉ }
certDeposit (regpool c _ ĉ pp = { PoolDeposit c , pp .poolDeposit }
certDeposit (regdrep c v ĉ _ = { DRepDeposit c , v ĉ }
certDeposit ĉ _ ĉ = ĉ

certRefund : DCert → P DepositPurpose
certRefund (dereg c ĉ = { CredentialDeposit c ĉ }
certRefund (deregdrep c ĉ = { DRepDeposit c ĉ }
certRefund ĉ = ĉ

updateProposalDeposits : List GovProposal → TxId → Coin → Deposits → Deposits
updateProposalDeposits [] _ _ _ deposits = deposits
updateProposalDeposits (_ :: ps) txid gaDep deposits =
updateProposalDeposits ps txid gaDep deposits
U* { GovActionDeposit (txid , length ps) , gaDep }

updateDeposits : PParams → TxBody → Deposits → Deposits
updateDeposits pp txb = updateCertDeposits pp txcerts
⊙ updateProposalDeposits txprop txid (pp .govActionDeposit)

depositsChange : PParams → TxBody → Deposits → Z
depositsChange pp txb deposits =
getCoin (updateDeposits pp txb deposits) = getCoin deposits

Figure 19: Functions used in UTxO rules
data inInterval (slot : Slot) : (Maybe Slot × Maybe Slot) → Set where
  both : ∀ {l r} → l ≤ slot × slot ≤ r → inInterval slot (just l , just r)
  lower : ∀ {l} → l ≤ slot → inInterval slot (just l , nothing)
  upper : ∀ {r} → slot ≤ r → inInterval slot (nothing , just r)
  none : inInterval slot (nothing , nothing)

feesOK : PParams → Tx → UTxO → Bool
feesOK pp tx utxo = minfee pp utxo tx ≤ᵇ txfee
  ∧ not (Δ⁻ᵇ (txrdmr s ^))
  ⇒ᵇ (allᵇ (λ (addr , _) → addr isVKeyAddr) collateralRange
     ∧ isAdaOnlyᵇ bal
     ∧ (coin bal * 100) ≥ᵇ (txfee * pp .collateralPercentage)
     ∧ not (Δ⁻ᵇ collateral))

  where
  open Tx tx; open TxBODY body; open TxBodies wits; open PParams pp
  collateralRange = range ((mapValues txOutHash utxo) | collateral)
  bal = balance (utxo | collateral)

< Figure 20: Functions used in UTxO rules, continued >

depositRefunds : PParams → UTxOState → TxBODY → Coin
depositRefunds pp st txb = negPart (depositsChange pp txb (st .deposits))

newDeposits : PParams → UTxOState → TxBODY → Coin
newDeposits pp st txb = posPart (depositsChange pp txb (st .deposits))

cConsumed : PParams → UTxOState → TxBODY → Value
cConsumed pp st txb
  = balance (st .utxo | txb .txins)
  + txb .mint
  + inject (depositRefunds pp st txb)

produced : PParams → UTxOState → TxBODY → Value
produced pp st txb
  = balance (outs txb)
  + inject (txb .txfee)
  + inject (newDeposits pp st txb)
  + inject (txb .txdonation)

< Figure 21: Functions used in UTxO rules, continued >
UTXO-inductive:

\[
\begin{align*}
& \text{let open Tx tx renaming (body to txb); open TxBody txb} \\
& \quad \text{open UTxOEnv } \Gamma \text{ renaming (pparams to pp)} \\
& \quad \text{open UTxOState } s \\
& \quad \text{txouts}^h = (\text{mapValues txOutHash txouts}) \\
& \text{in} \\
& \quad \text{txins} \not= \emptyset \\
& \quad \text{txins} \cap \text{refInputs} \not= \emptyset \\
& \quad \text{txins} \cup \text{refInputs} \subseteq \text{dom utxo} \\
& \quad \text{fesOK } \text{pp } \text{tx } \text{utxo} \equiv \text{true} \\
& \quad \text{consumed } \text{pp } \text{s } \text{txb} \equiv \text{produced } \text{pp } \text{s } \text{txb} \\
& \quad \text{coin mint} \equiv 0 \\
& \quad \text{txsize} \leq \text{maxTxSize } \text{pp} \\
& \quad \forall (\_, \text{txout}) \in \text{txouts}^h. \text{proj}_1 \\
& \quad \text{inInterval } \text{slot } \text{txvldt} \\
& \quad \forall (\_, \text{txout}) \in \text{txouts}^h. \text{proj}_1 \\
& \quad \text{serSize } (\text{getValue}^h \text{txout}) \leq \text{maxValSize } \text{pp} \\
& \quad \forall (a, \_ ) \in \text{range } \text{txouts}^h \\
& \quad \text{Sum.All } (\text{const } \top) (\lambda a \rightarrow a.\text{BootstrapAddr.attrsSize} \leq 64) a \\
& \quad \forall (a, \_ ) \in \text{range } \text{txouts}^h. \text{netId } a \equiv \text{networkId} \\
& \quad \forall (a \in \text{dom txwdrls }) \quad a.\text{RwdAddr.net} \equiv \text{networkId} \\
& \quad \Gamma \vdash s \triangleq \{ \text{tx}, \text{UTXO} \} s' \\
\end{align*}
\]

\[\Gamma \vdash s \triangleq \{ \text{tx}, \text{UTXO} \} s'\]

\textbf{Figure 22: UTXO inference rules}
11.2 Witnessing

Figure 23 defines functions used for witnessing. \texttt{witsVKeyNeeded} and \texttt{scriptsNeeded} are now defined by projecting the same information out of \texttt{credsNeeded}. Note that the last component of \texttt{credsNeeded} adds the script in the proposal policy only if it is present.

\begin{verbatim}
getVKeys : P Credential \to P KeyHash
getVKeys = mapPartial isInj₁

getScripts : P Credential \to P ScriptHash
getScripts = mapPartial isInj₂

credsNeeded : UTxO \to TxBody \to P (ScriptPurpose \times Credential)
credsNeeded utxo txb
    = mapˢ (\lambda (i, o) \to (Spend i, payCred (proj₁ o))) ((utxo \mid txins) $)
      \cup mapˢ (\lambda a \to (Rwrd a, Rwaddr.stake a)) (dom (txwdrls .proj₁))
      \cup mapˢ (\lambda c \to (Cert c, cwitness c)) (fromList txcerts)
      \cup mapˢ (\lambda x \to (Mint x, inj₂ x)) (policies mint)
      \cup mapˢ (\lambda v \to (Vote v, proj₂ v)) (fromList $ map GovVote.voter txvote)
    \cup mapPartial (\lambda p \to case p .GovProposal.policy of
                        (just sh) \to just (Propose p, inj₂ sh)
                        nothing \to nothing) (fromList txprop)

witsVKeyNeeded : UTxO \to TxBody \to P KeyHash
witsVKeyNeeded = getVKeys \circ₂ mapˢ proj₁ \circ₂ credsNeeded

scriptsNeeded : UTxO \to TxBody \to P ScriptHash
scriptsNeeded = getScripts \circ₂ mapˢ proj₂ \circ₂ credsNeeded
\end{verbatim}

\begin{figure}[h]
\centering
\begin{verbatim}
getVKeys : P Credential \to P KeyHash
getVKeys = mapPartial isInj₁

getScripts : P Credential \to P ScriptHash
getScripts = mapPartial isInj₂

credsNeeded : UTxO \to TxBody \to P (ScriptPurpose \times Credential)
credsNeeded utxo txb
    = mapˢ (\lambda (i, o) \to (Spend i, payCred (proj₁ o))) ((utxo \mid txins) $)
      \cup mapˢ (\lambda a \to (Rwrd a, Rwaddr.stake a)) (dom (txwdrls .proj₁))
      \cup mapˢ (\lambda c \to (Cert c, cwitness c)) (fromList txcerts)
      \cup mapˢ (\lambda x \to (Mint x, inj₂ x)) (policies mint)
      \cup mapˢ (\lambda v \to (Vote v, proj₂ v)) (fromList $ map GovVote.voter txvote)
    \cup mapPartial (\lambda p \to case p .GovProposal.policy of
                        (just sh) \to just (Propose p, inj₂ sh)
                        nothing \to nothing) (fromList txprop)

witsVKeyNeeded : UTxO \to TxBody \to P KeyHash
witsVKeyNeeded = getVKeys \circ₂ mapˢ proj₁ \circ₂ credsNeeded

scriptsNeeded : UTxO \to TxBody \to P ScriptHash
scriptsNeeded = getScripts \circ₂ mapˢ proj₂ \circ₂ credsNeeded
\end{verbatim}
\caption{Functions used for witnessing}
\end{figure}

\begin{verbatim}
ufunc_UTxOW_ : UTxOEnv \to UTxOState \to Tx \to UTxOState \to Set
\end{verbatim}

\begin{figure}[h]
\centering
\begin{verbatim}
ufunc_UTxOW_ : UTxOEnv \to UTxOState \to Tx \to UTxOState \to Set
\end{verbatim}
\caption{UTxOW transition-system types}
\end{figure}
UTXOW-inductive:
let open Tx tx renaming (body to txb); open TxBody txb; open TxWitnesses wits
   open UTxOState s
   witsKeyHashes = map^ hash (dom vkSigs)
   witsScriptHashes = map^ hash scripts
   inputHashes = getInputHashes tx utxo
   refScriptHashes = map^ hash (refScripts tx utxo)
   neededHashes = scriptsNeeded utxo txb
   txdatsHashes = dom txdats
   allOutHashes = getDataHashes (range txouts)
in
   ∀ [(vk, σ) ∈ vkSigs] isSigned vk (txidBytes txid) σ
   ∀ [s ∈ mapPartial isInj₁ (txscripts tx utxo)] validP1Script witsKeyHashes txvldt s
   witsVKKeyNeeded utxo txb ⊆ witsKeyHashes
   (neededHashes \ refScriptHashes) ≡⁺ witsScriptHashes
   inputHashes ⊆ txdatsHashes
   txdatsHashes ⊆ (inputHashes \ allOutHashes \ getDataHashes (range (utxo | refInputs)))
   txADhash = map hash txAD
   Γ ⊢ s ⊸ ⟨ tx,UTXO⟩ s’
______________
Γ ⊢ s ⊸ ⟨ tx,UTXOW⟩ s’

Figure 25: UTXOW inference rules
### 12 Governance

#### Derived types

```plaintext
GovState : Set
GovState = List (GovActionID × GovActionState)
```

```plaintext
record GovEnv : Set where
  constructor [_,_,_,_,_]ᵍ
    field txid : TxId
    epoch : Epoch
    pparams : PParams
    ppolicy : Maybe ScriptHash
    enactState : EnactState
```

#### Transition relation types

```plaintext
⊢ _↓ := _↓ Gov' ⊆ GovEnv × ℕ → GovState → GovVote ⊔ GovProposal → GovState → Set
⊢ _↓ := _↓ Gov ⊆ GovEnv → GovState → List (GovVote ⊔ GovProposal) → GovState → Set
```

#### Functions used in the GOV rules

- **addVote**:GovState → GovActionID → Voter → Vote → GovState
  ```plaintext
  addVote s aid voter v = map modifyVotes s
  where modifyVotes = λ (gid , s') → gid , record s'
    { votes = if gid ≡ aid then insert (votes s') voter v else votes s'}
  ```

  ```plaintext
  addAction s e aid addr a prev = s ::ʳ (aid , record
    { votes = ∅ ; returnAddr = addr ; expiresIn = e ; action = a ; prevAction = prev })
  ```

- **validHFAction**:GovProposal → GovState → EnactState → Set
  ```plaintext
  validHFAction (record { action = TriggerHF v ; prevAction = prev }) s e =
  (let (v' , oid) = EnactState.pv e in aid ≡ prev × pvCanFollow v' v)
  ∨ ∃₂ [ x , v' ] (prev , x) ∈ fromList s × x .action = TriggerHF v' × pvCanFollow v' v
  validHFAction _ _ _ ⊤
  ```

---

**Figure 26**: Types and functions used in the GOV transition system

*GovState* behaves similar to a queue. New proposals are appended at the end, but any proposal can be removed at the epoch boundary. However, for the purposes of enactment, earlier proposals take priority. Note that *EnactState* used in *GovEnv* is defined later, in Section 15.

- **addVote** inserts (and potentially overrides) a vote made for a particular governance action (identified by its ID) by a credential with a role.
- **addAction** adds a new proposed action at the end of a given *GovState*.
- **validHFAction** is the property whether a given proposal, if it is a *TriggerHF* action, can potentially be enacted in the future. For this to be the case, its *prevAction* needs to exist,

---

\(^2\) l :: x appends element x to list l.
be another TriggerHF action and have a compatible version.

The GOV transition system is now given as the reflexive-transitive closure of the system GOV’, described in Figure 28.

For GOV-Vote, we check that the governance action being voted on exists and the role is allowed to vote. canVote is defined in Figure 40. Note that there are no checks on whether the credential is actually associated with the role. This means that anyone can vote for, e.g., the CC role. However, during ratification those votes will only carry weight if they are properly associated with members of the constitutional committee.

For GOV-Propose, we check well-formedness, correctness of the deposit and some conditions depending on the type of the action:

- for ChangePParams or TreasuryWdrl, the proposal policy needs to be provided;
- for NewCommittee, no proposals with members expiring in the present or past epoch are allowed, and candidates cannot be added and removed at the same time;
GOV-Vote : ∀ \{x \text{ ast}\} \rightarrow \text{let}
    \begin{align*}
    &\text{open GovEnv } \Gamma \\
    &\text{sig} = \text{inj}_1 \text{ record } \{ \text{gid} = \text{aid} ; \text{voter} = \text{voter} ; \text{vote} = \text{v} ; \text{anchor} = x \} \\
    &\text{in}
    &\bullet (\text{aid} , \text{ast}) \in \text{fromList } s \\
    &\bullet \text{canVote } \text{pparams } (\text{action } \text{ast}) \ (\text{proj}_1 \text{voter})
    \end{align*}
(\Gamma , k) \vdash s \rightarrow \text{GOV'} \ \text{addVote } s \ \text{aid} \ \text{voter} \ \text{v}

GOV-Propose : ∀ \{x\} \rightarrow \text{let}
    \begin{align*}
    &\text{open GovEnv } \Gamma ; \text{open PParams } \text{pparams} \text{ hiding } (a) \\
    &\text{prop} = \text{record } \{ \text{returnAddr} = \text{addr} ; \text{action} = a ; \text{anchor} = x \\
    &\quad ; \text{policy} = p ; \text{deposit} = d ; \text{prevAction} = \text{prev} \} \\
    &s' = \text{addAction } s \ (\text{govActionLifetime} +^e \text{epoch}) \ (\text{txid} , k) \ \text{addr} a \ \text{prev}
    \end{align*}
in
\bullet \text{actionWellFormed } a \\
\bullet d \equiv \text{govActionDeposit} \\
\bullet (\exists [ u ] \ a \equiv \text{ChangePParams } u \ \forall [ w ] \ a \equiv \text{TreasuryWdrl } w \rightarrow p \equiv \text{ppolicy}) \\
\bullet (\forall \{\text{new } \text{rem } q\} \rightarrow a \equiv \text{NewCommittee } \text{new } \text{rem } q \\
\quad \rightarrow (\forall [ e \in \text{range } \text{new } ] \ \text{epoch} < e \times \text{dom } \text{new } \cap \text{rem} \equiv^e \emptyset) \\
\bullet \text{validHFAction } \text{prop } s \ 	ext{enactState} \\
\bullet \text{hasParent } \text{enactState } s \ \text{a } \ \text{prev}
\end{align*}
(\Gamma , k) \vdash s \rightarrow \text{GOV'} \ s'

\vdash \vdash \text{GOV'} = \text{ReflexiveTransitive Closure}_\vdash \vdash \text{GOV'}

\textbf{Figure 28}: Rules for the GOV transition system
• and we check the validity of hard-fork actions via `validHFAction`. 
13 Delegation

The rules for transition systems dealing with individual certificates are defined in Figure 32. GOVCERT deals with the new certificates relating to DReps and the constitutional committee.

- **GOVCERT-regdrep** registers (or re-registers) a DRep. In case of registration, a deposit needs to be paid. Either way, the activity period of the DRep is reset.

- **GOVCERT-deregdrep** deregisters a DRep.

- **GOVCERT-ccreghot** registers a hot credential for constitutional committee members. We check that the cold key did not previously resign from the committee. Note that we intentionally do not check if the cold key is actually part of the committee; if it isn’t, then the corresponding hot key does not carry any voting power. By allowing this, a newly elected member of the constitutional committee can immediately delegate their vote to a hot key and use it to vote. Since votes are counted after previous actions have been enacted, this allows constitutional committee members to act without a delay of one epoch.

Figure 33 assembles the CERTS transition system by bundling the previously defined pieces together into the CERT system, and then taking the reflexive-transitive closure of CERT together with CERTBASE as the base case. CERTBASE does the following:

- check the correctness of withdrawals and ensure that withdrawals only happen from credentials that have delegated their voting power;

- set the rewards of the credentials that withdrew funds to zero;

- and set the activity timer of all DReps that voted to $drepActivity$ epochs in the future.

```plaintext
record PoolParams : Set where
  field rewardAddr : Credential

data DCert : Set where
  delegate : Credential → Maybe VDeleg → Maybe Credential → Coin → DCert
  dereg : Credential → DCert
  regpool : Credential → PoolParams → DCert
  regdrep : Credential → Coin → Anchor → DCert
  deregdrep : Credential → DCert
  ccreghot : Credential → Maybe Credential → DCert

  cwitness : DCert → Credential
  cwitness (delegate c _ _ ) = c
  cwitness (dereg c) = c
  cwitness (regpool c _) = c
  cwitness (retirepool c _) = c
  cwitness (regdrep c _) = c
  cwitness (deregdrep c) = c
  cwitness (ccreghot c _) = c

Figure 29: Delegation definitions
```
record CertEnv : Set where
  constructor [_,_,_,_]ᶜ
  field epoch : Epoch
  pp : PParams
  votes : List GovVote
  wdlrs : RwdAddr -> Coin

record DState : Set where
  constructor [_,_,_]ᵈ
  field
  voteDelegs : Credential -> VDeleg
  stakeDelegs : Credential -> Credential
  rewards : Credential -> Coin

record PState : Set where
  constructor [_,_]ᵖ
  field pools : Credential -> PoolParams
  retiring : Credential -> Epoch

record GState : Set where
  constructor [_,_]ᵛ
  field dreps : Credential -> Epoch
  ccHotKeys : Credential -> Maybe Credential

record CertState : Set where
  constructor [_,_,_]ᶜˢ
  field dState : DState
  pState : PState
  gState : GState

record DelegEnv : Set where
  constructor [_,_]ᵈᵉ
  field pparams : PParams
  pools : Credential -> PoolParams

GovCertEnv = CertEnv
PoolEnv = PParams

Figure 30: Types used for CERTS transition system

getDRepVote : GovVote -> Maybe Credential
getDRepVote record { voter = (DRep, credential) } = just credential
getDRepVote _ = nothing

Figure 31: Functions used for CERTS transition system
\( \vdash \langle \_ , \text{DELEG} \rangle \_ : \text{DelegEnv} \to \text{DState} \to \text{DCert} \to \text{DState} \to \text{Set} \)

**DELEG-delegate**:
- let open PParams pp in
  - \( (c \not\in \text{dom } \text{rwds} \to d = \text{keyDeposit}) \)
  - \( (c \in \text{dom } \text{rwds} \to d = 0) \)
  - \( mc \in \text{map}^+ \text{ just } (\text{dom pools}) \cup \{ \text{nothing} \} \)

\[ [ pp \ , \ pools ]^d \vdash [ v\text{Delegs} \ , \ s\text{Delegs} \ , \ \text{rwds} ]^d \rightarrow [ \text{delegate } c \text{ } mv \text{ } d ]\text{ ,DELEG}\]
\[ [ \text{insertIfJust } c \text{ } mv \text{ } v\text{Delegs} \ , \ \text{insertIfJust } c \text{ } mc \text{ } s\text{Delegs} \ , \ \text{rwds} \cup^l \{ c \ , \ 0 \} ]^d \]

**DELEG-dereg**:
- \( (c , 0) \in \text{rwds} \)

\[ [ pp \ , \ pools ]^d \vdash [ v\text{Delegs} \ , \ s\text{Delegs} \ , \ \text{rwds} ]^d \rightarrow [ \text{dereg } c ]\text{ ,DELEG}\]
\[ [ v\text{Delegs} | \{ c \}^c , \ s\text{Delegs} | \{ c \}^c , \ \text{rwds} | \{ c \}^c ]^d \]

\( \vdash \langle \_ , \text{POOL} \rangle \_ : \text{PoolEnv} \to \text{PState} \to \text{DCert} \to \text{PState} \to \text{Set} \)

**POOL-regpool**:
- \( c \not\in \text{dom } \text{pools} \)

\[ pp \vdash [ pools , \text{retiring} ]^p \rightarrow [ \text{regpool } c \text{ } \text{poolParams} ]\text{ ,POOL}\]
\[ [ \{ c \ , \ \text{poolParams} \} \cup^l \text{pools} , \text{retiring} ]^p \]

**POOL-retirepool**:

\[ pp \vdash [ pools , \text{retiring} ]^p \rightarrow [ \text{retirepool } c \text{ } e ]\text{ ,POOL}\]
\[ [ \text{pools} , \{ c \ , \ e \} \cup^l \text{retiring} ]^p \]

\( \vdash \langle \_ , \text{GOVCERT} \rangle \_ : \text{GovCertEnv} \to \text{GState} \to \text{DCert} \to \text{GState} \to \text{Set} \)

**GOVCERT-regdrep**:
- let open PParams pp in
  - \( (d \equiv \text{drepDeposit} \times c \not\in \text{dom } \text{dReps}) \uplus (d = 0 \times c \in \text{dom } \text{dReps}) \)

\[ [ e , pp , vs , \text{wdrls} ]^c \vdash [ \text{dReps} , \text{ccKeys} ]^y \rightarrow [ \text{regdrep } d \text{ an } ]\text{ ,GOVCERT}\]
\[ [ \{ c \ , \ e + \text{drepActivity} \} \cup^l \text{dReps} , \text{ccKeys} ]^y \]

**GOVCERT-deregdrep**:
- \( c \in \text{dom } \text{dReps} \)

\[ \Gamma \vdash [ \text{dReps} , \text{ccKeys} ]^y \rightarrow [ \text{deregdrep } c ]\text{ ,GOVCERT}\]
\[ [ \text{dReps} | \{ c \}^c , \text{ccKeys} ]^y \]

**GOVCERT-ccreghot**:
- \( (c , \text{nothing}) \not\in \text{ccKeys} \)

\[ \Gamma \vdash [ \text{dReps} , \text{ccKeys} ]^y \rightarrow [ \text{ccreghot } c \text{ } mc ]\text{ ,GOVCERT}\]
\[ [ \text{dReps} , \{ c \ , \ mc \} \cup^l \text{ccKeys} ]^y \]

**Figure 32:** Auxiliary DELEG, POOL and GOVCERT transition systems
CERT-env : CertEnv → CertState → DCert → CertState → Set

CERT-deleg :
  • [ pp, PState.pools stP ]ᵈ → stᵈ → ΔCert,DELEG stᵈ′


CERT-pool :
  • pp ⊢ stP → ΔCert,POOL stP′


CERT-vdel :
  • Γ ⊢ stᵍ → ΔCert,GOVCERT stᵍ′

  Γ ⊢ [ stᵈ, stP, stᵍ ]ᶜˢ → ΔCert,CERT [ stᵈ′, stP, stᵍ′ ]ᶜˢ

CERT-base :
  let open PParams pp; open GState stᵍ; open DState stᵈ
  refresh = mapPartial getDRepVote (fromList vs)
  refreshedDReps = mapValueRestricted (const (e + drepActivity)) dreps refresh
  wdrlCreds = mapˢ RwdAddr.stake (dom wdrls)
  in
  • wdrlCreds ⊆ dom voteDelegs
  • mapˢ (map, RwdAddr.stake) (wdrls ᵈ) ⊆ rewards ᵈ

  [ e, pp, vs, wdrls ]ᶜ ⊢ [ stᵈ, stP, stᵍ ]ᶜˢ → ΔCERTBASE
  [ [ voteDelegs, stakeDelegs, constMap wdrlCreds 0 ᵈ rewards ᵈ ], stP
  , [ refreshedDReps, ccHotKeys ]ᵛ ]ᶜˢ

CERTS : CertEnv → CertState → List DCert → CertState → Set
CERTS = ReflexiveTransitiveClosureᵇ ⊢ ΔCERTBASE ⊢ ΔCERT

Figure 33: CERTS rules
The entire state transformation of the ledger state caused by a valid transaction can now be given as a combination of the previously defined transition systems.

```
record LEnv : Set where
  constructor [_,_,_,_]⁰
  field slot : Slot
    ppolicy : Maybe ScriptHash
    pparams : PParams
    enactState : EnactState

record LState : Set where
  constructor [_,_,_]¹
  field utxoSt : UTxOState
    govSt : GovState
    certState : CertState

txgov : TxBody → List (GovVote ⊎ GovProposal)
 txgov txb = map inj₁ txvote ++ map inj₂ txprop
  where open TxBody txb
```

**Figure 34:** Types and functions for the LEDGER transition system

```
_⊢_⇀⦇_,LEDGER⦈_ : LEnv → LState → Tx → LState → Set
```

**Figure 35:** The type of the LEDGER transition system
**LEDGER-V** : let open LState s; txb = tx.body; open TxBody txb; open LEnv Γ in
  * isValid tx = true
  * record { LEnv Γ } ⊢ utxoSt ≫ tx,UTXOW utxoSt'
  * [ epoch slot, pparams, txvote, txwdrls ]ᶜ ⊢ certState ≫ txcerts,CERTS certState'
  * [ txid, epoch slot, pparams, ppolicy, enactState ]ᶜ ⊢ govSt ≫ txgov txb,GOV govSt'

Γ ⊢ s ≫ tx,LEDGER [ utxoSt', govSt', certState' ]¹

**LEDGER-I** : let open LState s; txb = tx.body; open TxBody txb; open LEnv Γ in
  * isValid tx = false
  * record { LEnv Γ } ⊢ utxoSt ≫ tx,UTXOW utxoSt'

Γ ⊢ s ≫ tx,LEDGER [ utxoSt', govSt, certState ]¹

---

**Figure 36:** LEDGER transition system

---

**Figure 37:** LEDGERS transition system
15 Enactment

Figure 38 contains some definitions required to define the ENACT transition system. EnactEnv is the environment and EnactState the state of ENACT, which enacts a governance action. All governance actions except TreasuryWdrl and Info modify EnactState permanently, which of course can have further consequences. TreasuryWdrl accumulates withdrawal temporarily in EnactState, but this information is applied and discarded immediately in EPOCH. Also, enacting these governance actions is the only way of modifying EnactState. The withdrawals field of EnactState is special in that it is ephemeral—ENACT accumulates withdrawals there which are paid out at the next epoch boundary where this field will be reset.

Note that all other fields of EnactState also contain a GovActionID since they are HashProtected.

```haskell
record EnactEnv : Set where
  constructor [_,_,_] 
  field gid : GovActionID
   treasury : Coin
   epoch : Epoch

record EnactState : Set where
  field cc : HashProtected (Maybe ((Credential - Epoch) × ℚ))
  constitution : HashProtected (DocHash × Maybe ScriptHash)
  pv : HashProtected ProtVer
  pparams : HashProtected PParams
  withdrawals : RwdAddr - Coin

ccCreds : HashProtected (Maybe ((Credential - Epoch) × ℚ)) → P Credential
ccCreds (just x , _) = dom (x .proj₁)
ccCreds (nothing , _) = ∅

getHash : ∀ {a} → NeedsHash a → Maybe GovActionID
getHash {NoConfidence} h = just h
getHash {NewCommittee _ _} h = just h
getHash {NewConstitution _} h = just h
getHash {TriggerHF _} h = just h
getHash {ChangePParams _} h = just h
getHash {TreasuryWdrl _} _ = nothing
getHash {Info} _ = nothing

open EnactState

getHashES : EnactState → GovAction → Maybe GovActionID
getHashES es NoConfidence = just $ es .cc .proj₂
getHashES es (NewCommittee _ _ ) = just $ es .cc .proj₂
getHashES es (NewConstitution _ _) = just $ es .constitution .proj₂
getHashES es (TriggerHF _) = just $ es .pv .proj₂
getHashES es (ChangePParams _ ) = just $ es .pparams .proj₂
getHashES es (TreasuryWdrl _) = nothing
getHashES es Info = nothing
```

Figure 38: Types and function used for the ENACT transition system
Figure 39 defines the rules of the ENACT transition system. Usually no preconditions are checked, and the state is simply updated (including the GovActionID for the hash protection scheme, if required). The exceptions are NewCommittee and TreasuryWdrl:

- **NewCommittee** requires that maximum terms are respected, and
- **TreasuryWdrl** requires that the treasury is able to cover the sum of all withdrawals (old and new).

\[
\vdash \Rightarrow \langle \_, \text{ENACT} \rangle : \text{EnactEnv} \rightarrow \text{EnactState} \rightarrow \text{GovAction} \rightarrow \text{EnactState} \rightarrow \text{Set}
\]

\[
\text{Enact-NoConf}:
\]

\[
\begin{array}{c}
[ \text{gid}, \text{t}, \text{e}] \vdash s \rightarrow \langle \text{NoConfidence}, \text{ENACT} \rangle \text{ record } s \{ \text{cc} = \text{nothing}, \text{gid} \}
\end{array}
\]

\[
\text{Enact-NewComm}:
\]

\[
\begin{array}{c}
\text{let old} = \text{maybe proj}_1 \varnothing (s .\text{EnactState.cc}.\text{proj}_1)
\text{maxTerm} = s .\text{pparams}.\text{proj}_1.\text{PParams.ccMaxTermLength} + e
\in
\forall [\text{term} \in \text{range}\ \text{new}]\ \text{term} \leq \text{maxTerm}
\end{array}
\]

\[
[ \text{gid}, \text{t}, \text{e}] \vdash s \rightarrow \langle \text{NewCommittee}\ \text{new}\ \text{rem}\ q, \text{ENACT} \rangle
\text{ record } s \{ \text{cc} = \text{just ((new} \cup \text{old}) \mid \text{rem} ^ c, q), \text{gid} \}
\]

\[
\text{Enact-NewConst}:
\]

\[
[ \text{gid}, \text{t}, \text{e}] \vdash s \rightarrow \langle \text{NewConstitution}\ \text{dh}\ \text{sh}, \text{ENACT} \rangle
\text{ record } s \{ \text{constitution} = (\text{dh}, \text{sh}), \text{gid} \}
\]

\[
\text{Enact-HF}:
\]

\[
[ \text{gid}, \text{t}, \text{e}] \vdash s \rightarrow \langle \text{TriggerHF}\ v, \text{ENACT} \rangle \text{ record } s \{ \text{pv} = v, \text{gid} \}
\]

\[
\text{Enact-PPParams}:
\]

\[
[ \text{gid}, \text{t}, \text{e}] \vdash s \rightarrow \langle \text{ChangePParams}\ \text{up}, \text{ENACT} \rangle
\text{ record } s \{ \text{pparams} = \text{applyUpdate} (s .\text{pparams}.\text{proj}_1) \text{up}, \text{gid} \}
\]

\[
\text{Enact-Wdrl}:
\]

\[
\begin{array}{c}
\text{let newWdrls} = s .\text{withdrawals} \cup^+ \text{wdrl}
\sum [x \leftrightarrow \text{newWdrls}] x \leq t
\end{array}
\]

\[
[ \text{gid}, \text{t}, \text{e}] \vdash s \rightarrow \langle \text{TreasuryWdrl}\ \text{wdrl}, \text{ENACT} \rangle \text{ record } s \{ \text{withdrawals} = \text{newWdrls} \}
\]

\[
\text{Enact-Info}:
\]

\[
[ \text{gid}, \text{t}, \text{e}] \vdash s \rightarrow \langle \text{Info}, \text{ENACT} \rangle s
\]

\textbf{Figure 39: ENACT transition system}
16 Ratification

Governance actions are ratified through on-chain votes. Different kinds of governance actions have different ratification requirements but always involve at least two of the three governance bodies.

A successful motion of no-confidence, election of a new constitutional committee, a constitutional change, or a hard-fork delays ratification of all other governance actions until the first epoch after their enactment. This gives a new constitutional committee enough time to vote on current proposals, re-evaluate existing proposals with respect to a new constitution, and ensures that the in principle arbitrary semantic changes caused by enacting a hard-fork do not have unintended consequences in combination with other actions.

16.1 Ratification requirements

Figure 40 details the ratification requirements for each governance action scenario. For a governance action to be ratified, all of these requirements must be satisfied, on top of other conditions that are explained further down. The threshold function is defined as a table, with a row for each type of GovAction and the columns representing the CC, DRep and SPO roles in that order.

The symbols mean the following:

- vote $x$: To pass the action, the yes votes need to be over the threshold $x$.
- $\_\_\_\_\_\_\_\_\$: The body of governance does not participate in voting.
- $\checkmark$: The constitutional committee needs to approve an action, with the threshold assigned to it.
- $\checkmark^\dagger$: Voting is possible, but the action will never be enacted. This is equivalent to vote 2 (or any other number above 1).

Two rows in this table contain functions that compute the DRep and SPO thresholds simultaneously: the rows for NewCommittee and ChangePParams.

For NewCommittee, there can be different thresholds depending on whether the system is in a state of no-confidence or not. This information is provided via the ccThreshold argument: if the system is in a state of no-confidence, then ccThreshold is set to nothing.

In case of the ChangePParams action, the thresholds further depend on what groups that action is associated with. pparamThreshold associates a pair of thresholds to each individual group. Since an individual update can contain multiple groups, the actual thresholds are then given by taking the maximum of all those thresholds.

Note that each protocol parameter belongs to exactly one of the four groups that have a DRep threshold, so a DRep vote will always be required. A protocol parameter may or may not be in the SecurityGroup, so an SPO vote may not be required.

Each of the $P_x$ and $Q_x$ are protocol parameters.

16.2 Protocol parameters and governance actions

Voting thresholds for protocol parameters can be set by group, and we do not require that each protocol parameter governance action be confined to a single group. In case a governance action carries updates for multiple parameters from different groups, the maximum threshold of all the groups involved will apply to any given such governance action.

The purpose of the SecurityGroup is to add an additional check to security-relevant protocol parameters. Any proposal that includes a change to a security-relevant protocol parameter must also be accepted by at least half of the SPO stake.
16.3 Ratification restrictions

As mentioned earlier, most governance actions must include a GovActionID for the most recently enacted action of its given type. Consequently, two actions of the same type can be enacted at the same time, but they must be deliberately designed to do so.

Figure 41 defines some types and functions used in the RATIFY transition system. CCData is simply an alias to define some functions more easily.

Figure 42 defines the actualVotes function. Given the current state about votes and other parts of the system it calculates a new mapping of votes, which is the mapping that will actually be used during ratification. Things such as default votes or resignation/expiry are implemented in this way.

actualVotes is defined as the union of four voting maps, corresponding to the constitutional committee, predefined (or auto) DReps, regular DReps and SPOs.

- roleVotes filters the votes based on the given governance role and is a helper for definitions further down.
- if a CC member has not yet registered a hot key, has expired, or has resigned, then actualCCVote returns abstain; if none of these conditions is met, then
  - if the CC member has voted, then that vote is returned;
Figure 41: Types and functions for the RATIFY transition system

- if the CC member has not voted, then the default value of no is returned.
- actualDRepVotes adds a default vote of no to all active DReps that didn’t vote.
- actualSPOVotes adds a default vote to all SPOs who didn’t vote, with the default depending on the action.

Figure 43 defines the accepted and expired functions (together with some helpers) that are used in the rules of RATIFY.

- getStakeDist computes the stake distribution based on the given governance role and the corresponding delegations. Note that every constitutional committee member has a stake of 1, giving them equal voting power. However, just as with other delegation, multiple CC members can delegate to the same hot key, giving that hot key the power of those multiple votes with a single actual vote.
- acceptedStakeRatio is the ratio of accepted stake. It is computed as the ratio of yes votes over the votes that didn’t abstain. The latter is equivalent to the sum of yes and no votes. The special division symbol /₀ indicates that in case of a division by 0, the numbers 0 should be returned. This implies that in the absence of stake, an action can only pass if the threshold is also set to 0.
Figure 42: Vote counting

- `acceptedBy` looks up the threshold in the `threshold` table and compares it to the result of `acceptedStakeRatio`.  

```haskell
collectVotes = RatifyEnv \rightarrow PParams \rightarrow CCData \rightarrow GovAction 
\rightarrow (GovRole \times Credential \rightarrow Vote) \rightarrow (VDeleg \rightarrow Vote)
collectVotes \Gamma pparams cc ga votes = mapKeys (credVoter CC) actualCCVotes \uplus actualPDRepVotes ga
actualCCVotes u1 actualDRepVotes \uplus actualSPOVotes ga

where

collectVotes = RatifyEnv \rightarrow PParams \rightarrow CCData \rightarrow GovAction 
\rightarrow (GovRole \times Credential \rightarrow Vote) \rightarrow (VDeleg \rightarrow Vote)
collectVotes \Gamma pparams cc ga votes = mapKeys (credVoter CC) actualCCVotes \uplus actualPDRepVotes ga
actualCCVotes u1 actualDRepVotes \uplus actualSPOVotes ga

where

roleVotes : GovRole \rightarrow VDeleg \rightarrow Vote
roleVotes r = mapKeys (uncurry credVoter) (filter (\lambda (x , _ ) \rightarrow r \equiv proj1 x) votes)

activeDReps = dom (filter (\lambda (_, e) \rightarrow currentEpoch \leq e) dreps)
spos = filter^* IsSPO (dom (stakeDistr stakeDistrs))

collectHotCred : Credential \times Epoch \rightarrow Maybe Credential
collectHotCred (c , e) = case \& currentEpoch \leq e \& lookup^*? ccHotKeys c of
  (true , just (just c')) \rightarrow just c'
_ \rightarrow nothing -- expired, no hot key or resigned

actualCCVote : Credential \rightarrow Epoch \rightarrow Vote
actualCCVote c e = case collectHotCred (c , e) of
  (just c') \rightarrow maybe id Vote.no (lookup^*? votes (CC , c'))
_ \rightarrow Vote.abstain

activeCC : (Credential \times Epoch) \rightarrow P Credential
activeCC m = mapPartial collectHotCred (m ^)

actualCCVotes = case cc of
  nothing \rightarrow \emptyset
  (just (m , q)) \rightarrow if ccMinSize \leq length^* (activeCC m)
  then mapWithKey actualCCVote m
  else constMap (dom m) Vote.no

actualPDRepVotes = GovAction \rightarrow VDeleg \rightarrow Vote
actualPDRepVotes NoConfidence
  = { abstainRep , Vote.abstain } \uplus { noConfidenceRep , Vote.yes }
actualPDRepVotes _ = { abstainRep , Vote.abstain } \uplus { noConfidenceRep , Vote.no }

actualDRepVotes = VDeleg \rightarrow Vote
actualDRepVotes = roleVotes DRep
  \uplus constMap (map^* (credVoter DRep) activeDReps) Vote.no

actualSPOVotes = GovAction \rightarrow VDeleg \rightarrow Vote
actualSPOVotes (TriggerHF _) = roleVotes SPO \uplus constMap spos Vote.no
actualSPOVotes _ = roleVotes SPO \uplus constMap spos Vote.abstain
```

getStakeDist : GovRole → P VDeleg → StakeDistrs → VDeleg → Coin
getStakeDist CC cc sd = constMap (filter⁺ IsCC cc) 1
getStakeDist DRep sd = filterKeys IsDRep (sd .stakeDistr)
getStakeDist SPO sd = filterKeys IsSPO (sd .stakeDistr)

acceptedStakeRatio : GovRole → P VDeleg → StakeDistrs → (VDeleg → Vote) → Ω
acceptedStakeRatio r cc dists votes = acceptedStake /₀ totalStake

where
acceptedStake totalStake : Coin
  acceptedStake = ∑[ x ← getStakeDist r cc dists | votes⁻¹ Vote.yes ] x
  totalStake = ∑[ x ← getStakeDist r cc dists | votes⁻¹ Vote.abstain ] x

acceptedBy : RatifyEnv → EnactState → GovActionState → GovRole → Set
acceptedBy Γ (record { cc = cc , _; pparams = pparams , _ }) gs role =
  let open GovActionState gs
      votes' = actualVotes Γ pparams cc action votes
      t = maybe id Ω (threshold pparams (proj₂ <$> cc) action role)
in acceptedStakeRatio role (dom votes') (stakeDistrs Γ) votes' ≥ t

accepted : RatifyEnv → EnactState → GovActionState → Set
accepted Γ es gs = acceptedBy Γ es gs CC ∧ acceptedBy Γ es gs DRep ∧ acceptedBy Γ es gs SPO

expired : Epoch → GovActionState → Set
expired current record { expiresIn = expiresIn } = expiresIn < current

Figure 43: Functions used in RATIFY rules, without delay

- **accepted** then checks if an action is accepted by all roles; and
- **expired** checks whether a governance action is expired in a given epoch.

Figure 44 defines functions that deal with delays. A given action can either be delayed if
the action contained in EnactState isn’t the one the given action is building on top of, which is
checked by verifyPrev, or if a previous action was a delayingAction. Note that delayingAction
affects the future: whenever a delayingAction is accepted all future actions are delayed. delayed
then expresses the condition whether an action is delayed. This happens either because the
previous action doesn’t match the current one, or because the previous action was a delaying
one. This information is passed in as an argument.

The RATIFY transition system is defined as the reflexive-transitive closure of RATIFY’,
which is defined via three rules, defined in Figure 45.

- **RATIFY-Accept** checks if the votes for a given GovAction meet the threshold required for
  acceptance, that the action is accepted and not delayed, and RATIFY-Accept ratifies the
  action.
- **RATIFY-Reject** asserts that the given GovAction is not accepted and expired; it removes
  the governance action.
- **RATIFY-Continue** covers the remaining cases and keeps the GovAction around for further
  voting.
verifyPrev : (a : GovAction) → NeedsHash a → EnactState → Set
verifyPrev NoConfidence h es = h ≡ es . cc . proj₂
verifyPrev (NewCommittee _) h es = h ≡ es . cc . proj₂
verifyPrev (NewConstitution _) h es = h ≡ es . constitution . proj₂
verifyPrev (TriggerHF _) h es = h ≡ es . pv . proj₂
verifyPrev (ChangePParams _) h es = h ≡ es . pparams . proj₂
verifyPrev (TreasuryWdrl _) _ _ = ⊤
verifyPrev Info _ _ = ⊤

delayingAction : GovAction → Bool
delayingAction NoConfidence = true
delayingAction (NewCommittee _) = true
delayingAction (NewConstitution _) = true
delayingAction (TriggerHF _) = true
delayingAction (ChangePParams _) = false
delayingAction (TreasuryWdrl _) = false
delayingAction Info = false

delayed : (a : GovAction) → NeedsHash a → EnactState → Bool → Set
delayed a h es d = ¬ verifyPrev a h es ∨ d = true

Figure 44: Functions relating to delays

Note that all governance actions eventually either get accepted and enacted via RATIFY-Accept or rejected via RATIFY-Reject. If an action satisfies all criteria to be accepted but cannot be enacted anyway, it is kept around and tried again at the next epoch boundary.

We never remove actions that do not attract sufficient yes votes before they expire, even if it is clear to an outside observer that this action will never be enacted. Such an action will simply keep getting checked every epoch until it expires.
\textbf{RATIFY-Accept}: let open RatifyEnv \( \Gamma \); \( st = a \cdot \text{proj}_2 \); open GovActionState \( st \) in accepted \( \Gamma \) \( es \)\( st \) 
\( \rightarrow \neg \) delayed action \( \text{prevAction} \) \( es \)\( d \) 
\( \rightarrow \) \( [ a \cdot \text{proj}_1, \text{treasury}, \text{currentEpoch} ]^e \) \( \vdash es \rightarrow \emptyset \) \( \text{action} \cdot \text{ENACT} \)\( es' \)
\( \Gamma \vdash [ es, \text{removed}, d ]^2 \rightarrow [ a \cdot \text{RATIFY}', es, \{ a \} \cup \text{removed}, \text{delayingAction} \text{action} ]^2 \)

\textbf{RATIFY-Reject}: let open RatifyEnv \( \Gamma \); \( st = a \cdot \text{proj}_2 \) in 
\( \neg \) accepted \( \Gamma \) \( es \)\( st \) 
\( \rightarrow \) expired currentEpoch \( st \)
\( \Gamma \vdash [ es, \text{removed}, d ]^2 \rightarrow [ a \cdot \text{RATIFY}', es, \{ a \} \cup \text{removed}, d ]^2 \)

\textbf{RATIFY-Continue}: let open RatifyEnv \( \Gamma \); \( st = a \cdot \text{proj}_2 \); open GovActionState \( st \) in 
\( \neg \) accepted \( \Gamma \) \( es \)\( st \) \( \times \) \( \neg \) expired currentEpoch \( st \) 
\( \forall \) accepted \( \Gamma \) \( es \)\( st \) 
\( \times \) ( delayed action \( \text{prevAction} \) \( es \)\( d \) 
\( \forall \) (\( \forall es' \rightarrow [ a \cdot \text{proj}_1, \text{treasury}, \text{currentEpoch} ]^e \) \( \vdash es \rightarrow \emptyset \) \( \text{action} \cdot \text{ENACT} \)\( es' \))
\( \Gamma \vdash [ es, \text{removed}, d ]^2 \rightarrow [ a \cdot \text{RATIFY}', es, \text{removed}, d ]^2 \)

\( \vdash \text{RATIFY}_- : \text{RatifyEnv} \rightarrow \text{RatifyState} \rightarrow \text{List} (\text{GovActionID} \times \text{GovActionState}) \rightarrow \text{RatifyState} \rightarrow \text{Set} \)
\( \vdash \text{RATIFY}_- = \text{ReflexiveTransitiveClosure } \vdash \_ \rightarrow \_ \text{RATIFY'} \_ \)

\textbf{Figure 45}: The RATIFY transition system
17 Epoch boundary

```haskell
record EpochState : Set where
  constructor [_,_,_,_]ₑ' 
  field acnt : Acnt
    ls : LState
    es : EnactState
    fut : RatifyState

record NewEpochEnv : Set where
  field stakeDistrs : StakeDistrs
    -- TODO: compute this from LState instead

record NewEpochState : Set where
  constructor [_,_]ⁿₑ
  field lastEpoch : Epoch
  epochState : EpochState
```

Figure 46: Definitions for the EPOCH and NEWEPOCH transition systems

Figure 47 defines the rule for the EPOCH transition system. Currently, this contains some logic that is handled by POOLREAP in the Shelley specification, since POOLREAP is not implemented here.

The EPOCH rule now also needs to invoke RATIFY and properly deal with its results, i.e:

- Pay out all the enacted treasury withdrawals.
- Remove expired and enacted governance actions & refund deposits.
- If \( \text{govSt}' \) is empty, increment the activity counter for DReps.
- Remove all hot keys from the constitutional committee delegation map that do not belong to currently elected members.
- Apply the resulting enact state from the previous epoch boundary \( \text{fut} \) and store the resulting enact state \( \text{fut}' \).
EPOCH: let

open EpochState eps hiding (es)
open RatifyState fut using (removed) renaming (es to esW)
-- ^ this rolls over the future enact state into es
open LState ls; open UTxOState utxoSt; open CertState certState
open PState pState; open DState dState; open GState gState
open Acnt acnt

trWithdrawals = esW .EnactState.withdrawals
totWithdrawals = ∑[ x ← trWithdrawals ] x

removedGovActions = flip concatMapˢ removed λ (gaid , gaSt) →
    mapˢ (GovActionState.returnAddr gaSt ,_)
    ((deposits | { GovActionDeposit gaid } ) ^)
govActionReturns = aggregate₊ (mapˢ (λ (a , _ , d) → a , d) removedGovActions f*)

es = record esW { withdrawals = ∅ }
retired = retiring ¬¹ e
payout = govActionReturns U* trWithdrawals
refunds = pullbackMap payout (λ x → record { net = NetworkId ; stake = x }) (dom rewards)
unclaimed = getCoin payout - getCoin refunds

govSt' = filter (λ x → ¿ proj₁ x # mapˢ proj₁ removed ¿) govSt

certState' =
    [ record dState { rewards = rewards U* refunds }
    , [ pools | retired ⌈ , retiring | retired ⌈ ]ᵖ
    , [ if null govSt' then mapValues (1 +_) dreps else dreps 
    , ccHotKeys | ccCreds (es .EnactState.cc) ]ᵀ ]ₑˢ

utxoSt' = [ utxo , 0 , deposits | mapˢ (proj₁ o proj₂) removedGovActions c , 0 ]ᵤ

ls' = [ utxoSt' , govSt' , certState' ]₁

acnt' = record acnt
    { treasury = treasury + fees + unclaimed + donations - totWithdrawals }
in
record { currentEpoch = e ; treasury = treasury ; GState gState ; NewEpochEnv Γ }
    → [ es , φ , false ]ᵀ -{ govSt' , RATIFY } fut'

Γ ⊢ eps -{ e , EPOCH } [ acnt' , ls' , es , fut' ]*
\[\Gamma \vdash\langle \mathcal{E}, \text{NEWEPOCH} \rangle \]  

_\text{NEWEPOCH-New}_:
\begin{align*}
& e = \text{lastEpoch} + 1 \\
& \rightarrow \Gamma \vdash \epsilon_0 \triangleright \langle e, \text{EPOCH} \rangle \epsilon_0' \\
& \Gamma \vdash [\text{lastEpoch}, \epsilon_0]^{\mathcal{E}} \rightarrow \langle e, \text{NEWEPOCH} \rangle [e, \epsilon_0']^{\mathcal{E}}
\end{align*}

_\text{NEWEPOCH-Not-New}_:
\begin{align*}
& e \neq \text{lastEpoch} + 1 \\
& \Gamma \vdash [\text{lastEpoch}, \epsilon_0]^{\mathcal{E}} \rightarrow \langle e, \text{NEWEPOCH} \rangle [\text{lastEpoch}, \epsilon_0]^{\mathcal{E}}
\end{align*}

**Figure 48**: NEWEPOCH transition system
18 Blockchain layer

```haskell
record ChainState : Set where
  field newEpochState : NewEpochState

record Block : Set where
  field ts : List Tx
    slot : Slot
```

**Figure 49:** Definitions CHAIN transition system

```haskell
⊢_¬¬≪_,CHAIN≫_ : τ → ChainState → Block → ChainState → Set
```

**Figure 50:** Type of the CHAIN transition system

```haskell
CHAIN :
  let open ChainState s; open Block b; open NewEpochState newEpochState
   open EpochState epochState; open EnactState es
  in
    record { stakeDistrs = calculateStakeDistrs ls }
     → newEpochState ←₀ epoch slot ,NEWEPOCH nes
    → [ slot , constitution .proj₁ .proj₂ , pparams .proj₁ , es ]^{e
        → ls ←₀ ts ,LEDGERS ls'}
    _ ⊢ s ←₀ b ,CHAIN
    record s { newEpochState = record nes { epochState = record epochState { ls = ls' } } }
```

**Figure 51:** CHAIN transition system
19 Properties

19.1 UTxO

Here, we state the fact that the UTxO relation is computable.

![Diagram](image.png)

Figure 52: Computing the UTxO transition system

Property 19.1 (Preserve Balance)

For all $\Gamma \in UTxOEnv$, utxo, utxo' $\in UTxO$, fees, fees' $\in Coin$ and $tx \in Tx$, if

\[
    tx.body.txd \notin \text{map}^3 \text{proj}_1(\text{dom} \text{utxo})
\]

and

\[
    \Gamma \models [\text{utxo}, \text{fees}, \text{deposits}, \text{donations}]^\nu \rightarrow [\text{tx},UTXO]
\]

\[
    [\text{utxo}', \text{fees}', \text{deposits}', \text{donations}]^\nu
\]

then

\[
    \text{getCoin} [\text{utxo}, \text{fees}, \text{deposits}, \text{donations}]^\nu
\]

\[
    = \text{getCoin} [\text{utxo}', \text{fees}', \text{deposits}', \text{donations}]^\nu
\]

Property 19.2 (General Minimum Spending Condition)

References


A Appendix: Agda essentials

Here we describe some of the essential concepts and syntax of the Agda programming language
and proof assistant. The goal is to provide some background for readers who are not already
familiar with Agda, to help them understand the other sections of the specification.

A.1 Record types

A record is a product with named accessors for the individual fields. It provides a way to define
a type that groups together inhabitants of other types.

Example.

```agda
record Pair (A B : Set) : Set where
  constructor ⟨_,_⟩
  field
    fst : A
    snd : B
```

We can construct an element of the type `Pair ℕ ℕ` (i.e., a pair of natural numbers) as follows:

```agda
p23 : Pair ℕ ℕ
p23 = record { fst = 2; snd = 3 }
```

Since our definition of the `Pair` type provides an (optional) constructor `⟨_,_⟩`, we can have
defined `p23` as follows:

```agda
p23' : Pair ℕ ℕ
p23' = ⟨ 2 , 3 ⟩
```

Finally, we can “update” a record by deriving from it a new record whose fields may contain
new values. The syntax is best explained by way of example.

```agda
p24 : Pair ℕ ℕ
p24 = record p23 { snd = 4 }
```

This results a new record, `p24`, which denotes the pair `⟨ 2 , 4 ⟩`.


B Bootstrapping EnactState

To form an `EnactState`, some governance action IDs need to be provided. However, at the time
of the initial hard fork into Conway there are no such previous actions. There are effectively
two ways to solve this issue:

- populate those fields with IDs chosen in some manner (e.g. random, all zeros, etc.), or
- add a special value to the types to indicate this situation.

In the Haskell implementation the latter solution was chosen. This means that everything
that deals with `GovActionID` needs to be aware of this special case and handle it properly.

This specification could have mirrored this choice, but it is not necessary here: since it is
already necessary to assume the absence of hash-collisions (specifically first pre-image resistance)
for various properties, we could pick arbitrary initial values to mirror this situation. Then, since
`GovActionID` contains a hash, that arbitrary initial value behaves just like a special case.
C Bootstrapping the Governance System

As described in [2], the governance system needs to be bootstrapped. During the bootstrap period, the following changes will be made to the ledger described in this document:

- Transactions containing any proposal except \texttt{TriggerHF}, \texttt{ChangePParams} or \texttt{Info} will be rejected.
- Transactions containing a vote other than a \texttt{CC} vote, a \texttt{SPO} vote on a \texttt{TriggerHF} action or any vote on an \texttt{Info} action will be rejected.
- \(Q4, P5\) and \(Q5e\) are set to 0.

This allows for a governance mechanism similar to the old, Shelley-era governance during the bootstrap phase, where the constitutional committee is mostly in charge. These restrictions will be removed during a subsequent hard fork, once enough DRep stake is present in the system to properly govern and secure itself.