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 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:

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

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

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

These are modelled 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 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.

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

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

Rel : Set → Set → Set
Rel A B = ℙ (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

```
SKey VKey Sig Ser : Set
isKeyPair : SKey → VKey → Set
isSigned : VKey → Ser → Sig → Set
sign : SKey → Ser → Sig

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 1: Definitions for the public key signature scheme

3 Base types

```
Coin  = ℕ
Slot  = ℕ
Epoch = ℕ
```

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
      _≤ᵗ_                      : 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
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       ⊎ BootstrapAddr
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


module _ (khs : ℙ KeyHash) (I : Maybe Slot × Maybe Slot) where
  data evalTimelock : Timelock → Set where
    evalAll : All evalTimelock ss
            → evalTimelock (RequireAllOf ss)
    evalAny : Any evalTimelock ss
            → evalTimelock (RequireAnyOf ss)
    evalMOf : ss' S.⊆ ss → All evalTimelock ss'
            → evalTimelock (RequireMOf (length ss') ss)
    evalSig : x ∈ khs
            → evalTimelock (RequireSig x)
    evalTSt : I .proj₁ ≡ just l → a ≤ l
            → evalTimelock (RequireTimeStart a)
    evalTEx : I .proj₂ ≡ just r → r ≤ a
            → evalTimelock (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 DRep 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     : 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 → Bool
actionWellFormed (ChangePParams x) = ppdWellFormed x
actionWellFormed _ = true
```

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 11). Figure 9 defines types that are used in ratification (for verifyPrev) where we 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 14).
- *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 14 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 role, a credential, and possibly an anchor.

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, a deposit (required to propose a governance action) and an anchor (see Figure 10).

To submit a governance action proposal 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 13). Specifically, we have `NetworkGroup`, `EconomicGroup`, `TechnicalGroup`, and `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 11).

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.

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 12).

```
threshold : PParams → Maybe ℚ → GovAction → GovRole → Maybe ℚ
threshold pp ccThreshold' = λ where
    NoConfidence           → ⟨ noVote           , vote P1     , vote Q1 ⟩
    (NewCommittee _ _ _)   → case ccThreshold' of λ where
      (just _)             → ⟨ noVote           , vote P2a    , vote Q2a ⟩
      nothing              → ⟨ noVote           , vote P2b    , vote Q2b ⟩
    (NewConstitution _ _) → ⟨ vote ccThreshold , vote P3     , noVote ⟩
    (TriggerHF _)          → ⟨ vote ccThreshold , vote P4     , vote Q4 ⟩
    (ChangePParams x)      → ⟨ vote ccThreshold , vote (P5 x) , noVote ⟩
    (TreasuryWdrl _)       → ⟨ vote ccThreshold , vote P6     , noVote ⟩
    Info                   → ⟨ vote 2ℚ          , vote 2ℚ     , vote 2ℚ ⟩
  where
    open PParams pp
    open DrepThresholds drepThresholds
    open PoolThresholds poolThresholds

    -- Here, 2 can just be any number strictly greater than one. It just
    -- means that a threshold can never be cleared, i.e. that the action
    -- cannot be enacted.

    ccThreshold : ℚ
    ccThreshold = case ccThreshold' of λ where
      (just x) → x
      nothing  → 2ℚ

    pparamThreshold : PParamGroup → ℚ
    pparamThreshold NetworkGroup    = P5a
    pparamThreshold EconomicGroup   = P5b
    pparamThreshold TechnicalGroup  = P5c
    pparamThreshold GovernanceGroup = P5d

    P5 : PParamsUpdate → ℚ
    P5 ppu = maximum $ mapˢ pparamThreshold (updateGroups ppu)

    noVote : Maybe ℚ
    noVote = nothing

    vote : ℚ → Maybe ℚ
    vote = just

-- TODO: this doesn't actually depend on PParams so we could remove that argument,
--       but we don't have a default ATM
canVote : PParams → GovAction → GovRole → Set
canVote pp a r = Is-just (threshold pp nothing a r)
```

Figure 8: Functions related to voting

```
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 9: NeedsHash and HashProtected types

```
data Vote : Set where
  yes no 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 10: Governance action proposals and votes

```
record EnactEnv : Set where
  constructor ⟦_,_,_⟧ᵉ
  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 , _) = ∅
```

Figure 11: Enactment types

```
data _⊢_⇀⦇_,ENACT⦈_ : EnactEnv → EnactState → GovAction → EnactState → Set where

  Enact-NoConf :
    ⟦ gid , t , e ⟧ᵉ ⊢   s ⇀⦇ NoConfidence ,ENACT⦈
               record s { cc = nothing , gid }

  Enact-NewComm : let old = maybe proj₁ ∅ᵐ (s .EnactState.cc .proj₁) in
    ∀[ term ∈ range new ] term ≤ (s .pparams .proj₁ .PParams.ccMaxTermLength +ᵉ e)
    ────────────────────────────────
    ⟦ gid , t , e ⟧ᵉ ⊢  s ⇀⦇ NewCommittee new rem q ,ENACT⦈
              record s { cc = just ((new ∪ᵐˡ old) ∣ rem ᶜ , q) , gid }

  Enact-NewConst :
    ⟦ gid , t , e ⟧ᵉ ⊢  s ⇀⦇ NewConstitution dh sh ,ENACT⦈
              record s { constitution = (dh , sh) , gid }

  Enact-HF :
    ⟦ gid , t , e ⟧ᵉ ⊢   s ⇀⦇ TriggerHF v ,ENACT⦈
               record s { pv = v , gid }

  Enact-PParams :
    ⟦ gid , t , e ⟧ᵉ ⊢  s ⇀⦇ ChangePParams up ,ENACT⦈
              record s { pparams = applyUpdate (s .pparams .proj₁) up , gid }

  Enact-Wdrl : let newWdrls = s .withdrawals ∪⁺ wdrl in
    Σᵐᵛ[ x ← newWdrls ᶠᵐ ] x ≤ t
    ────────────────────────────────
    ⟦ gid , t , e ⟧ᵉ ⊢  s ⇀⦇ TreasuryWdrl wdrl ,ENACT⦈
              record s { withdrawals = newWdrls }

  Enact-Info :
    ⟦ gid , t , e ⟧ᵉ ⊢  s ⇀⦇ Info ,ENACT⦈ s
```

Figure 12: 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. `ProtVer` represents the protocol version used in the Cardano ledger. It is a pair of natural numbers, representing the major and minor version, respectively.

`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.

The first three of these 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 `GovernanceGroup`. These new parameters are declared in Figure 13 and denote the following concepts.

- `drepThresholds`: governance thresholds for `DRep`s; 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`;
- `ccMinSize`: minimum constitutional committee size;
- `ccMaxTermLength`: maximum term limit (in epochs) of constitutional committee members;
- `govActionLifetime`: governance action expiration;
- `govActionDeposit`: 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 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
    govActionLifetime : ℕ
    govActionDeposit  : Coin
    drepDeposit       : Coin
    drepActivity      : Epoch
    ccMinSize         : ℕ
    ccMaxTermLength   : ℕ
    minimumAVS        : Coin

paramsWellFormed : PParams → Bool
paramsWellFormed pp = ¿ 0 ∉ fromList
  ( maxBlockSize ∷ maxTxSize ∷ maxHeaderSize ∷ maxValSize ∷ minUTxOValue ∷ poolDeposit
  ∷ collateralPercent ∷ ccMaxTermLength ∷ govActionLifetime ∷ govActionDeposit
  ∷ drepDeposit ∷ [])
  × ℕtoEpoch govActionLifetime ≤ drepActivity ¿ᵇ
  where open PParams pp
```

Figure 13: Protocol parameter declarations

9 Governance

Derived types

```
record GovActionState : Set where
  field votes      : (GovRole × Credential) ⇀ Vote
        returnAddr : RwdAddr
        expiresIn  : Epoch
        action     : GovAction
        prevAction : NeedsHash action

GovState : Set
GovState = List (GovActionID × GovActionState)

record GovEnv : Set where
  constructor ⟦_,_,_⟧ᵗ
  field txid    : TxId
        epoch   : Epoch
        pparams : PParams
```

Transition relation types

```
_⊢_⇀⦇_,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 → GovRole → Credential → Vote → GovState
addVote s aid r kh v =
  modifyMatch
    (λ (x , _) → aid ≡ᵇ x)
    (λ (gid , s') → gid , record s' { votes = insert (votes s') (r , kh) v }) s

addAction : GovState
          → Epoch → GovActionID → RwdAddr → (a : GovAction) → NeedsHash a
          → GovState
addAction s e aid addr a prev = s ∷ʳ (aid , record
  { votes = ∅ᵐ ; returnAddr = addr ; expiresIn = e ; action = a ; prevAction = prev })
```

Figure 14: Types and functions used in the GOV transition system

```
GOV-Vote : ∀ {x ast} → let open GovEnv Γ in
  (aid , ast) ∈ fromList s
  → canVote pparams (action ast) role
  ────────────────────────────────────────
  let sig = inj₁ record { gid = aid ; role = role ; credential = cred
                        ; vote = v ; anchor = x }
  in (Γ , k) ⊢ s ⇀⦇ sig ,GOV'⦈ addVote s aid role cred v

GOV-Propose : ∀ {x} → let open GovEnv Γ; open PParams pparams hiding (a) in
  actionWellFormed a ≡ true
  → d ≡ govActionDeposit
  → (∀ {new rem q} → a ≡ NewCommittee new rem q
    → ∀[ e ∈ range new ] epoch < e × dom new ∩ rem ≡ᵉ ∅)
  ────────────────────────────────────────
  let sig = inj₂ record { returnAddr = addr ; action = a ; anchor = x
                        ; deposit = d ; prevAction = prev }
      s' = addAction s (govActionLifetime +ᵉ epoch) (txid , k) addr a prev
  in
  (Γ , k) ⊢ s ⇀⦇ sig ,GOV'⦈ s'

_⊢_⇀⦇_,GOV⦈_ = SS⇒BSᵢ _⊢_⇀⦇_,GOV'⦈_
```

Figure 15: Rules for the GOV transition system

10 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 Credential → DCert

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

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 : Credential ⇀ Maybe Credential

record CertState : Set where
  constructor ⟦_,_,_⟧ᶜ
  field dState : DState
        pState : PState
        gState : GState

GovCertEnv = CertEnv
DelegEnv   = PParams
PoolEnv    = PParams
```

Figure 16: Types used for CERTS transition system

```
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

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

getDRepVote : GovVote → Maybe Credential
getDRepVote record { role = DRep ; credential = credential } = just credential
getDRepVote _                                                = nothing
```

Figure 17: Functions used for CERTS transition system

```
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
    ────────────────────────────────
    pp ⊢ ⟦ pools , retiring ⟧ᵖ ⇀⦇ regpool c poolParams ,POOL⦈
         ⟦ ❴ c , poolParams ❵ᵐ ∪ᵐˡ pools , retiring ⟧ᵖ

  POOL-retirepool :
    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) ⊎ (d ≡ 0 × c ∈ dom dReps)
    ────────────────────────────────
    ⟦ 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
    ────────────────────────────────
    Γ ⊢ ⟦ dReps , ccKeys ⟧ᵛ ⇀⦇ deregdrep c ,GOVCERT⦈
        ⟦ dReps ∣ ❴ c ❵ ᶜ , ccKeys ⟧ᵛ

  GOVCERT-ccreghot :
    (c , nothing) ∉ ccKeys
    ────────────────────────────────
    Γ ⊢ ⟦ dReps , ccKeys ⟧ᵛ ⇀⦇ ccreghot c mc ,GOVCERT⦈
        ⟦ dReps , singletonᵐ c mc ∪ᵐˡ ccKeys ⟧ᵛ
```

Figure 18: Auxiliary DELEG and POOL rules

```
data _⊢_⇀⦇_,CERT⦈_ : CertEnv → CertState → DCert → CertState → Set where
  CERT-deleg :
    pp ⊢ stᵈ ⇀⦇ dCert ,DELEG⦈ stᵈ'
    ────────────────────────────────
    ⟦ e , pp , vs ⟧ᶜ ⊢ ⟦ stᵈ , stᵖ , stᵍ ⟧ᶜ ⇀⦇ dCert ,CERT⦈ ⟦ stᵈ' , stᵖ , stᵍ ⟧ᶜ

  CERT-pool :
    pp ⊢ stᵖ ⇀⦇ dCert ,POOL⦈ stᵖ'
    ────────────────────────────────
    ⟦ e , pp , vs ⟧ᶜ ⊢ ⟦ stᵈ , stᵖ , stᵍ ⟧ᶜ ⇀⦇ dCert ,CERT⦈ ⟦ stᵈ , stᵖ' , stᵍ ⟧ᶜ

  CERT-vdel :
    Γ ⊢ stᵍ ⇀⦇ dCert ,GOVCERT⦈ stᵍ'
    ────────────────────────────────
    Γ ⊢ ⟦ stᵈ , stᵖ , stᵍ ⟧ᶜ ⇀⦇ dCert ,CERT⦈ ⟦ stᵈ , stᵖ , stᵍ' ⟧ᶜ

data _⊢_⇀⦇_,CERTBASE⦈_ : CertEnv → CertState → ⊤ → CertState → Set where
  CERT-base :
    let open PParams pp; open CertState st; open GState gState
        refresh = mapPartial getDRepVote (fromList vs)
    in ⊤ -- TODO: check that the withdrawals are correct here
    ────────────────────────────────
    ⟦ e , pp , vs ⟧ᶜ ⊢ st ⇀⦇ _ ,CERTBASE⦈ record st
      { gState = record gState
        { dreps = mapValueRestricted (const (e + drepActivity)) dreps refresh } }

_⊢_⇀⦇_,CERTS⦈_ : CertEnv → CertState → List DCert → CertState → Set
_⊢_⇀⦇_,CERTS⦈_ = SS⇒BSᵇ _⊢_⇀⦇_,CERTBASE⦈_ _⊢_⇀⦇_,CERT⦈_
```

Figure 19: CERTS rules

11 Transactions

Transactions are defined in Figure 20. 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 : Coin
        txup       : Maybe Update
        txADhash   : Maybe ADHash
        netwrk     : Maybe Network
        txsize     : ℕ
        txid       : TxId

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

  scriptsP1 : ℙ P1Script
  scriptsP1 = mapPartial isInj₁ scripts

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

Figure 20: 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))
```

12 UTxO

12.1 Accounting

Figure 21 defines functions needed for the UTxO transition system. Figure 22 defines the types needed for the UTxO transition system. The UTxO transition system is given in Figure 24.

- 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 (tx .txid ,_) (tx .txouts)

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 = pp .a * tx .txsize + pp .b

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 , pp .poolDeposit)
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

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

-- 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 ≤ 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)
```

Figure 21: Functions used in UTxO rules

Derived types

```
Deposits = DepositPurpose ⇀ Coin
```

UTxO environment

```
record UTxOEnv : Set where
  field slot    : Slot
        ppolicy : Maybe ScriptHash
        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 22: UTxO transition-system types

```
updateCertDeposits : PParams → List DCert → DepositPurpose ⇀ Coin
  → DepositPurpose ⇀ Coin
updateCertDeposits pp [] deposits = deposits
updateCertDeposits pp (cert :: certs) deposits
  = updateCertDeposits pp certs deposits ∪⁺ certDepositᵐ pp cert
  ∣ certRefundˢ cert ᶜ

updateProposalDeposits : PParams → TxId → List GovProposal → DepositPurpose ⇀ Coin
  → DepositPurpose ⇀ Coin
updateProposalDeposits pp txid [] deposits = deposits
updateProposalDeposits pp txid (prop :: props) deposits
  =  updateProposalDeposits pp txid props deposits
  ∪⁺ propDepositᵐ pp (txid , length props) prop

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

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 (st .deposits))

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

consumed : PParams → UTxOState → TxBody → Value
consumed 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 23: Functions used in UTxO rules, continued

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

Figure 24: UTXO inference rules

12.2 Witnessing

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

getScripts : ℙ Credential → ℙ ScriptHash
getScripts = mapPartial isInj₂

credsNeeded : Maybe ScriptHash → UTxO → TxBody → ℙ Credential
credsNeeded sh utxo txb
  = mapˢ (payCred ∘ proj₁) ((utxo ˢ) ⟪$⟫ txins)
  ∪ mapˢ cwitness (fromList txcerts)
  ∪ mapˢ GovVote.credential (fromList txvote)
  ∪ mapPartial (const (inj₂ <$> sh)) (fromList txprop)
  where open TxBody txb

witsVKeyNeeded : Maybe ScriptHash → UTxO → TxBody → ℙ KeyHash
witsVKeyNeeded sh = getVKeys ∘₂ credsNeeded sh

scriptsNeeded : Maybe ScriptHash → UTxO → TxBody → ℙ ScriptHash
scriptsNeeded sh = getScripts ∘₂ credsNeeded sh
```

Figure 25: Functions used for witnessing

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

Figure 26: UTxOW transition-system types

```
UTXOW-inductive :
  let open Tx tx renaming (body to txb); open TxBody txb; open TxWitnesses wits
      open UTxOState s; open UTxOEnv Γ
      witsKeyHashes     = mapˢ hash (dom vkSigs)
      witsScriptHashes = mapˢ hash scripts
    in
      ∀[ (vk , σ) ∈ vkSigs ] isSigned vk (txidBytes txid) σ
  → ∀[ s ∈ scriptsP1 ] validP1Script witsKeyHashes txvldt s
  → witsVKeyNeeded ppolicy utxo txb ⊆ witsKeyHashes
  → scriptsNeeded ppolicy utxo txb ≡ᵉ witsScriptHashes
  → txADhash ≡ map hash txAD
  → Γ ⊢ s ⇀⦇ txb ,UTXO⦈ s'
      ────────────────────────────────
      Γ ⊢ s ⇀⦇ tx ,UTXOW⦈ s'
```

Figure 27: UTXOW inference rules

13 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
        ppolicy : Maybe ScriptHash
        pparams : PParams

record LState : Set where
  constructor ⟦_,_,_⟧ˡ
  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 28: Types and functions for the LEDGER transition system

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

Figure 29: The type of the LEDGER transition system

```
LEDGER : let open LState s; txb = tx .body; open TxBody txb; open LEnv Γ in
    record { LEnv Γ } ⊢ utxoSt ⇀⦇ tx ,UTXOW⦈ utxoSt'
  → ⟦ epoch slot , pparams , txvote ⟧ᶜ ⊢ certState ⇀⦇ txcerts ,CERTS⦈ certState'
  → ⟦ txid , epoch slot , pparams ⟧ᵗ ⊢ govSt ⇀⦇ txgov txb ,GOV⦈ govSt'
  → mapˢ stake (dom txwdrls) ⊆ dom (certState' .dState .voteDelegs)
    ────────────────────────────────
    Γ ⊢ s ⇀⦇ tx ,LEDGER⦈ ⟦ utxoSt' , govSt' , certState' ⟧ˡ
```

Figure 30: LEDGER transition system

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

Figure 31: LEDGERS transition system

14 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.

14.1 Ratification requirements

Figure 32 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.

14.2 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 33 defines three more types and some helper functions 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;

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 32: Ratification requirements

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

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

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

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

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 33: Types and functions for the RATIFY transition system

- `RatifyState` denotes an enactment state that exists during ratification;
- `CCData` stores data about the constitutional committee.
- `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 34 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.
- `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`.

The code in Figure 35 defines `votedHashes`, which returns the set of delegates who voted a certain way on the given governance role. The code in Figure 36 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`);

```
actualVotes : RatifyEnv → CCData → (GovRole × Credential) ⇀ Vote → GovAction → PParams
            → VDeleg ⇀ Vote
actualVotes Γ cc votes ga pparams
  =   mapKeys (credVoter CC) actualCCVotes
  ∪ˡ actualPDRepVotes ∪ˡ actualDRepVotes
  ∪ˡ actualSPOVotes
  where
    open RatifyEnv Γ
    open PParams pparams

    roleVotes : GovRole → VDeleg ⇀ Vote
    roleVotes r = mapKeys (uncurry credVoter) (filterᵐ? ((r ≟_) ∘ proj₁ ∘ proj₁) votes)

    activeCC activeDReps : ℙ Credential
    activeCC = case cc of λ where
      (just (cc , _)) → dom (filterᵐᵇ (is-just ∘ proj₂) (ccHotKeys ∣ dom cc))
      nothing         → ∅

    activeDReps = dom (filterᵐ? (currentEpoch ≤ᵉ?_ ∘ proj₂) dreps)

    actualCCVote : Credential → Epoch → Vote
    actualCCVote c e =
      case ¿ currentEpoch ≤ e ¿ᵇ , lookupᵐ? ccHotKeys c of λ where
        (true , just (just c')) → maybe id Vote.no $ lookupᵐ? votes (CC , c')
        _                       → Vote.abstain -- expired, no hot key or resigned

    actualCCVotes : Credential ⇀ Vote
    actualCCVotes = case cc , ¿ ccMinSize ≤ lengthˢ activeCC ¿ᵇ of λ where
      (just (cc , _) , true)  → mapWithKey actualCCVote cc
      (just (cc , _) , false) → constMap (dom cc) Vote.no
      (nothing       , _)     → ∅ᵐ

    actualPDRepVotes
      =   ❴ abstainRep       , Vote.abstain ❵ᵐ
      ∪ˡ ❴ noConfidenceRep , (case ga of λ where NoConfidence → Vote.yes
                                                 _            → Vote.no) ❵ᵐ

    actualDRepVotes
      =   roleVotes GovRole.DRep
      ∪ˡ constMap (mapˢ (credVoter DRep) activeDReps) Vote.no

    actualSPOVotes
      =   roleVotes GovRole.SPO
      ∪ˡ constMap spos (if isHF then Vote.no else Vote.abstain)
      where
        spos : ℙ VDeleg
        spos = filterˢ isSPOProp $ dom (StakeDistrs.stakeDistr stakeDistrs)

        isHF : Bool
        isHF = case ga of λ where
          (TriggerHF _) → true
          _             → false
```

Figure 34: Types and proofs for the ratification of governance actions

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

Figure 35: Calculation of the votes as they will be counted

```
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 ᶜ ᶠᵐ ] x

accepted' : RatifyEnv → EnactState → GovActionState → Set
accepted' Γ (record { cc = cc , _ ; pparams = pparams , _ }) gs =
  acceptedBy CC ∧ acceptedBy DRep ∧ acceptedBy SPO ∧ meetsMinAVS
  where
    open RatifyEnv Γ; open GovActionState gs; open PParams pparams

    votes' = actualVotes Γ cc votes action pparams
    cc' = dom votes'
    redStakeDistr = restrictedDists coinThreshold rankThreshold stakeDistrs

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

    acceptedBy : GovRole → Set
    acceptedBy role =
      let t = maybe id R.0ℚ $ threshold pparams (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 < current
```

Figure 36: Calculation of stake distributions

```
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 37: Determination of the status of ratification of the governance action

- `expired` checks whether a governance action is expired in a given epoch.

The code in Figure 37 defines still more types required for ratification of a governance action.

- `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 38 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`.

- `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.

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.

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

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

RATIFY-Continue : let open RatifyEnv Γ; st = a .proj₂; open GovActionState st in
    ¬ accepted Γ es st × ¬ expired currentEpoch st
  ⊎ delayed action prevAction es d
  ⊎ accepted Γ es st
    × ¬ delayed action prevAction es d
    × (∀ es' → ¬ ⟦ a .proj₁ , treasury , currentEpoch ⟧ᵉ ⊢ es ⇀⦇ action ,ENACT⦈ es')
  ────────────────────────────────────────
  Γ ⊢ ⟦ es , removed , d ⟧ʳ ⇀⦇ a ,RATIFY'⦈ ⟦ es , removed , d ⟧ʳ

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

Figure 38: The RATIFY transition system

15 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        : EnactState
        fut       : RatifyState

record ChainState : Set where
  field newEpochState : NewEpochState

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

Figure 39: Definitions for the NEWEPOCH and CHAIN transition systems

```
NEWEPOCH-New : ∀ {Γ} → let
  open NewEpochState nes 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 = Σᵐᵛ[ x ← trWithdrawals ᶠᵐ ] x

  removedGovActions = flip concatMapˢ removed λ (gaid , gaSt) →
      mapˢ (GovActionState.returnAddr gaSt ,_)
           ((deposits ∣ ❴ GovActionDeposit gaid ❵) ˢ)
  govActionReturns = aggregate₊ $ mapˢ (λ (a , _ , d) → a , d) removedGovActions ᶠˢ

  es      = record esW { withdrawals = ∅ᵐ }
  retired = retiring ⁻¹ e
  refunds = govActionReturns ∪⁺ trWithdrawals ∣ dom rewards
  unclaimed = govActionReturns ∪⁺ trWithdrawals ∣ dom rewards ᶜ

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

  gState' = record gState { ccHotKeys = ccHotKeys ∣ ccCreds (es .EnactState.cc) }

  certState' = record certState {
      pState = record pState
        { pools = pools ∣ retired ᶜ ; retiring = retiring ∣ retired ᶜ };
      dState = record dState
        { rewards = rewards ∪⁺ refunds };
      gState = if not (null govSt') then gState' else record gState'
        { dreps = mapValues sucᵉ dreps }
    }
  utxoSt' = record utxoSt
      { fees = 0
      ; deposits = deposits ∣ mapˢ (proj₁ ∘ proj₂) removedGovActions ᶜ
      ; donations = 0
      }
  ls' = record ls
      { govSt = govSt' ; utxoSt = utxoSt' ; certState = certState' }
  acnt' = record acnt
      { treasury = treasury + fees + getCoin unclaimed + donations ∸ totWithdrawals }
  in
  e ≡ sucᵉ lastEpoch
  → record { currentEpoch = e ; treasury = treasury ; GState gState ; NewEpochEnv Γ }
      ⊢ ⟦ es , ∅ , false ⟧ʳ ⇀⦇ govSt' ,RATIFY⦈ fut'
  ────────────────────────────────
  Γ ⊢ nes ⇀⦇ e ,NEWEPOCH⦈ ⟦ e , acnt' , ls' , es , fut' ⟧ⁿᵉ

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


Figure 40: NEWEPOCH transition system

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

Figure 41: Type of the CHAIN transition system

```
CHAIN :
  let open ChainState s; open Block b; open NewEpochState nes; open EnactState es in
    record { stakeDistrs = calculateStakeDistrs ls }
      ⊢ newEpochState ⇀⦇ epoch slot ,NEWEPOCH⦈ nes
  → ⟦ slot , constitution .proj₁ .proj₂ , pparams .proj₁ ⟧ˡᵉ
      ⊢ ls ⇀⦇ ts ,LEDGERS⦈ ls'
  ────────────────────────────────
  _ ⊢ s ⇀⦇ b ,CHAIN⦈ record s { newEpochState = record nes { ls = ls' } }
```

Figure 42: CHAIN transition system

16 Properties

16.1 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 43: Computing the UTXO transition system

Property 16.1 (Preserve Balance)
For all $env \in$ `UTxOEnv`, $utxo, utxo' \in$ `UTxO`, $fees, fees' \in$ `Coin` *and* $tx \in$ `TxBody`,
if

```
txid tx ∉ mapˢ proj₁ (dom utxo)
```

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
```