Contents

1 Introduction

Repository: https://github.com/input-output-hk/formal-ledger-specifications

This document describes the formalization of the Cardano ledger specification in the Agda programming language and proof assistant. The specification formalized here is that of the Conway era, described in detail in the Cardano Improvement Proposal (CIP) 1694, github.com/cardano-foundation/CIPs/CIP-1694.

1.1 Separation of concerns

The *Cardano Node* consists of three pieces:

- Networking layer, which deals with sending messages accross 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:

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

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

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

These are modeled as 4-ary relations between the environment Γ, 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.

1.2 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 is captured by the following record, which is parametrized over the step relation.

```
record Computational (_⊢_⇀⦇_,X⦈_ : C → S → Sig → S → Set) : Set where
  field
    compute      : C → S → Sig → Maybe S
    ≡-just⇔STS : compute Γ s b ≡ just s' ⇔ Γ ⊢ s ⇀⦇ b ,X⦈ s'
```

1.3 Sets & maps

The ledger heavily uses set theory. For various reasons it was necessary to implement our own set theory (there'll 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.

```
_⊆_ : {A : Set} →   A →   A → Set
X ⊆ Y = ∀ {x} → x   X → x   Y

_≡ᵉ_ : {A : Set} →   A →   A → Set
X ≡ᵉ Y = X ⊆ Y × Y ⊆ X

Rel : Set → Set → Set
Rel A B =   (A × B)

left-unique : {A B : Set} → Rel A B → Set
left-unique R = ∀ {a b b'} → (a , b)   R → (a , b')   R → b ≡ b'

_⇀_ : Set → Set → Set
A ⇀ B = Σ (Rel A B) left-unique
```

2 Cryptographic primitives

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

Types & functions

$$\mathsf{SKey}\ \mathsf{VKey}\ \mathsf{Sig}\ \mathsf{Ser} : \mathsf{Set}$$
$$\mathsf{isKeyPair} : \mathsf{SKey} \to \mathsf{VKey} \to \mathsf{Set}$$
$$\mathsf{isSigned} : \mathsf{VKey} \to \mathsf{Ser} \to \mathsf{Sig} \to \mathsf{Set}$$
$$\mathsf{sign} : \mathsf{SKey} \to \mathsf{Ser} \to \mathsf{Sig}$$

$$\mathsf{KeyPair} = \Sigma[\ sk \in \mathsf{SKey}\ ]\ \Sigma[\ vk \in \mathsf{VKey}\ ]\ \mathsf{isKeyPair}\ sk\ vk$$

Property of signatures

$$((sk\ ,\ vk\ ,\ \_) : \mathsf{KeyPair})\ (d : \mathsf{Ser})\ (\sigma : \mathsf{Sig}) \to \mathsf{sign}\ sk\ d \equiv \sigma \to \mathsf{isSigned}\ vk\ d\ \sigma$$

Figure 1: Definitions for the public key signature scheme

3 Base types

$$\begin{aligned}
\mathrm{Coin} &= \mathbb{N} \\
\mathrm{Slot} &= \mathbb{N} \\
\mathrm{Epoch} &= \mathbb{N}
\end{aligned}$$

Figure 2: Some basic types used in many places in the ledger

```
record TokenAlgebra : Set₁ where
  field Value-CommutativeMonoid : CommutativeMonoid 0ℓ 0ℓ

  MemoryEstimate : Set
  MemoryEstimate = ℕ


field coin                     : Value → Coin
      inject                   : Coin → Value
      policies                 : Value → ℙ PolicyId
      size                     : Value → MemoryEstimate
      _≤▯_                     : Value → Value → Set
      AssetName                : Set
      specialAsset             : AssetName
      property                 : coin ∘ inject ≗ id
      coinIsMonoidHomomorphism : IsMonoidHomomorphism coin


sumᵛ : List Value → Value
sumᵛ = foldr _+ᵛ_ (inject 0)
```

Figure 3: Token algebras, used for multi-assets

4 Token algebras

5 Addresses

We define credentials and various types of addresses here.

Abstract types

```
    Network
    KeyHash
    ScriptHash
```

Derived types

```
  Credential = KeyHash ⊎ ScriptHash

  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

  Addr = BaseAddr ⊎ BootstrapAddr

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

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

  VKeyAddr = VKeyBaseAddr ⊎ VKeyBootstrapAddr
  ScriptAddr = ScriptBaseAddr ⊎ ScriptBootstrapAddr
```

Helper functions

```
  payCred    : Addr → Credential
  netId      : Addr → Network
  isVKeyAddr : Addr → Set
  isVKeyAddr = isVKey ∘ payCred
```

Figure 4: Definitions used in Addresses

6 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


data 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 : ss' S.⊆ ss → All (evalTimelock khs I) ss' → evalTimelock khs I (RequireMOf (length ss') ss)
  evalSig : x ∈ khs → evalTimelock khs I (RequireSig x)
  evalTSt : proj₁ I ≡ just l → a ≤ l → evalTimelock khs I (RequireTimeStart a)
  evalTEx : proj₂ I ≡ just r → r ≤ a → evalTimelock khs I (RequireTimeStart a)
```

Figure 5: Timelock scripts and their evaluation

7 Governance actions

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

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

```
GovActionID : Set
GovActionID = TxId × ℕ

data GovRole : Set where
  CC   : GovRole
  DRep : GovRole
  SPO  : GovRole

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     : KeyHash ⇀ Epoch → ℙ KeyHash → ℚ → GovAction
  NewConstitution  : DocHash                          → GovAction
  TriggerHF        : ProtVer                          → GovAction
  ChangePParams    : UpdateT → PPHash                 → GovAction
  TreasuryWdrl     : (RwdAddr ⇀ Coin)                 → GovAction
  Info             : GovAction
```

Figure 6: Governance actions

Figure 6 defines several data types used to represent governance actions including:

- *identifier*—a pair consisting of a TxId (transaction ID) and a natural number;
- *role*—one of three available voter roles defined above (CC, DRep, SPO);
- *voter delegation type*—one of three ways to delegate votes: by credential, abstention, or no confidence (credVoter, abstainRep, or noConfidenceRep);
- *anchor*—a url and a document hash;
- *governance action*—one of seven possible actions (see Figure 7 for definitions).

[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.

Action	Description
NoConfidence	a motion to create a *state of no-confidence* in the current constitutional committee
NewCommittee	changes to the members of the constitutional committee and/or to its signature threshold and/or term limits
NewConstitution	a modification to the off-chain Constitution, recorded as an on-chain hash of the text document
TriggerHF[1]	triggers a non-backwards compatible upgrade of the network; requires a prior software upgrade
ChangePParams	a change to *one or more* updatable protocol parameters, excluding changes to major protocol versions ("hard forks")
TreasuryWdrl	movements from the treasury, sub-categorized into small, medium or large withdrawals (based on the amount of Lovelace to be withdrawn)
Info	an action that has no effect on-chain, other than an on-chain record

Figure 7: Types of governance actions

7.1 Voting and ratification

Every governance action must be ratified by at least two of these three bodies using their on-chain *votes*. The type of action and the state of the governance system determines which bodies must ratify it. Ratified actions are then *enacted* on-chain, following a set of rules (see Section 7.3 and Figure 10). Figure 8 defines types that are used in ratification (for verifyPrev) where we

```
NeedsHash : GovAction → Set
NeedsHash NoConfidence            = GovActionID
NeedsHash (NewCommittee _ _ _) = GovActionID
NeedsHash (NewConstitution _)   = GovActionID
NeedsHash (TriggerHF _)            = GovActionID
NeedsHash (ChangePParams _ _) = GovActionID
NeedsHash (TreasuryWdrl _)       = ⊤
NeedsHash Info                          = ⊤

HashProtected : Set → Set
HashProtected A = A × GovActionID
```

Figure 8: NeedsHash and HashProtected types

check that the stored hash matches the one attached to the action we want to ratify.

- *Ratification.* An action is said to be *ratified* when it gathers enough votes in its favor (according to the rules described in Section 13).
- *Expiration.* An action that doesn't collect sufficient 'yes' votes before its deadline is said to have *expired*.
- *Enactment.* An action that has been ratified is said to be *enacted* once it has been activated on the network.

See Section 13 for more on the ratification process.

The data type Vote represents the different voting options: 'yes', 'no', or 'abstain'. Each vote is recorded in a GovVote record along with the following data: a governance action ID, a

```
data Vote : Set where
  yes     : Vote
  no      : Vote
  abstain : Vote

record GovVote : Set where
  field gid        : GovActionID
        role       : GovRole
        credential : Credential
        vote       : Vote
        anchor     : Maybe Anchor

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

Figure 9: Governance action proposals and votes

role, a credential, and an anchor (of types GovActionID, GovRole, Credential, and Maybe Anchor, respectively).

A *governance action proposal* is recorded in a GovProposal record which includes fields for a return address, the proposed governance action, a hash of the previous governance action, and an anchor (see Figure 9).

To submit a governance action to the chain one must provide a deposit which will be returned when the action is finalized (whether it is *ratified* or has *expired*). The deposit amount will be added to the *deposit pot*, similar to stake key deposits. It will also be counted towards the stake of the reward address it will be paid back to, to not reduce the submitter's voting power to vote on their own (and competing) actions.

Remarks.

1. A motion of no-confidence is an extreme measure that enables Ada holders to revoke the power that has been granted to the current constitutional committee.

2. A *single* governance action might contain *multiple* protocol parameter updates. Many parameters are inter-connected and might require moving in lockstep.

7.2 Protocol parameters and governance actions

Recall from Section 8, parameters used in the Cardano ledger are grouped according to the general purpose that each parameter serves (see Figure 12). Specifically, we have a NetworkGroup, an EconomicGroup, a TechnicalGroup, and a GovernanceGroup. This allows voting/ratification thresholds to be set by group, though 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.

7.3 Enactment

Enactment of a governance action is carried out as an *enact transition* which requires an *enact environment* an *enact state* representing the existing state (prior to enactment), the voted on governance action (that achieved enough votes to enact), and the state that results from enacting the given governance action (see Figure 10).

A record of type EnactEnv represents the environment for enacting a governance action. A record of type EnactState represents the state for enacting a governance action. The latter contains fields for the constitutional committee, constitution, protocol version, protocol parameters, withdrawals from treasury, and treasury balance.

```
record EnactEnv : Set where
  constructor ⟦_⟧ᵉ
  field gid : GovActionID

record EnactState : Set where
  field cc           : HashProtected (Maybe (KeyHash ⇀ Epoch × ℚ))
        constitution : HashProtected DocHash
        pv           : HashProtected ProtVer
        pparams      : HashProtected PParams
        withdrawals  : RwdAddr ⇀ Coin
        treasury     : Coin
```

Figure 10: Enactment types

The relation `_⊢_⇀⦇_,ENACT⦈_` is the transition relation for enacting a governance action. It represents how the *EnactState* changes when a specific governance action is enacted (see Figure 11).

```
_⊢_⇀⦇_,ENACT⦈_ : EnactEnv → EnactState → GovAction → EnactState → Set where
Enact-NoConf    : ⟦ gid ⟧ᵉ ⊢ s ⇀⦇ NoConfidence ,ENACT⦈ record s { cc = nothing , gid }
Enact-NewComm : ⟦ gid ⟧ᵉ ⊢ s ⇀⦇ NewCommittee new rem q ,ENACT⦈ let
  old = maybe proj₁ ∅ᵐ (proj₁ (EnactState.cc s))
  in record s { cc = just ((new ∪ᵐˡ old) ∣ rem ᶜ , q) , gid }
Enact-NewConst  : ⟦ gid ⟧ᵉ ⊢ s ⇀⦇ NewConstitution dh ,ENACT⦈ record s { constitution = dh , gid }
Enact-HF        : ⟦ gid ⟧ᵉ ⊢ s ⇀⦇ TriggerHF v ,ENACT⦈ record s { pv = v , gid }
Enact-PParams   : ⟦ gid ⟧ᵉ ⊢ s ⇀⦇ ChangePParams up h ,ENACT⦈
  record s { pparams = applyUpdate (proj₁ (s .pparams)) up , gid }
Enact-Wdrl      :
  let newWdrls = Σᵐᵛ[ x ← wdrl ᶠᵐ ] x
  in newWdrls ≤ s .treasury
  ────────────────────────────────────────
  ⟦ gid ⟧ᵉ ⊢ s ⇀⦇ TreasuryWdrl wdrl ,ENACT⦈
    record s { withdrawals = s .withdrawals ∪⁺ wdrl
             ; treasury    = s .treasury    ∸ newWdrls }
Enact-Info      : ⟦ gid ⟧ᵉ ⊢ s ⇀⦇ Info ,ENACT⦈ s
```

Figure 11: ENACT transition system

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 ProtVer type represents the protocol version used in the Cardano ledger. It is a pair of natural numbers, the first of which represents the major version, the second represents the minor version.

The PParam type is a record containing 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.

The *Network*, *Economic*, and *Technical* parameter groups contain protocol parameters that were already introduced during the Shelley, Alonzo and Babbage eras. The new protocol parameters introduced in the Conway era (CIP-1694) belong to the *Governance* group. These new parameters are declared in Figure 12 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, and Q4;
- minCCSize: minimum constitutional committee size;
- ccTermLimit: maximum term limit (in epochs) of constitutional committee members;
- govExpiration: governance action expiration;
- govDeposit: governance action deposit;
- drepDeposit: DRep deposit amount;
- drepActivity: DRep activity period;
- minimumAVS: the minimum active voting threshold.

```
ProtVer : Set
ProtVer = ℕ × ℕ

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

data PParamGroup : Set where
  NetworkGroup EconomicGroup TechnicalGroup GovernanceGroup : PParamGroup

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 : ℚ

record PParams : Set where
  field
```

Network group

```
        maxBlockSize   : ℕ
        maxTxSize      : ℕ
        maxHeaderSize  : ℕ
        maxValSize     : ℕ
        pv             : ProtVer -- retired, keep for now
```

Economic group

```
        a              : ℕ
        b              : ℕ
        minUTxOValue   : Coin
        poolDeposit    : Coin
```

Technical group

```
        Emax             : Epoch
        collateralPercent : ℕ
```

Governance group

```
        drepThresholds  : DrepThresholds
        poolThresholds  : PoolThresholds
        minCCSize       : ℕ
        ccTermLimit     : ℕ
        govExpiration   : ℕ
        govDeposit      : Coin
        drepDeposit     : Coin
        drepActivity    : Epoch
        minimumAVS      : Coin

paramsWellFormed : PParams → Bool
paramsWellFormed pp = ⌊ ¬? (0 ∈? setFromList
  (maxBlockSize ∷ maxTxSize ∷ maxHeaderSize ∷ maxValSize ∷ minUTxOValue ∷ poolDeposit
  ∷ collateralPercent ∷ ccTermLimit ∷ govExpiration ∷ govDeposit ∷ drepDeposit ∷ [])) ⌋
  where open PParams pp
```

Figure 12: Protocol parameter declarations

9 Transactions

Transactions are defined in Figure 13. 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 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
    UTxO  = TxIn ⇀ TxOut
    Wdrl  = RwdAddr ⇀ Coin

    ProposedPPUpdates = KeyHash ⇀ PParamsUpdate
    Update            = ProposedPPUpdates × Epoch
```

Transaction types

```
    record TxBody : Set where
      field txins      : ℙ TxIn
            txouts     : Ix ⇀ TxOut
            txfee      : Coin
            mint       : Value
            txvldt     : Maybe Slot × Maybe Slot
            txcerts    : List DCert
            txwdrls    : Wdrl
            txvote     : List GovVote
            txprop     : List GovProposal
            txdonation : ℕ
            txup       : Maybe Update
            txADhash   : Maybe ADHash
            netwrk     : Maybe Network
            txsize     : ℕ
            txid       : TxId

    record TxWitnesses : Set where
      field vkSigs : VKey ⇀ Sig
            scripts : ℙ Script

    record Tx : Set where
      field body  : TxBody
            wits  : TxWitnesses
            txAD  : Maybe AuxiliaryData
```

Figure 13: Definitions used in the UTxO transition system

```
    getValue : TxOut → Value
    getValue (_ , v) = v

    txinsVKey : ℙ TxIn → UTxO → ℙ TxIn
    txinsVKey txins utxo = txins ∩ dom ((utxo ∣' to-sp (isVKeyAddr? ∘ proj₁)) ˢ)
```

10 UTxO

10.1 Accounting

Figure 14 defines functions needed for the UTxO transition system. Figure 15 defines the types needed for the UTxO transition system. The UTxO transition system is given in Figure 17.

- 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.

```
outs : TxBody → UTxO
outs tx = mapKeys (txid tx ,_) (txouts tx) λ where _ _ refl → refl

balance : UTxO → Value
balance utxo = Σᵐᵛ[ x ← utxo ᶠᵐ ] getValue x

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

minfee : PParams → TxBody → Coin
minfee pp tx = a * txsize tx + b
  where open PParams pp

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

certDeposit : PParams → DCert → Maybe (DepositPurpose × Coin)
certDeposit _   (delegate c _ _ v) = just (CredentialDeposit c , v)
certDeposit pp (regpool c _)       = just (PoolDeposit   c , PParams.poolDeposit pp)
certDeposit _   (regdrep c v _)     = just (DRepDeposit c , v)
certDeposit _   _                      = nothing

certDepositᵐ : PParams → DCert → DepositPurpose ⇀ Coin
certDepositᵐ pp cert = case certDeposit pp cert of λ where
  (just (p , v))   → ❴ p , v ❵ᵐ
  nothing           → ∅ᵐ

certRefund : DCert → Maybe DepositPurpose
certRefund (delegate c nothing nothing x) = just (CredentialDeposit c)
certRefund (deregdrep c)                          = just (DRepDeposit c)
certRefund _                                          = nothing

certRefundˢ : DCert → ℙ DepositPurpose
certRefundˢ = partialToSet certRefund

propDepositᵐ : PParams → GovActionID → GovProposal → DepositPurpose ⇀ Coin
propDepositᵐ pp gaid record { returnAddr = record { stake = c } }
  = ❴ GovActionDeposit gaid , PParams.govDeposit pp ❵ᵐ

-- this has to be a type definition for inference to work
data inInterval (slot : Slot) : (Maybe Slot × Maybe Slot) → Set where
  both  : ∀ {l r} → l ≤ˢ slot × slot ≤ˢ r → inInterval slot (just l , just r)
  lower : ∀ {l}   → l ≤ˢ slot                → inInterval slot (just l , nothing)
  upper : ∀ {r}   → slot ≤ˢ r                → inInterval slot (nothing , just r)
  none :                                          inInterval slot (nothing , nothing)
```

Figure 14: Functions used in UTxO rules

Derived types

```
Deposits = DepositPurpose ⇀ Coin
```

UTxO environment

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

UTxO states

```
record UTxOState : Set where
  constructor ⟦_,_,_,_⟧ᵘ
  field utxo      : UTxO
        fees      : Coin
        deposits  : Deposits
        donations : Coin
```

UTxO transitions

```
_⊢_⇀⦇_,UTXO⦈_ : UTxOEnv → UTxOState → TxBody → UTxOState → Set
```

Figure 15: UTxO transition-system types

```
updateCertDeposits : PParams → List DCert → DepositPurpose ⇀ Coin → DepositPurpose ⇀ Coin
updateCertDeposits _ []              deposits = deposits
updateCertDeposits pp (cert ∷ certs) deposits =
  ((updateCertDeposits pp certs deposits) ∪⁺ certDeposit$^m$ pp cert) | certRefund$^s$ cert $^c$

updateProposalDeposits : PParams → TxId → List GovProposal → DepositPurpose ⇀ Coin → DepositPurpos
updateProposalDeposits pp _ [] deposits = deposits
updateProposalDeposits pp txid (prop ∷ props) deposits =
  updateProposalDeposits pp txid props deposits ∪⁺ propDeposit$^m$ pp (txid , length props) prop

updateDeposits : PParams → TxBody → DepositPurpose ⇀ Coin → DepositPurpose ⇀ Coin
updateDeposits pp txb = updateCertDeposits pp (txcerts txb)
                      ∘ updateProposalDeposits pp (txid txb) (txprop txb)

depositsChange : PParams → TxBody → DepositPurpose ⇀ Coin → ℤ
depositsChange pp txb deposits = getCoin (updateDeposits pp txb deposits) ⊖ getCoin deposits

depositRefunds : PParams → UTxOState → TxBody → Coin
depositRefunds pp st txb = negPart $ depositsChange pp txb deposits
  where open UTxOState st

newDeposits : PParams → UTxOState → TxBody → Coin
newDeposits pp st txb = posPart $ depositsChange pp txb deposits
  where open UTxOState st

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

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

Figure 16: Functions used in UTxO rules, continued

```
UTXO-inductive :
  ∀ {Γ} {s} {tx}
  → let slot       = UTxOEnv.slot Γ
        pp         = UTxOEnv.pparams Γ
        utxo       = UTxOState.utxo s
        fees       = UTxOState.fees s
        deposits   = UTxOState.deposits s
        donations  = UTxOState.donations s
    in
  txins tx ≢ ∅                              → txins tx ⊆ dom (utxo ˢ)
  → inInterval slot (txvldt tx)             → minfee pp tx ≤ txfee tx
  → consumed pp s tx ≡ produced pp s tx → coin (mint tx) ≡ 0
  → txsize tx ≤ maxTxSize pp
  ────────────────────────────────────────
  Γ ⊢ s ⇀⦇ tx ,UTXO⦈ ⟦ (utxo ∣ txins tx ᶜ) ∪ᵐˡ outs tx
                     , fees + txfee tx
                     , updateDeposits pp tx deposits
                     , donations + txdonation tx
                     ⟧ᵘ
```

Figure 17: UTXO inference rules

10.2 Witnessing

```
getVKeys : ℙ Credential → ℙ KeyHash
getVKeys = mapPartial isInj₁

getScripts : ℙ Credential → ℙ ScriptHash
getScripts = mapPartial isInj₂

credsNeeded : UTxO → TxBody → ℙ Credential
credsNeeded utxo txb =
  map (payCred ∘ proj₁) ((utxo ˢ) ⟪$⟫ txins txb)
  ∪ map cwitness (setFromList $ txcerts txb)
  ∪ map GovVote.credential (setFromList $ txvote txb)

witsVKeyNeeded : UTxO → TxBody → ℙ KeyHash
witsVKeyNeeded utxo = getVKeys ∘ credsNeeded utxo

scriptsNeeded : UTxO → TxBody → ℙ ScriptHash
scriptsNeeded utxo = getScripts ∘ credsNeeded utxo

scriptsP1 : TxWitnesses → ℙ P1Script
scriptsP1 txw = mapPartial isInj₁ (scripts txw)
```

Figure 18: Functions used for witnessing

```
_⊢_⇀⦇_,UTXOW⦈_ : UTxOEnv → UTxOState → Tx → UTxOState → Set
```

Figure 19: UTxOW transition-system types

```
UTXOW-inductive :
  ∀ {Γ} {s} {tx} {s'}
  → let utxo = UTxOState.utxo s
        txb = body tx
        txw = wits tx
        witsKeyHashes = map hash (dom (vkSigs txw ˢ))
        witsScriptHashes = map hash (scripts txw)
    in
  ∀[ (vk , σ) ∈ vkSigs txw ˢ ] isSigned vk (txidBytes (txid txb)) σ
  → ∀[ s ∈ scriptsP1 txw ] validP1Script witsKeyHashes (txvldt txb) s
  → witsVKeyNeeded utxo txb ⊆ witsKeyHashes
  → scriptsNeeded utxo txb ≡ᵉ witsScriptHashes
  → txADhash txb ≡ M.map hash (txAD tx)
  → Γ ⊢ s ⇀⦇ txb ,UTXO⦈ s'
  ────────────────────────────────────────────────
  Γ ⊢ s ⇀⦇ tx ,UTXOW⦈ s'
```

Figure 20: UTXOW inference rules

11 Delegation

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

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

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

record CertEnv : Set where
  constructor ⟦_,_,_⟧ᶜ
  field epoch : Epoch
        pp    : PParams
        votes : List GovVote

GovCertEnv = CertEnv
DelegEnv   = PParams
PoolEnv    = PParams

record DState : Set where
  constructor ⟦_,_,_⟧ᵈ

  field voteDelegs  : Credential ⇀ VDeleg
  --     ^ stake credential to DRep credential
        stakeDelegs : Credential ⇀ Credential
  --     ^ stake credential to pool credential
        rewards     : RwdAddr ⇀ 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 : KeyHash ⇀ Maybe KeyHash -- TODO: maybe replace with credential

record CertState : Set where
  field dState : DState
        pState : PState
        gState : GState


requiredDeposit : {A : Set} → PParams → Maybe A → Coin
requiredDeposit pp (just _) = PParams.poolDeposit pp
requiredDeposit pp nothing = 0
```

```
data _⊢_⇀⦇_,DELEG⦈_ : DelegEnv → DState → DCert → DState → Set where
  DELEG-delegate :
    d ≡ requiredDeposit pp mv ⊔ requiredDeposit pp mc
    ────────────────────────────────
    pp ⊢ ⟦ vDelegs , sDelegs , rwds ⟧ᵈ ⇀⦇ delegate c mv mc d ,DELEG⦈
         ⟦ insertIfJust c mv vDelegs , insertIfJust c mc sDelegs , rwds ⟧ᵈ

data _⊢_⇀⦇_,POOL⦈_ : PoolEnv → PState → DCert → PState → Set where
  POOL-regpool : let open PParams pp ; open PoolParams poolParams in
    c ∉ dom (pools ˢ)
    ────────────────────────────────
    pp ⊢ ⟦ pools , retiring ⟧ᵖ ⇀⦇ regpool c poolParams ,POOL⦈ ⟦ ❴ c , poolParams ❵ᵐ ∪ᵐˡ pools , retiring

  POOL-retirepool : let open PoolParams poolParams in
    pp ⊢ ⟦ pools , retiring ⟧ᵖ ⇀⦇ retirepool c e ,POOL⦈
         ⟦ pools , ❴ c , e ❵ᵐ ∪ᵐˡ retiring ⟧ᵖ

data _⊢_⇀⦇_,GOVCERT⦈_ : GovCertEnv → GState → DCert → GState → Set where
  GOVCERT-regdrep : let open PParams pp in
    (d ≡ drepDeposit × c ∉ dom (dReps ˢ)) ⊎ (d ≡ 0 × c ∈ dom (dReps ˢ))
    ────────────────────────────────
    ⟦ e , pp , vs ⟧ᶜ ⊢ ⟦ dReps , ccKeys ⟧ᵛ ⇀⦇ regdrep c d an ,GOVCERT⦈
                       ⟦ ❴ c , e + drepActivity ❵ᵐ ∪ᵐˡ dReps , ccKeys ⟧ᵛ

  GOVCERT-deregdrep :
    c ∈ dom (dReps ˢ)
    ────────────────────────────────
    Γ ⊢ ⟦ dReps , ccKeys ⟧ᵛ ⇀⦇ deregdrep c ,GOVCERT⦈
        ⟦ dReps ∣ ❴ c ❵ ᶜ , ccKeys ⟧ᵛ

  GOVCERT-ccreghot :
    (kh , nothing) ∉ ccKeys ˢ
    ────────────────────────────────
    Γ ⊢ ⟦ dReps , ccKeys ⟧ᵛ ⇀⦇ ccreghot (inj₁ kh) mkh ,GOVCERT⦈
        ⟦ dReps , singletonᵐ kh mkh ∪ᵐˡ ccKeys ⟧ᵛ

data _⊢_⇀⦇_,CERT⦈_ : CertEnv → CertState → DCert → CertState → Set where
  CERT-deleg :
    pp ⊢ stᵈ ⇀⦇ dCert ,DELEG⦈ stᵈ '
    ────────────────────────────────
    ⟦ e , pp , vs ⟧ᶜ ⊢ st ⇀⦇ dCert ,CERT⦈ record st { dState = stᵈ ' }

  CERT-vdel :
    Γ ⊢ st ⇀⦇ dCert ,GOVCERT⦈ st '
    ────────────────────────────────
    Γ ⊢ st ⇀⦇ dCert ,CERT⦈ record st { gState = st ' }

  CERT-pool :
    pp ⊢ stᵖ ⇀⦇ dCert ,POOL⦈ stᵖ '
    ────────────────────────────────
    ⟦ e , pp , vs ⟧ᶜ ⊢ st ⇀⦇ dCert ,CERT⦈ record st { pState = stᵖ ' }

data _⊢_⇀⦇_,CERTBASE⦈_ : CertEnv → CertState → ⊤ → CertState → Set where
  CERT-base :
    let open PParams pp; open CertState st; open GState gState
        refresh = mapPartial (λ v → let open GovVote v in case role of λ where
          GovRole.DRep → just credential
            _ → nothing) (fromList vs)
```

12 Ledger State Transition

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 ⟦_,_⟧ˡᵉ
  field slot     : Slot
        pparams : PParams

record LState : Set where
  constructor ⟦_,_,_⟧ˡ
  field utxoSt    : UTxOState
        govSt     : GovState
        certState : CertState

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

Figure 23: Types and functions for the LEDGER transition system

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

Figure 24: The type of the LEDGER transition system

```
LEDGER : let open LState s; txb = body tx; open LEnv Γ in
  record { LEnv Γ } ⊢ utxoSt ⇀⦇ tx ,UTXOW⦈ utxoSt'
  → ⟦ epoch slot , pparams , txvote txb ⟧ᶜ ⊢ certState ⇀⦇ txcerts txb ,CERTS⦈ certState'
  → ⟦ txid txb , epoch slot , pparams ⟧▯ ⊢ govSt ⇀⦇ txgov txb ,GOV⦈ govSt'
  → map stake (dom (txwdrls txb ˢ)) ⊆ dom (voteDelegs (dState certState') ˢ)
  ────────────────────────────────────────────
  Γ ⊢ s ⇀⦇ tx ,LEDGER⦈ ⟦ utxoSt' , govSt' , certState' ⟧ˡ
```

Figure 25: LEDGER transition system

```
_⊢_⇀⦇_,LEDGERS⦈_ : LEnv → LState → List Tx → LState → Set
_⊢_⇀⦇_,LEDGERS⦈_ = SS⇒BS (λ where (Γ , _) → Γ ⊢_⇀⦇_,LEDGER⦈_)
```

Figure 26: LEDGERS transition system

13 Ratification

Governance actions are *ratified* through on-chain voting actions. Different kinds of governance actions have different ratification requirements but always involve *two of the three* governance bodies, with the exception of a hard-fork initiation, which requires ratification by all governance bodies. Depending on the type of governance action, an action will thus be ratified when a combination of the following occurs:

- the *constitutional committee* (CC) approves of the action; for this to occur, the number of CC members who vote yes must meet the CC vote threshold;
- the *delegation representatives* (DReps) approve of the action; for this to occur, the stake controlled by the DReps who vote yes must meet the DRep vote threshold as a percentage of the *total participating voting stake* (totalStake);
- the stake pool operators (SPOs) approve of the action; for this to occur, the stake controlled by the SPOs who vote yes must meet a certain threshold as a percentage of the *total registered voting stake* (totalStake).

Warning. Different stake distributions apply to DReps and SPOs.

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.

13.1 Ratification requirements

Figure 27 details the ratification requirements for each governance action scenario. The columns represent

- GovAction: the action under consideration;
- CC: a ✓ indicates that the constitutional committee must approve this action; a - symbol means that constitutional committee votes do not apply;
- DRep: the vote threshold that must be met as a percentage of totalStake;
- SPO: the vote threshold that must be met as a percentage of the stake held by all stake pools; a - symbol means that SPO votes do not apply.

Each of these thresholds is a governance parameter. The two thresholds for the Info action are set to 100% since setting it any lower would result in not being able to poll above the threshold.

13.2 Ratification restrictions

As mentioned earlier, each GovAction 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 28 defines three more types used in the ratification transition system.

- StakeDistrs represents a map relating each voting delegate to an amount of stake;
- RatifyEnv denotes an environment with data required for ratification;
- RatifyState denotes an enactment state that exists during ratification;

GovAction	CC	DRep	SPO
1. Motion of no-confidence	-	P1	Q1
2a. New committee/threshold (*normal state*)	-	P2a	Q2a
2b. New committee/threshold (*state of no-confidence*)	-	P2b	Q2b
3. Update to the Constitution	✓	P3	-
4. Hard-fork initiation	✓	P4	Q4
5a. Changes to protocol parameters in the NetworkGroup	✓	P5a	-
5b. Changes to protocol parameters in the EconomicGroup	✓	P5b	-
5c. Changes to protocol parameters in the TechnicalGroup	✓	P5c	-
5d. Changes to protocol parameters in the GovernanceGroup	✓	P5d	-
6. Treasury withdrawal	✓	P6	-
7. Info	✓	100	100

Figure 27: Retification requirements

```
record StakeDistrs : Set where
  field stakeDistr : VDeleg ⇀ Coin

record RatifyEnv : Set where
  field stakeDistrs   : StakeDistrs
        currentEpoch : Epoch
        dreps         : Credential ⇀ Epoch
        ccHotKeys     : KeyHash ⇀ Maybe KeyHash

record RatifyState : Set where
  constructor ⟦_,_,_,_⟧ʳ
  field es       : EnactState
        future   : List (GovActionID × GovActionState)
        removed  : List (GovActionID × GovActionState)
        delay    : Bool

CCData : Set
CCData = Maybe (KeyHash ⇀ Epoch × R.ℚ)
```

Figure 28: Types and functions for the RATIFY transition system

- CCData stores data about the constitutional committee.

The code in Figure 29 defines some of the types required for ratification of a governance action.

- Assuming a ratification environment Γ,
  - cc contains constitutional committee data;
  - votes is a relation associating each role-credential pair with the vote cast by the individual denoted by that pair;
  - ga denotes the governance action being voted upon.
- roleVotes filters the votes based on the given governance role.

```
-- Module Parameters:
(Γ     : RatifyEnv)                   -- ratification environment
(cc    : CCData)                      -- constitutional committee data
(votes : (GovRole × Credential) ⇀ Vote) -- the map relating delegates to their votes
(ga    : GovAction)                   -- the governance action that was voted on


roleVotes : GovRole → (GovRole × Credential) ⇀ Vote
roleVotes r = filterᵐ (to-sp ((r ≟_) ∘ proj₁ ∘ proj₁)) votes

actualCCVote : KeyHash → Epoch → Vote
actualCCVote kh e = case ⌊ currentEpoch ≤ᵉ? e ⌋ ,′ lookupᵐ? ccHotKeys kh ⦃ _ ∈? _ ⦄ of λ where
  (true , just (just hk)) → maybe′ id Vote.no $ lookupᵐ? votes (CC , (inj₁ hk)) ⦃ _ ∈? _ ⦄
  _ → Vote.abstain -- expired, no hot key or resigned

actualCCVotes : Credential ⇀ Vote
actualCCVotes = case cc of λ where
  (just (cc , _)) → mapKeys inj₁ (mapWithKey actualCCVote cc) (λ where _ _ refl → refl)
  nothing → ∅ᵐ

actualPDRepVotes : VDeleg ⇀ Vote
actualPDRepVotes = ❴ abstainRep , Vote.abstain ❵ᵐ
  ∪ᵐˡ ❴ noConfidenceRep , (case ga of λ where
                          NoConfidence → Vote.yes
                          _ → Vote.no) ❵ᵐ
actualDRepVotes : VDeleg ⇀ Vote
actualDRepVotes = mapKeys (uncurry credVoter) (roleVotes GovRole.DRep) (λ where _ _ refl → refl)
                ∪ᵐˡ constMap (map (credVoter DRep) activeDReps) Vote.no
  where
    activeDReps : ℙ Credential
    activeDReps = dom (filterᵐ (to-sp (currentEpoch ≤ᵉ?_ ∘ proj₂)) dreps ˢ)

actualVotes : VDeleg ⇀ Vote
actualVotes = mapKeys (credVoter CC) actualCCVotes (λ where _ _ refl → refl)
            ∪ᵐˡ actualPDRepVotes ∪ᵐˡ actualDRepVotes
            ∪ᵐˡ mapKeys (uncurry credVoter) votes (λ where _ _ refl → refl)
            -- TODO: make `actualVotes` for SPO
```

Figure 29: Types and proofs for the ratification of governance actions

- actualCCVote determines how the vote of each CC member will be counted; specifically, if a CC member has not yet registered a hot key, has expired, or has resigned, then actualCCVote returns abstain; if those none of these conditions is met, then
  - if the CC member has voted, then that vote is returned;
  - if the CC member has not voted, then the default value of no is returned.
- actualCCVotes uses actualCCVote to determine how the votes of all CC members will be counted.
- actualPDRepVotes determines how the votes will be counted for DReps; here, abstainRep is mapped to abstain and noConfidenceRep is mapped to either yes or no, depending on the value of ga.
- actualDRepVotes determines how the votes of DReps will be counted; activeDReps that didn't vote count as a no.

- actualVotes is a partial function relating delegates to the actual vote that will be counted on their behalf; it accomplishes this by aggregating the results of actualCCVotes, actualPDRepVotes, and actualDRepVotes.

```
votedHashes : Vote → (VDeleg ⇀ Vote) → GovRole → ℙ VDeleg
votedHashes v votes r = votes ⁻¹ v

votedYesHashes : (VDeleg ⇀ Vote) → GovRole → ℙ VDeleg
votedYesHashes = votedHashes Vote.yes

votedAbstainHashes : (VDeleg ⇀ Vote) → GovRole → ℙ VDeleg
votedAbstainHashes = votedHashes Vote.abstain

participatingHashes : (VDeleg ⇀ Vote) → GovRole → ℙ VDeleg
participatingHashes votes r = votedYesHashes votes r ∪ votedHashes Vote.no votes r

isCC : VDeleg → Bool
isCC (credVoter CC _) = true
isCC _                = false

isDRep : VDeleg → Bool
isDRep (credVoter DRep _) = true
isDRep (credVoter _ _)    = false
isDRep abstainRep         = true
isDRep noConfidenceRep    = true

isSPO : VDeleg → Bool
isSPO (credVoter SPO _) = true
isSPO _                 = false
```

Figure 30: Calculation of the votes as they will be counted

The code in Figure 30 defines votedHashes, which returns the set of delegates who voted a certain way on the given governance role, as well as isCC, isDRep, and isSPO, which return true if the given delegate is a CC member, a DRep, or an SPO (resp.) and false otherwise. The code in Figure 31 defines yet more types required for ratification of a governance action.

- getStakeDist computes the stake distribution based on the given governance role and the corresponding delegations;
- acceptedStake calculates the sum of stakes for all delegations that voted yes for the specified role;
- totalStake calculates the sum of stakes for all delegations that didn't vote abstain for the given role;
- activeVotingStake computes the total stake for the role of DRep for active voting; it calculates the sum of stakes for all active delegates that have not voted (i.e., their delegation is present in CC but not in the votes mapping);
- accepted checks if an action is accepted for the CC, DRep, and SPO roles and whether it meets the minimum active voting stake (meetsMinAVS);
- expired checks whether a governance action is expired in a given epoch.

The code in Figure 32 defines still more types required for ratification of a governance action.

```
getStakeDist : GovRole → ℙ VDeleg → StakeDistrs → VDeleg ⇀ Coin
getStakeDist CC    cc _                                 = constMap (filterˢ isCCProp cc) 1
getStakeDist DRep _  record { stakeDistr = dist } = filterᵐ (sp-∘ isDRepProp proj₁) dist
getStakeDist SPO   _  record { stakeDistr = dist } = filterᵐ (sp-∘ isSPOProp proj₁) dist

acceptedStake : GovRole → ℙ VDeleg → StakeDistrs → (VDeleg ⇀ Vote) → Coin
acceptedStake r cc dists votes =
  Σᵐᵛ[ x ← (getStakeDist r cc dists ∣ votedYesHashes votes r) ᶠᵐ ] x

totalStake : GovRole → ℙ VDeleg → StakeDistrs → (VDeleg ⇀ Vote) → Coin
totalStake r cc dists votes = Σᵐᵛ[ x ← getStakeDist r cc dists ∣ votedAbstainHashes votes r ᶜ ᶠᵐ ] x

activeVotingStake : ℙ VDeleg → StakeDistrs → (VDeleg ⇀ Vote) → Coin
activeVotingStake cc dists votes = Σᵐᵛ[ x ← getStakeDist DRep cc dists ∣ dom (votes ˢ) ᶜ ᶠᵐ ] x

-- For now, consider a proposal as accepted if the CC and half of the SPOs
-- and DReps agree.
accepted' : RatifyEnv → EnactState → GovActionState → Set
accepted' Γ es@record { cc = cc , _ ; pparams = pparams , _ }
            s@record { votes = votes' ; action = action } =
  acceptedBy CC ∧ acceptedBy DRep ∧ acceptedBy SPO ∧ meetsMinAVS
  where
    open RatifyEnv Γ
    open PParams pparams

    votes = actualVotes Γ cc votes' action
    cc' = dom (votes ˢ)
    redStakeDistr = restrictedDists coinThreshold rankThreshold stakeDistrs

    meetsMinAVS : Set
    meetsMinAVS = activeVotingStake cc' redStakeDistr votes ≥ minimumAVS

    acceptedBy : GovRole → Set
    acceptedBy role = let t = threshold pparams (Data.Maybe.map proj₂ cc) action role in
      case totalStake role cc' redStakeDistr votes of λ where
        0         → t ≡ R.0ℚ -- if there's no stake, accept only if threshold is zero
        x@(suc _) → Z.+ acceptedStake role cc' redStakeDistr votes R./ x R.≥ t

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

Figure 31: Calculation of stake distributions

- verifyPrev takes a governance action, its NeedsHash, and EnactState and checks whether the ratification restrictions are met;
- delayingAction takes a governance action and returns true if it is a “delaying action” (NoConfidence, NewCommittee, NewConstitution, TriggerHF) and returns false otherwise;
- delayed checks whether a given GovAction is delayed.

Figure 33 defines three rules, RATIFY-Accept, RATIFY-Reject, and RATIFY-Continue, along with the relation _⊢_⇀⦇_,RATIFY⦈_. The latter is the transition relation for ratification of a GovAction. The three rules are briefly described here, followed by more details about how they work.

```
verifyPrev : (a : GovAction) → NeedsHash a → EnactState → Set
verifyPrev NoConfidence           h es = let open EnactState es in h ≡ proj₂ cc
verifyPrev (NewCommittee _ _ _) h es = let open EnactState es in h ≡ proj₂ cc
verifyPrev (NewConstitution _)    h es = let open EnactState es in h ≡ proj₂ constitution
verifyPrev (TriggerHF _)          h es = let open EnactState es in h ≡ proj₂ pv
verifyPrev (ChangePParams _ _)    h es = let open EnactState es in h ≡ proj₂ pparams
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 32: Determination of the status of ratification of the governance action

```
RATIFY-Accept : let st = proj₂ a; open GovActionState st in
  accepted Γ es st
  → ¬ delayed action prevAction es d
  → ⟦ proj₁ a ⟧ᵉ ⊢ es ⇀⦇ action ,ENACT⦈ es'
  ────────────────────────────────
  Γ ⊢ ⟦ es , f , removed , d ⟧ʳ ⇀⦇ a ,RATIFY'⦈ ⟦ es' , f , a ∷ removed , delayingAction action ⟧ʳ

-- remove expired actions
-- NOTE:  We don't have to remove actions that can never be accpted because of
--        sufficient no votes.

RATIFY-Reject : let open RatifyEnv Γ; st = proj₂ a in
  ¬ accepted Γ es st
  → expired currentEpoch st
  ────────────────────────────────
  Γ ⊢ ⟦ es , f , removed , d ⟧ʳ ⇀⦇ a ,RATIFY'⦈ ⟦ es , f , a ∷ removed , d ⟧ʳ

-- Continue voting in the next epoch

RATIFY-Continue : let open RatifyEnv Γ; st = proj₂ a; open GovActionState st in
  ¬ accepted Γ es st × ¬ expired currentEpoch st ⊎ delayed action prevAction es d
  ────────────────────────────────
  Γ ⊢ ⟦ es , f , removed , d ⟧ʳ ⇀⦇ a ,RATIFY'⦈ ⟦ es , a ∷ f , removed , d ⟧ʳ

_⊢_⇀⦇_,RATIFY⦈_ : RatifyEnv → RatifyState → List (GovActionID × GovActionState)
                   → RatifyState → Set
_⊢_⇀⦇_,RATIFY⦈_ = SS⇒BS (λ where (Γ , _) → Γ ⊢_⇀⦇_,RATIFY'⦈_)
```

Figure 33: The RATIFY transition system

- RATIFY-Accept asserts that the votes for a given GovAction meets the threshold required for acceptance; 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.

14 Blockchain layer

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

record NewEpochState : Set where
  constructor ⟦_,_,_,_,_⟧ⁿᵉ
  field lastEpoch : Epoch
        acnt      : Acnt
        ls        : LState
        es esFut  : EnactState

record ChainState : Set where
  field newEpochState : NewEpochState

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

instance
  _ = +-0-monoid
```

Figure 34: Definitions for the NEWEPOCH and CHAIN transition systems

```
NEWEPOCH-New : ∀ {Γ} → let
  open NewEpochState nes hiding (es) renaming (esFut to es) -- this rolls over esFut into es
  open LState ls
  -- TODO Wire CertState together with treasury and withdrawals
  open CertState certState
  open PState pState
  removedGovActions = map GovActionDeposit (map proj₁ (fromList removed))
  pup = PPUpdateState.pup ppup
  deposits = UTxOState.deposits utxoSt
  retired = retiring ⁻¹ e
  govActionReturns' = concatMapˢ
       (λ where
         (gaid , gaSt) → map
           (GovActionState.returnAddr gaSt ,_)
           ((deposits ˢ) ⟪$⟫ { GovActionDeposit gaid })
       )
       (fromList removed)
  govActionReturns = aggregate▯ (govActionReturns' , finiteness govActionReturns')
  rewards = DState.rewards dState
  refunds    = govActionReturns ∣ dom (rewards ˢ)
  unclaimed = govActionReturns ∣ dom (rewards ˢ) ᶜ
  certState' = record certState {
       pState = record pState { pools = pools ∣ retired ᶜ ; retiring = retiring ∣ retired ᶜ };
       dState = record dState { rewards = rewards ∪⁺ refunds } }
  utxoSt' = record utxoSt
       { fees = 0
       ; deposits = deposits ∣ removedGovActions ᶜ
       }
  ls' = record ls { govSt = govSt' ; utxoSt = utxoSt' ; certState = certState' }
  acnt' = record acnt { treasury = Acnt.treasury acnt + UTxOState.fees utxoSt + getCoin unclaimed }
  in
  e ≡ sucᵉ lastEpoch
  → record { currentEpoch = e ; GState gState ; NewEpochEnv Γ }
       ⊢ ⟦ es , [] , [] , false ⟧ʳ ⇀⦇ govSt ,RATIFY⦈ ⟦ es' , govSt' , removed , d ⟧ʳ
  -- TODO: remove keys that aren't in the CC from the hot key map
  ────────────────────────────────
  Γ ⊢ nes ⇀⦇ e ,NEWEPOCH⦈ ⟦ e , acnt' , ls' , es , es' ⟧ⁿᵉ

NEWEPOCH-Not-New : ∀ {Γ} → let open NewEpochState nes in
  e ≢ sucᵉ lastEpoch
  ────────────────────────────────
  Γ ⊢ nes ⇀⦇ e ,NEWEPOCH⦈ nes
```

Figure 35: NEWEPOCH transition system

```
_⊢_⇀⦇_,CHAIN⦈_ : ⊤ → ChainState → Block → ChainState → Set
```

Figure 36: Type of the CHAIN transition system

```
CHAIN :
  let open ChainState s; open Block b; open NewEpochState
      stakeDistrs = calculateStakeDistrs (ls nes)
  in
  record { stakeDistrs = stakeDistrs } ⊢ newEpochState ⇀⦇ epoch slot ,NEWEPOCH⦈ nes
  → ⟦ slot , proj₁ (EnactState.pparams (es nes)) ⟧ˡᵉ ⊢ ls nes ⇀⦇ ts ,LEDGERS⦈ ls'
  ────────────────────────────────
  _ ⊢ s ⇀⦇ b ,CHAIN⦈ record s { newEpochState = record nes { ls = ls' } }
```

Figure 37: CHAIN transition system

15 Properties

15.1 UTxO

Here, we state the fact that the UTxO relation is computable. This just follows from our automation.

```
UTXO-step : UTxOEnv → UTxOState → TxBody → Maybe UTxOState
UTXO-step = compute Computational-UTXO

UTXO-step-computes-UTXO :
  UTXO-step Γ utxoState tx ≡ just utxoState' ⇔ Γ ⊢ utxoState ⇀⦇ tx ,UTXO⦈ utxoState'
UTXO-step-computes-UTXO = ≡-just⇔STS Computational-UTXO
```

Figure 38: Computing the UTXO transition system

Property 15.1 (Preserve Balance) *For all* $env \in$ UTxOEnv, $utxo, utxo' \in$ UTxO, $fees, fees' \in$ Coin *and* $tx \in$ TxBody, *if* txid *tx* ∉ map proj₁ (dom (*utxo* ˢ)) *and* Γ ⊢ ⟦ *utxo* , *fees* , *deposits* , *donations* ⟧ᵘ ⇀⦇ *tx* ,UTXO⦈ ⟦ *utxo'* , *fees'* , *deposits'* , *donations'* ⟧ᵘ *then*

getCoin ⟦ *utxo* , *fees* , *deposits* , *donations* ⟧ᵘ ≡ getCoin ⟦ *utxo'* , *fees'* , *deposits'* , *donations'* ⟧ᵘ