Contents

1 Introduction

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

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

Era	Figures	Prose	Cleanup
Shelley	Partial	Partial	Not started
Shelley-MA	Partial	Partial	Not started
Alonzo	Partial	Partial	Not started
Babbage	Not started	Not started	Not started
Conway [2]	Complete	Partial	Partial

Table 1: Specification progress

1.1 A note on Agda

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

1.2 Separation of concerns

The *Cardano Node* consists of three pieces:

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

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

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

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.

1.3 Reflexive-transitive closure

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

The closure `_⊢_⇀⦇_⦈*_`of a relation `_⊢_⇀⦇_⦈_`is defined in Figure 1. In the remainder of the text, the closure operation is called `ReflexiveTransitiveClosure`.

Closure type

```
_⊢_⇀⦇_⦈*_ : C → S → List Sig → S → Set
```

Closure rules

```
RTC-base :
  Γ ⊢ s ⇀⦇ [] ⦈* s

RTC-ind :
  • Γ ⊢ s ⇀⦇ sig ⦈ s'
  • Γ ⊢ s' ⇀⦇ sigs ⦈* s''
  ──────────────────────────────
  Γ ⊢ s ⇀⦇ sig :: sigs ⦈* s''
```

Figure 1: Reflexive transitive closure

1.4 Computational

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

```
record Computational (_⊢_⇀⦇_,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'

  nothing⇒∀¬STS : compute Γ s b ≡ nothing → ∀ s' → ¬ Γ ⊢ s ⇀⦇ b ,X⦈ s'
```

Figure 2: Computational relations

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

This has two further implications:

- Since `compute` is a function, the step relation is necessarily right-unique, i.e. there is at most one possible final state for each input data. Otherwise, we could prove that `compute` could evaluates to two different states on the same inputs, which is impossible since it is a function.
- The actual definition of `compute` is irrelevant - any two implementations of `compute` have to produce the same result on any input. This is because we can simply chain the equivalences for two different `compute` functions together.

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

1.5 Sets & maps

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

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

```
_⊆_ : {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 = r ∈ Rel A B , left-unique r
```

1.6 Superscripts and other special notations

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

- $\cup^{l}$ for left-biased union
- ^{c} in the context of set restrictions, where it indicates the complement

Also, non-letter superscripts do carry meaning.

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

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

2 Notation

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

Propositions, sets and types This document loosly treats sets and types as the same thing.

Sums and products The sum (or disjoint union, coproduct, etc.) of `A` and `B` is denoted by `A ⊎ B`, and their product is denoted by `A × B`.

Record types Record types are explained in Appendix A.

Postfix projections Projections can be written using postix notation. For example, we may write `x .proj₁` instead of `proj₁ x`.

Restriction, corestriction and complements The restriction of a function or map `f` to some domain `A` is denoted by `f ∣ A`, and the restriction to the complement of `A` is written `f ∣ A ᶜ`. Corestriction or range restriction is denoted the same, except that `∣` is replaced by `↾`.

Inverse image The expression `m ⁻¹ B` denotes the inverse image of the set `B` under the map `m`.

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

3 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 3: Definitions for the public key signature scheme

4 Base types

```
Coin  = ℕ
Slot  = ℕ
Epoch = ℕ
```

Figure 4: Some basic types used in many places in the ledger

5 Token algebras

Abstract types

```
PolicyId
```

Derived types

```
record TokenAlgebra : Set₁ where
  field Value : Set
        ⦃ Value-IsCommutativeMonoid' ⦄ : IsCommutativeMonoid' 0ℓ 0ℓ Value

  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 -- FIXME: rename!
        coinIsMonoidHomomorphism : IsMonoidHomomorphism coin
```

Helper functions

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

Figure 5: Token algebras, used for multi-assets

6 Addresses

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

Abstract types

```
Network
KeyHash
ScriptHash
```

Derived types

```
Credential = KeyHash ⊎ 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
isScriptAddr : Addr → Set

isVKeyAddr      = isVKey ∘ payCred
isScriptAddr    = isScript ∘ payCred
isScriptRwdAddr = isScript ∘ RwdAddr.stake
```

Figure 6: Definitions used in Addresses

7 Scripts

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

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


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

Figure 7: Timelock scripts and their evaluation

8 Protocol parameters

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

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

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

ProtVer : Set
ProtVer = ℕ × ℕ

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

Figure 8: Definitions related to protocol parameters

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

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

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

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

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

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

```
  data PParamGroup : Set where
    NetworkGroup EconomicGroup TechnicalGroup GovernanceGroup SecurityGroup : 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 Q5e : ℚ

  record PParams : Set where
    field
```

Network group

```
      maxBlockSize maxTxSize       : ℕ
      maxHeaderSize maxValSize     : ℕ
      maxCollateralInputs          : ℕ
      pv                           : ProtVer -- retired, keep for now
      maxTxExUnits maxBlockExUnits : ExUnits
```

Economic group

```
      a b                          : ℕ
      minUTxOValue poolDeposit     : Coin
      keyDeposit                   : Coin
      coinsPerUTxOWord             : Coin
      minFeeRefScriptCoinsPerByte  : ℚ
      prices                       : Prices
```

Technical group

```
      a0                           : ℚ
      Emax                         : Epoch
      nopt                         : ℕ
      collateralPercentage         : ℕ
      -- costmdls                   : Language ↛/⇀ CostModel (Does not work with DecEq)
      costmdls                     : CostModel
```

Governance group

```
      drepThresholds               : DrepThresholds
      poolThresholds               : PoolThresholds
      govActionLifetime            : ℕ
      govActionDeposit drepDeposit : Coin
      drepActivity                 : Epoch
      ccMinSize ccMaxTermLength    : ℕ

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

Figure 9: Protocol parameter declarations

- `govActionDeposit`: governance action deposit;

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

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

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

Abstract types & functions

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

Well-formedness condition

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

Figure 10: Abstract type for parameter updates

9 Governance actions

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

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

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

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

Voter       = GovRole × Credential
GovActionID = TxId × ℕ

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

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

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

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

Figure 11: Governance actions

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

- `GovActionID`—a unique identifier for a governance action, consisting of the `TxId` of the proposing transaction and an index to identify a proposal within a transaction;
- `GovRole` (*governance role*)—one of three available voter roles defined above (`CC`, `DRep`, `SPO`);
- `VDeleg` (*voter delegation*)—one of three ways to delegate votes: by credential, abstention, or no confidence (`credVoter`, `abstainRep`, or `noConfidenceRep`);

- `Anchor`—a url and a document hash;
- `GovAction` (*governance action*)—one of seven possible actions (see Figure 12 for definitions).
- `actionWellFormed`—in the case of protocol parameter changes, an action is well-formed if it preserves the well-formedness of parameters. `ppdWellFormed` is effectively the same as `paramsWellFormed`, except that it only applies to the parameters that are being changed.

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 terms
`NewConstitution`	a modification to the off-chain Constitution and the proposal policy script
`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
`Info`	an action that has no effect on-chain, other than an on-chain record

Figure 12: Types of governance actions

9.1 Hash protection

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

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

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

9.2 Votes and proposals

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

[1] There are many varying definitions of the term "hard fork" in the blockchain industry. Hard forks typically refer to non-backwards compatible updates of a network. In Cardano, we formalize the definition slightly more by calling any upgrade that would lead to *more blocks* being validated a "hard fork" and force nodes to comply with the new protocol version, effectively obsoleting nodes that are unable to handle the upgrade.

```
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 13: NeedsHash and HashProtected types

To propose a governance action, a `GovProposal` needs to be submitted. Beside the proposed action, it requires:

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

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

`GovActionState` is the state of an individual governance action. It contains the individual votes, its lifetime, and information necessary to enact the action and to repay the deposit.

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

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

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

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

Figure 14: Vote and proposal types

10 Transactions

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

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

Abstract types

```
Ix TxId AuxiliaryData : Set
```

Derived types

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

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

Transaction types

```
record TxBody : Set where
  field txins         : ℙ TxIn
        refInputs     : ℙ TxIn
        txouts        : Ix → TxOut
        txfee         : Coin
        mint          : Value
        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
        collateral    : ℙ TxIn
        reqSigHash    : ℙ KeyHash
        scriptIntHash : Maybe ScriptHash
record TxWitnesses : Set where
  field vkSigs  : VKey → Sig
        scripts : ℙ Script
        txdats  : DataHash → Datum
        txrdmrs : RdmrPtr → Redeemer × ExUnits

  scriptsP1 : ℙ P1Script
  scriptsP1 = mapPartial isInj₁ scripts

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

Figure 15: Transactions and related types

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

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

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

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

txinsVKey : ℙ TxIn → UTxO → ℙ TxIn
txinsVKey txins utxo = txins ∩ dom (utxo ↑' (isVKeyAddr ∘ proj₁))

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

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

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

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

lookupScriptHash : ScriptHash → Tx → UTxO → Maybe Script
lookupScriptHash sh tx utxo =
  if sh ∈ mapˢ proj₁ (m ˢ) then
    just (lookupᵐ m sh)
  else
    nothing
  where m = setToHashMap (txscripts tx utxo)
```

Figure 16: Functions related to transactions

11 UTxO

11.1 Accounting

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


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

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

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

Figure 17: Functions supporting UTxO rules

Figures 17, 18, and 19 define functions needed for the UTxO transition system. Note the special multiplication symbol $*↓$ used in Figure 18: it means multiply and round down the result.

Figure 20 defines the types needed for the UTxO transition system. The UTxO transition system is given in Figure 22.

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

```
  outs : TxBody → UTxO
  outs tx = mapKeys (tx .txid ,_) (tx .txouts)

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

  cbalance : UTxO → Coin
  cbalance utxo = coin (balance utxo)
  minfee : PParams → UTxO → Tx → Coin
  minfee pp utxo tx =
    pp .a * tx .body .txsize + pp .b
    + txscriptfee (pp .prices) (totExUnits tx)
    + pp .minFeeRefScriptCoinsPerByte
    *↓ ∑[ x ← mapValues scriptSize (setToHashMap (refScripts tx utxo)) ] x

  data DepositPurpose : Set where
    CredentialDeposit : Credential  → DepositPurpose
    PoolDeposit       : Credential  → DepositPurpose
    DRepDeposit       : Credential  → DepositPurpose
    GovActionDeposit  : GovActionID → DepositPurpose
updateCertDeposits : PParams → List DCert → DepositPurpose ⇀ Coin → DepositPurpose ⇀ Coin
updateCertDeposits _  []            deposits = deposits
updateCertDeposits pp (cert :: certs) deposits
  = (updateCertDeposits pp certs deposits ∪⁺ certDeposit cert {pp}) ∣ certRefund cert ᶜ
  where
  certDeposit : DCert → {pp : PParams} → DepositPurpose ⇀ Coin
  certDeposit (delegate c _ _ v) = ❴ CredentialDeposit c , v            ❵
  certDeposit (regpool c _) {pp} = ❴ PoolDeposit     c , pp .poolDeposit ❵
  certDeposit (regdrep c v _)    = ❴ DRepDeposit     c , v               ❵
  certDeposit _                  = ∅

  certRefund : DCert → ℙ DepositPurpose
  certRefund (dereg c)     = ❴ CredentialDeposit c ❵
  certRefund (deregdrep c) = ❴ DRepDeposit c ❵
  certRefund _             = ∅

updateProposalDeposits : List GovProposal → TxId → Coin → DepositPurpose ⇀ Coin
  → DepositPurpose ⇀ Coin
updateProposalDeposits []       _    _     deposits = deposits
updateProposalDeposits (_ :: ps) txid gaDep deposits =
  updateProposalDeposits ps txid gaDep deposits
  ∪⁺ ❴ GovActionDeposit (txid , length ps) , gaDep ❵

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

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

Figure 18: Functions used in UTxO rules

```
data inInterval (slot : Slot) : (Maybe Slot × Maybe Slot) → Set where
  both  : ∀ {l r} → l ≤ slot × slot ≤ r → inInterval slot (just l  , just r)
  lower : ∀ {l}   → l ≤ slot            → inInterval slot (just l  , nothing)
  upper : ∀ {r}   → slot ≤ r            → inInterval slot (nothing , just r)
  none  :                                 inInterval slot (nothing , nothing)

feesOK : PParams → Tx → UTxO → Bool
feesOK pp tx utxo = minfee pp utxo tx ≤ᵇ txfee
                  ∧ not (¿-∅ᵇ (txrdmrs ˢ))
                  =>ᵇ ( allᵇ (λ (addr , _) → ¿ isVKeyAddr addr ¿) collateralRange
                      ∧ isAdaOnlyᵇ bal
                      ∧ (coin bal * 100) ≥ᵇ (txfee * pp .collateralPercentage)
                      ∧ not (¿-∅ᵇ collateral)
                      )
  where
    open Tx tx; open TxBody body; open TxWitnesses wits; open PParams pp
    collateralRange = range   ((mapValues txOutHash utxo) ⁉ collateral)
    bal             = balance (utxo ⁉ collateral)
```

Figure 19: Functions used in UTxO rules, continued

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 → Tx → UTxOState → Set
```

Figure 20: UTxO transition-system types

```
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 21: Functions used in UTxO rules, continued

```
UTXO-inductive :
  let open Tx tx renaming (body to txb); open TxBody txb
      open UTxOEnv Γ renaming (pparams to pp)
      open UTxOState s
      txouts^h = (mapValues txOutHash txouts)
  in
  • txins ≢ ∅                  • txins ∪ refInputs ⊆ dom utxo
  • txins ∩ refInputs ≡ ∅      • inInterval slot txvldt
  • feesOK pp tx utxo ≡ true   • consumed pp s txb ≡ produced pp s txb
  • coin mint ≡ 0              • txsize ≤ maxTxSize pp

  • ∀[ (_ , txout) ∈ txouts^h .proj₁ ]
      inject (utxoEntrySize txout * minUTxOValue pp) ≤^t getValue^h txout
  • ∀[ (_ , txout) ∈ txouts^h .proj₁ ]
      serSize (getValue^h txout) ≤ maxValSize pp
  • ∀[ (a , _) ∈ range txouts^h ]
      Sum.All (const ⊤) (λ a → a .BootstrapAddr.attrsSize ≤ 64) a
  • ∀[ (a , _) ∈ range txouts^h ] netId a ≡ networkId
  • ∀[ a ∈ dom txwdrls ] a .RwdAddr.net ≡ networkId
  • Γ ⊢ s ⇀⦇ tx ,UTXOS⦈ s'
    ──────────────────────────────
    Γ ⊢ s ⇀⦇ tx ,UTXO⦈ s'
```

Figure 22: UTXO inference rules

11.2 Witnessing

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

getScripts : ℙ Credential → ℙ ScriptHash
getScripts = mapPartial isInj₂

credsNeeded : UTxO → TxBody → ℙ (ScriptPurpose × Credential)
credsNeeded utxo txb
  = mapˢ (λ (i , o) → (Spend i , payCred (proj₁ o))) ((utxo ∣ txins) ˢ)
  ∪ mapˢ (λ a → (Rwrd a , RwdAddr.stake a)) (dom (txwdrls .proj₁))
  ∪ mapˢ (λ c → (Cert c , cwitness c)) (fromList txcerts)
  ∪ mapˢ (λ x → (Mint x , inj₂ x)) (policies mint)
  ∪ mapˢ (λ v → (Vote v , proj₂ v)) (fromList $ map GovVote.voter txvote)
  ∪ mapPartial (λ p → case p .GovProposal.policy of
                      (just sh) → just (Propose p , inj₂ sh)
                      nothing   → nothing) (fromList txprop)

witsVKeyNeeded : UTxO → TxBody → ℙ KeyHash
witsVKeyNeeded = getVKeys ∘₂ mapˢ proj₂ ∘₂ credsNeeded

scriptsNeeded : UTxO → TxBody → ℙ ScriptHash
scriptsNeeded = getScripts ∘₂ mapˢ proj₂ ∘₂ credsNeeded
```

Figure 23: Functions used for witnessing

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

Figure 24: UTxOW transition-system types

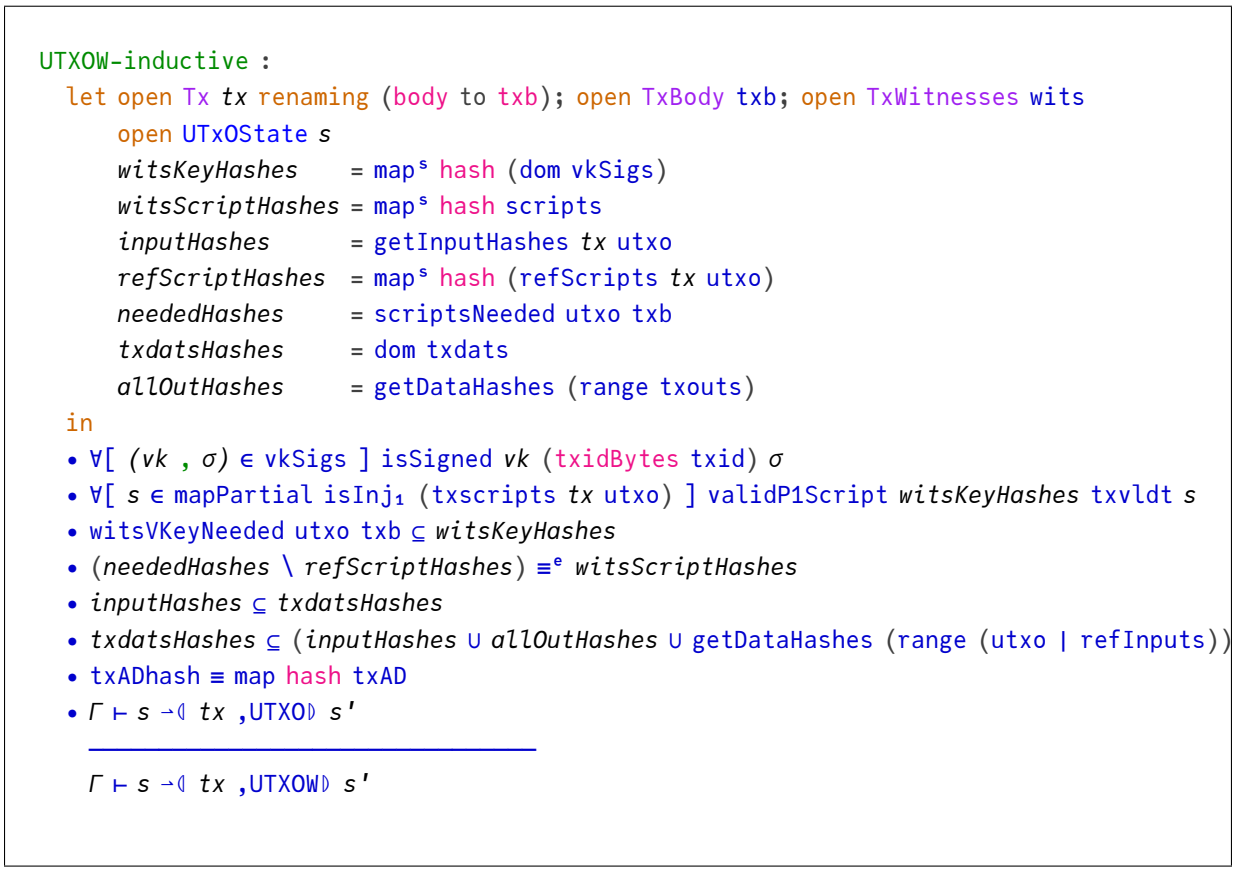

Figure 25: UTXOW inference rules

12 Governance

Derived types

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

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

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

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

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

Figure 26: Types and functions used in the GOV transition system[2]

`GovState` behaves similar to a queue. New proposals are appended at the end, but any proposal can be removed at the epoch boundary. However, for the purposes of enactment, earlier proposals take priority.

- `addVote` inserts (and potentially overrides) a vote made for a particular governance action (identified by its ID) by a credential with a role.
- `addAction` adds a new proposed action at the end of a given `GovState`.
- `validHFAction` is the property whether a given proposal, if it is a `TriggerHF` action, can

[2] l ∷ʳ x appends element x to list l.

```
-- convert list of (GovActionID,GovActionState)-pairs to list GovActionID pairs.
getAidPairsList : GovState → List (GovActionID × GovActionID)
getAidPairsList aid×states =
  mapMaybe (λ (aid , aState) → (aid ,_) <$> getHash (prevAction aState)) $ aid×states

-- convert list of (GovActionID,GovActionState)-pairs to SET of GovActionID pairs.
getAidPairsSet : GovState → ℙ (GovActionID × GovActionID)
getAidPairsSet aid×states =
  mapPartial (λ (aid , as) → (aid ,_) <$> getHash (prevAction as)) $ fromList aid×states

-- a list of GovActionID pairs connects the first GovActionID to the second
_connects_to_ : List (GovActionID × GovActionID) → GovActionID → GovActionID → Set
[] connects aidNew to aidOld = aidNew ≡ aidOld
((aid , aidPrev) :: s) connects aidNew to aidOld = aid ≡ aidNew × s connects aidPrev to aidOld
                                                 ⊎ s connects aidNew to aidOld

enactable : EnactState → List (GovActionID × GovActionID) → GovActionID × GovActionState → Set
enactable e aidPairs = λ (aidNew , as) → case getHashES e (GovActionState.action as) of λ where
  nothing → ⊤
  (just aidOld) → ∃[ t ] fromList t ⊆ fromList aidPairs × Unique t × t connects aidNew to aidOld

allEnactable : EnactState → GovState → Set
allEnactable e aid×states = All (enactable e (getAidPairsList aid×states)) aid×states

hasParentE : EnactState → GovActionID → GovAction → Set
hasParentE e aid a = case getHashES e a of λ where
  nothing → ⊤
  (just id) → id ≡ aid

hasParent : EnactState → GovState → (a : GovAction) → NeedsHash a → Set
hasParent e s a aid with getHash aid
... | just aid' = hasParentE e aid' a ⊎ Any (λ x → proj₁ x ≡ aid') s
... | nothing = ⊤
```

Figure 27: Enactability predicate

potentially be enacted in the future. For this to be the case, its `prevAction` needs to exist, be another `TriggerHF` action and have a compatible version.

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

For `GOV-Vote`, we check that the governance action being voted on exists and the role is allowed to vote. `canVote` is defined in Figure 40.

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

- for `ChangePParams` or `TreasuryWdrl`, the proposal policy needs to be provided;
- for `NewCommittee`, no proposals with members expiring in the present or past epoch are allowed, and candidates cannot be added and removed at the same time;
- and we check the validity of hard-fork actions via `validHFAction`.

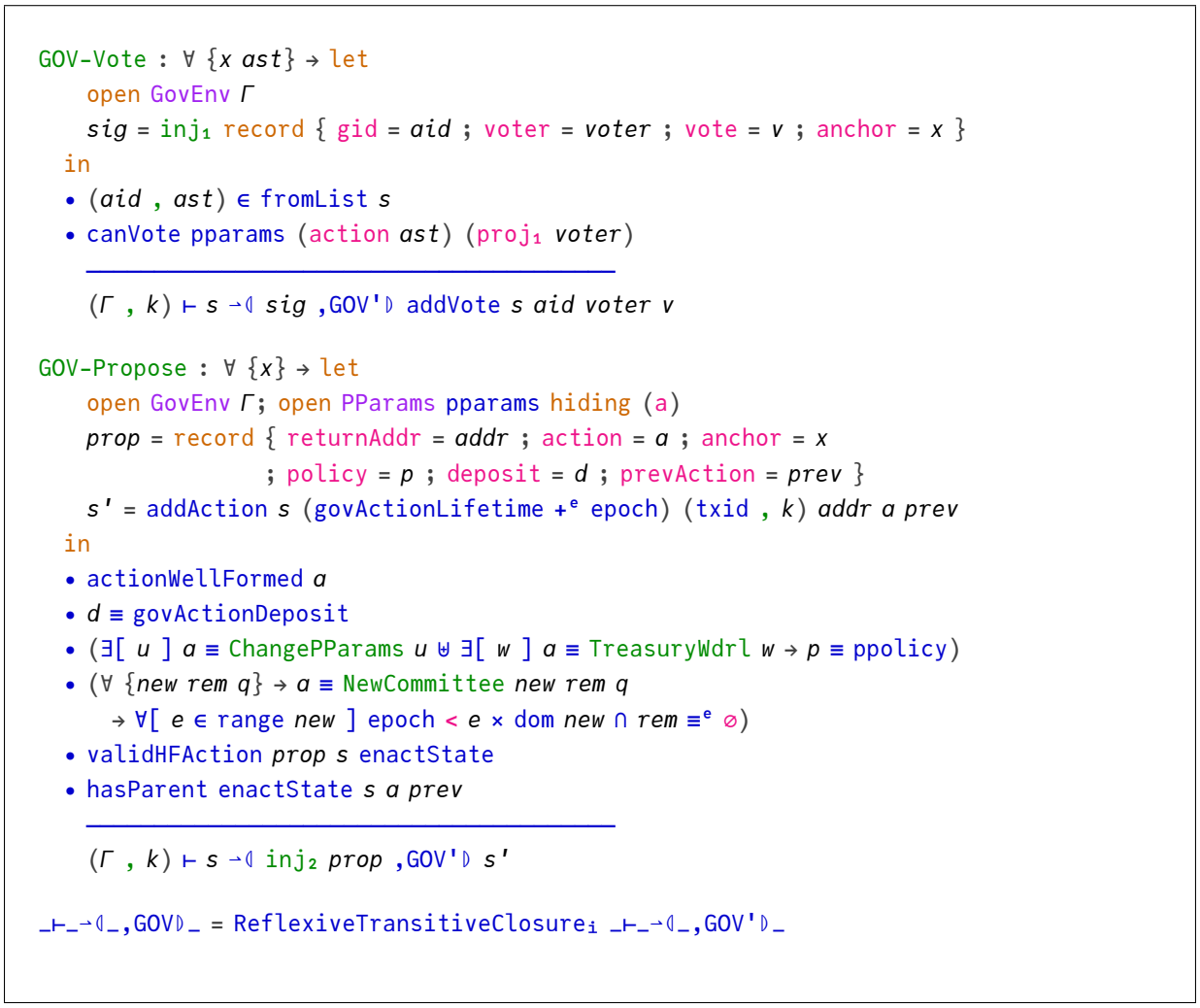

Figure 28: Rules for the GOV transition system

13 Delegation

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

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

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

Figure 29: Delegation definitions

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

- `GOVCERT-regdrep` registers (or re-registers) a DRep. In case of registation, a deposit needs to be paid. Either way, the activity period of the DRep is reset.
- `GOVCERT-deregdrep` deregisters a DRep.
- `GOVCERT-ccreghot` registers a hot credential for constitutional committee members. We check that the cold key did not previously resign from the committee. Note that we intentionally do not check if the cold key is actually part of the committee; if it isn't, then the corresponding hot key does not carry any voting power. By allowing this, a newly elected member of the constitutional committee can immediately delegate their vote to a hot key and use it to vote. Since votes are counted after previous actions have been enacted, this allows constitutional committee members to act without a delay of one epoch.

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

- check the correctness of withdrawals and ensure that withdrawals only happen from credentials that have delegated their voting power;
- set the rewards of the credentials that withdrew funds to zero;
- and set the activity timer of all DReps that voted to `drepActivity` epochs in the future.

```
record CertEnv : Set where
  constructor ⟦_,_,_,_⟧ᶜ
  field epoch : Epoch
        pp    : PParams
        votes : List GovVote
        wdrls : RwdAddr → Coin

record DState : Set where
  constructor ⟦_,_,_⟧ᵈ
  field
    voteDelegs  : Credential → VDeleg
    stakeDelegs : Credential → Credential
    rewards     : Credential → Coin

record PState : Set where
  constructor ⟦_,_⟧ᵖ
  field pools    : Credential → PoolParams
        retiring : Credential → Epoch

record GState : Set where
  constructor ⟦_,_⟧ᵛ
  field dreps     : Credential → Epoch
        ccHotKeys : Credential → Maybe Credential

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

record DelegEnv : Set where
  constructor ⟦_,_⟧ᵈᵉ
  field pparams : PParams
        pools   : Credential → PoolParams

GovCertEnv = CertEnv
PoolEnv    = PParams
```

Figure 30: Types used for CERTS transition system

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

Figure 31: Functions used for CERTS transition system

```
_⊢_⇀⦇_,DELEG⦈_ : DelegEnv → DState → DCert → DState → Set

DELEG-delegate : let open PParams pp in
  • (c ∉ dom rwds → d ≡ keyDeposit)
  • (c ∈ dom rwds → d ≡ 0)
  • mc ∈ mapˢ just (dom pools) ∪ { nothing }
    ──────────────────────────────
    ⟦ pp , pools ⟧ᵈᵉ ⊢
      ⟦ vDelegs , sDelegs , rwds ⟧ᵈ ⇀⦇ delegate c mv mc d ,DELEG⦈
      ⟦ insertIfJust c mv vDelegs , insertIfJust c mc sDelegs , rwds ∪ˡ { c , 0 } ⟧ᵈ

DELEG-dereg :
  • (c , 0) ∈ rwds
    ──────────────────────────────
    ⟦ pp , pools ⟧ᵈᵉ ⊢ ⟦ vDelegs , sDelegs , rwds ⟧ᵈ ⇀⦇ dereg c ,DELEG⦈
                        ⟦ vDelegs ∣ { c }ᶜ , sDelegs ∣ { c }ᶜ , rwds ∣ { c }ᶜ ⟧ᵈ

_⊢_⇀⦇_,POOL⦈_ : PoolEnv → PState → DCert → PState → Set

POOL-regpool :
  • c ∉ dom pools
    ──────────────────────────────
    pp ⊢ ⟦ pools , retiring ⟧ᵖ ⇀⦇ regpool c poolParams ,POOL⦈
         ⟦ { c , poolParams } ∪ˡ pools , retiring ⟧ᵖ

POOL-retirepool :
  ──────────────────────────────
  pp ⊢ ⟦ pools , retiring ⟧ᵖ ⇀⦇ retirepool c e ,POOL⦈ ⟦ pools , { c , e } ∪ˡ retiring ⟧ᵖ

_⊢_⇀⦇_,GOVCERT⦈_ : GovCertEnv → GState → DCert → GState → Set
GOVCERT-regdrep : let open PParams pp in
  • (d ≡ drepDeposit × c ∉ dom dReps) ⊎ (d ≡ 0 × c ∈ dom dReps)
    ──────────────────────────────
    ⟦ e , pp , vs , wdrls ⟧ᶜ ⊢ ⟦ dReps , ccKeys ⟧ᵛ ⇀⦇ regdrep c d an ,GOVCERT⦈
                               ⟦ { c , e + drepActivity } ∪ˡ 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 , { c , mc } ∪ˡ ccKeys ⟧ᵛ
```

Figure 32: Auxiliary DELEG, POOL and GOVCERT transition systems

```
_⊢_⇀⦇_,CERT⦈_ : CertEnv → CertState → DCert → CertState → Set


CERT-deleg :
  • ⟦ pp , PState.pools stᵖ ⟧ᵈᵉ ⊢ stᵈ ⇀⦇ dCert ,DELEG⦈ stᵈ'
    ──────────────────────────────
    ⟦ e , pp , vs , wdrls ⟧ᶜ ⊢ ⟦ stᵈ , stᵖ , stᵍ ⟧ᶜˢ ⇀⦇ dCert ,CERT⦈ ⟦ stᵈ' , stᵖ , stᵍ ⟧ᶜˢ

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

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



_⊢_⇀⦇_,CERTBASE⦈_ : CertEnv → CertState → ⊤ → CertState → Set


CERT-base :
  let open PParams pp; open GState stᵍ; open DState stᵈ
      refresh        = mapPartial getDRepVote (fromList vs)
      refreshedDReps = mapValueRestricted (const (e + drepActivity)) dreps refresh
      wdrlCreds      = mapˢ RwdAddr.stake (dom wdrls)
  in
  • wdrlCreds ⊆ dom voteDelegs
  • mapˢ (map₁ RwdAddr.stake) (wdrls ˢ) ⊆ rewards ˢ
    ──────────────────────────────
    ⟦ e , pp , vs , wdrls ⟧ᶜ ⊢ ⟦ stᵈ , stᵖ , stᵍ ⟧ᶜˢ ⇀⦇ _ ,CERTBASE⦈
      ⟦ ⟦ voteDelegs , stakeDelegs , constMap wdrlCreds 0 ∪ˡ rewards ⟧ᵈ
      , stᵖ
      , ⟦ refreshedDReps , ccHotKeys ⟧ᵛ ⟧ᶜˢ

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

Figure 33: CERTS rules

14 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
        enactState : EnactState

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 34: Types and functions for the LEDGER transition system

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

Figure 35: The type of the LEDGER transition system

```
LEDGER-V : let open LState s; txb = tx .body; open TxBody txb; open LEnv Γ in
  • isValid tx ≡ true
  • record { LEnv Γ } ⊢ utxoSt ⇀⦇ tx ,UTXOW⦈ utxoSt'
  • ⟦ epoch slot , pparams , txvote , txwdrls ⟧ᶜ ⊢ certState ⇀⦇ txcerts ,CERTS⦈ certState'
  • ⟦ txid , epoch slot , pparams , ppolicy , enactState ⟧ᵍ ⊢ govSt ⇀⦇ txgov txb ,GOV⦈ govSt'
    ──────────────────────────────────
    Γ ⊢ s ⇀⦇ tx ,LEDGER⦈ ⟦ utxoSt' , govSt' , certState' ⟧ˡ

LEDGER-I : let open LState s; txb = tx .body; open TxBody txb; open LEnv Γ in
  • isValid tx ≡ false
  • record { LEnv Γ } ⊢ utxoSt ⇀⦇ tx ,UTXOW⦈ utxoSt'
    ──────────────────────────────────
    Γ ⊢ s ⇀⦇ tx ,LEDGER⦈ ⟦ utxoSt' , govSt , certState ⟧ˡ
```

Figure 36: LEDGER transition system

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

Figure 37: LEDGERS transition system

15 Enactment

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

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

```
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 , _) = ∅

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

open EnactState

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

Figure 38: Types and function used for the ENACT transition system

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

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

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


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

Enact-NewComm : let old     = maybe proj₁ ∅ (s .EnactState.cc .proj₁)
                    maxTerm = s .pparams .proj₁ .PParams.ccMaxTermLength +ᵉ e
                in
  ∀[ term ∈ range new ] term ≤ maxTerm
  ──────────────────────────────────────
  ⟦ gid , t , e ⟧ᵉ ⊢ s ⇀⦇ NewCommittee new rem q ,ENACT⦈
            record s { cc = just ((new ∪ˡ 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
  ∑[ x ← newWdrls ] 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 39: ENACT transition system

16 Ratification

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

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

16.1 Ratification requirements

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

The symbols mean the following:

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

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

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

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

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

Each of the P_x and Q_x are protocol parameters.

16.2 Protocol parameters and governance actions

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

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

```
threshold : PParams → Maybe ℚ → GovAction → GovRole → Maybe ℚ
threshold pp ccThreshold =
  NoConfidence          → | –  | vote P1 | vote Q1          |
  (NewCommittee _ _ _)  → | –  ‖ P/Q2a/b                    |
  (NewConstitution _ _) → | ✓  | vote P3 | –               |
  (TriggerHF _)         → | ✓  | vote P4 | vote Q4         |
  (ChangePParams x)     → | ✓  ‖ P/Q5 x                     |
  (TreasuryWdrl _)      → | ✓  | vote P6 | –               |
  Info                  → | ✓† | ✓†      | ✓†              |
    where
    P/Q2a/b : Maybe ℚ × Maybe ℚ
    P/Q2a/b = case ccThreshold of
              (just _) → (vote P2a , vote Q2a)
              nothing → (vote P2b , vote Q2b)

    pparamThreshold : PParamGroup → Maybe ℚ × Maybe ℚ
    pparamThreshold NetworkGroup    = (vote P5a , –        )
    pparamThreshold EconomicGroup   = (vote P5b , –        )
    pparamThreshold TechnicalGroup  = (vote P5c , –        )
    pparamThreshold GovernanceGroup = (vote P5d , –        )
    pparamThreshold SecurityGroup   = (–        , vote Q5e )

    P/Q5 : PParamsUpdate → Maybe ℚ × Maybe ℚ
    P/Q5 ppu = maxThreshold (mapˢ (proj₁ ∘ pparamThreshold) (updateGroups ppu))
             , maxThreshold (mapˢ (proj₂ ∘ pparamThreshold) (updateGroups ppu))

canVote : PParams → GovAction → GovRole → Set
canVote pp a r = Is-just (threshold pp nothing a r)
```

Figure 40: Functions related to voting

16.3 Ratification restrictions

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

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

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

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

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

```
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) × ℚ)

govRole : VDeleg → GovRole
govRole (credVoter gv _) = gv
govRole abstainRep       = DRep
govRole noConfidenceRep  = DRep

IsCC IsDRep IsSPO : VDeleg → Set
IsCC    v = govRole v ≡ CC
IsDRep  v = govRole v ≡ DRep
IsSPO   v = govRole v ≡ SPO
```

Figure 41: Types and functions for the RATIFY transition system

– if the `CC` member has not voted, then the default value of `no` is returned.

- `actualDRepVotes` adds a default vote of `no` to all active DReps that didn't vote.
- `actualSPOVotes` adds a default vote to all SPOs who didn't vote, with the default depending on the action.

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

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

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

  roleVotes : GovRole → VDeleg ⇀ Vote
  roleVotes r = mapKeys (uncurry credVoter) (filter (λ (x , _) → r ≡ proj₁ x) votes)

  activeDReps = dom (filter (λ (_ , e) → currentEpoch ≤ e) dreps)
  spos = filterˢ IsSPO (dom (stakeDistr stakeDistrs))

  getCCHotCred : Credential × Epoch → Maybe Credential
  getCCHotCred (c , e) = case ¿ currentEpoch ≤ e ¿ᵇ , lookupᵐ? ccHotKeys c of
     (true , just (just c')) → just c'
     _                       → nothing -- expired, no hot key or resigned

  actualCCVote : Credential → Epoch → Vote
  actualCCVote c e = case getCCHotCred (c , e) of
     (just c') → maybe id Vote.no (lookupᵐ? votes (CC , c'))
     _         → Vote.abstain

  activeCC : (Credential ⇀ Epoch) → ℙ Credential
  activeCC m = mapPartial getCCHotCred (m ˢ)

  actualCCVotes : Credential ⇀ Vote
  actualCCVotes = case cc of
     nothing         → ∅
     (just (m , q))  → if ccMinSize ≤ lengthˢ (activeCC m)
                       then mapWithKey actualCCVote m
                       else constMap (dom m) Vote.no

  actualPDRepVotes : GovAction → VDeleg ⇀ Vote
  actualPDRepVotes NoConfidence
                    = ❴ abstainRep , Vote.abstain ❵ ∪ˡ ❴ noConfidenceRep , Vote.yes ❵
  actualPDRepVotes _ = ❴ abstainRep , Vote.abstain ❵ ∪ˡ ❴ noConfidenceRep , Vote.no ❵

  actualDRepVotes : VDeleg ⇀ Vote
  actualDRepVotes =  roleVotes DRep
                 ∪ˡ constMap (mapˢ (credVoter DRep) activeDReps) Vote.no

  actualSPOVotes : GovAction → VDeleg ⇀ Vote
  actualSPOVotes (TriggerHF _) = roleVotes SPO ∪ˡ constMap spos Vote.no
  actualSPOVotes _             = roleVotes SPO ∪ˡ constMap spos Vote.abstain
```

Figure 42: Vote counting

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

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

acceptedStakeRatio : GovRole → ℙ VDeleg → StakeDistrs → (VDeleg ⇀ Vote) → ℚ
acceptedStakeRatio r cc dists votes = acceptedStake /₀ totalStake
  where
    acceptedStake totalStake : Coin
    acceptedStake = ∑[ x ← getStakeDist r cc dists | votes ⁻¹ Vote.yes       ] x
    totalStake    = ∑[ x ← getStakeDist r cc dists | votes ⁻¹ Vote.abstain ᶜ ] x

acceptedBy : RatifyEnv → EnactState → GovActionState → GovRole → Set
acceptedBy Γ (record { cc = cc , _; pparams = pparams , _ }) gs role =
  let open GovActionState gs
      votes' = actualVotes Γ pparams cc action votes
      t      = maybe id 0ℚ (threshold pparams (proj₂ <$> cc) action role)
  in acceptedStakeRatio role (dom votes') (stakeDistrs Γ) votes' ≥ t

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

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

Figure 43: Functions used in RATIFY rules, without delay

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

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

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

- `RATIFY-Accept` checks if the votes for a given `GovAction` meet the threshold required for acceptance, that the action is accepted and not delayed, and `RATIFY-Accept` ratifies the action.
- `RATIFY-Reject` asserts that the given `GovAction` is not `accepted` and `expired`; it removes the governance action.
- `RATIFY-Continue` covers the remaining cases and keeps the `GovAction` around for further voting.

```
verifyPrev : (a : GovAction) → NeedsHash a → EnactState → Set
verifyPrev NoConfidence          h es = h ≡ es .cc .proj₂
verifyPrev (NewCommittee _ _ _)  h es = h ≡ es .cc .proj₂
verifyPrev (NewConstitution _ _) h es = h ≡ es .constitution .proj₂
verifyPrev (TriggerHF _)         h es = h ≡ es .pv .proj₂
verifyPrev (ChangePParams _)     h es = h ≡ es .pparams .proj₂
verifyPrev (TreasuryWdrl _)      _ _  = ⊤
verifyPrev Info                  _ _  = ⊤

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

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

Figure 44: Functions relating to delays

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

We never remove actions that do not attract sufficient `yes` votes before they expire, even if it is clear to an outside observer that this action will never be enacted. Such an action will simply keep getting checked every epoch until it expires.

```
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
  ⊎ 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⦈_ = ReflexiveTransitiveClosure _⊢_⇀⦇_,RATIFY'⦈_
```

Figure 45: The RATIFY transition system

17 Epoch boundary

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

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

record NewEpochState : Set where
  constructor ⟦_,_⟧ⁿᵉ
  field lastEpoch  : Epoch
        epochState : EpochState
```

Figure 46: Definitions for the EPOCH and NEWEPOCH transition systems

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

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

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

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

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

  removedGovActions = flip concatMapˢ removed λ (gaid , gaSt) →
    mapˢ (GovActionState.returnAddr gaSt ,_)
         ((deposits ⁉ { GovActionDeposit gaid }) ˢ)
  govActionReturns = aggregate₊ (mapˢ (λ (a , _ , d) → a , d) removedGovActions ᶠˢ)

  es       = record esW { withdrawals = ∅ }
  retired = retiring ⁻¹ e
  payout  = govActionReturns ∪⁺ trWithdrawals
  refunds = pullbackMap payout (λ x → record { net = NetworkId ; stake = x }) (dom rewards)
  unclaimed = getCoin payout ∸ getCoin refunds

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

  certState' =
    ⟦ record dState { rewards = rewards ∪⁺ refunds }
    , ⟦ pools ⁉ retiredᶜ , retiring ⁉ retiredᶜ ⟧ᵖ
    , ⟦ if null govSt' then mapValues (1 +_) dreps else dreps
      , ccHotKeys ⁉ ccCreds (es .EnactState.cc) ⟧ᵛ ⟧ᶜˢ

  utxoSt' = ⟦ utxo , 0 , deposits ⁉ mapˢ (proj₁ ∘ proj₂) removedGovActionsᶜ , 0 ⟧ᵘ

  ls' = ⟦ utxoSt' , govSt' , certState' ⟧ˡ

  acnt' = record acnt
    { treasury = treasury + fees + unclaimed + donations ∸ totWithdrawals }
  in
  record { currentEpoch = e ; treasury = treasury ; GState gState ; NewEpochEnv Γ }
    ⊢ ⟦ es , ∅ , false ⟧ʳ ⇀⦇ govSt' ,RATIFY⦈ fut'
  ──────────────────────────────
  Γ ⊢ eps ⇀⦇ e ,EPOCH⦈ ⟦ acnt' , ls' , es , fut' ⟧ᵉ'
```

Figure 47: EPOCH transition system

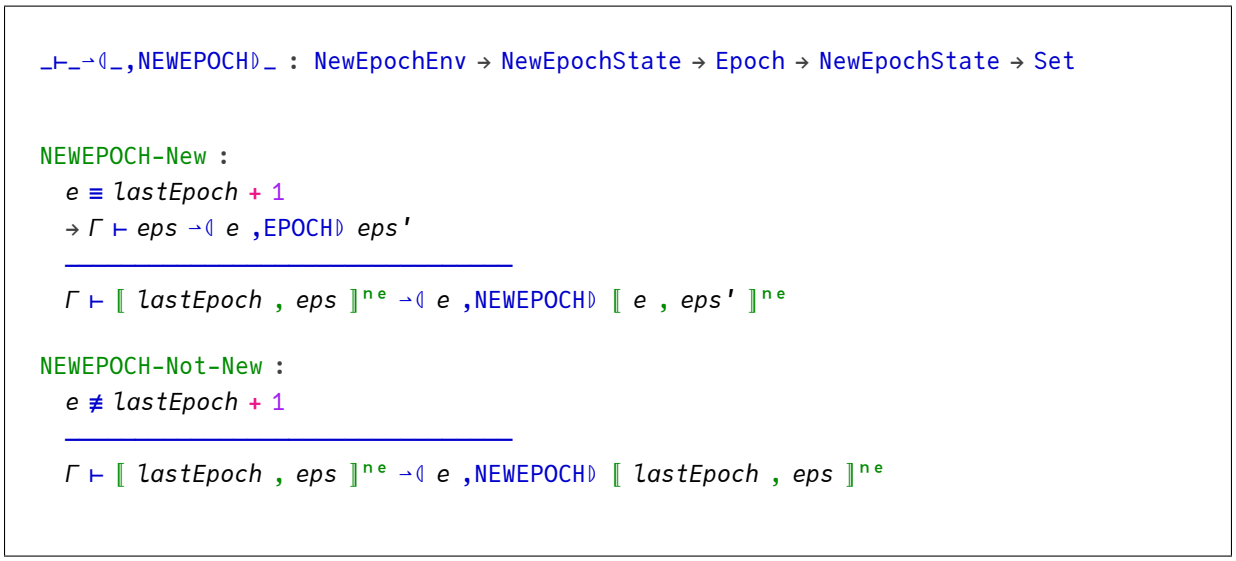

Figure 48: NEWEPOCH transition system

18 Blockchain layer

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

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

Figure 49: Definitions CHAIN transition system

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

Figure 50: Type of the CHAIN transition system

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

Figure 51: CHAIN transition system

19 Properties

19.1 UTxO

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

```
UTXO-step : UTxOEnv → UTxOState → Tx → ComputationResult String UTxOState
UTXO-step = compute ⦃ Computational-UTXO ⦄

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

Figure 52: Computing the UTXO transition system

Property 19.1 (Preserve Balance)
For all Γ ∈ *UTxOEnv*, utxo, utxo' ∈ *UTxO*, fees, fees' ∈ *Coin and* tx ∈ *Tx*,
if

```
tx .body .txid ∉ map ˢ 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
```

Property 19.2 (General Minimum Spending Condition)

References

[1] Agda development team. Agda 2.6.4 documentation. https://agda.readthedocs.io/en/v2.6.4/, December 2023.

[2] J. Corduan, M. Benkort, K. Hammond, C. Hoskinson, A. Knispel, and S. Leathers. A first step towards on-chain decentralized governance. https://cips.cardano.org/cip/CIP-1694, 2023.

A Appendix: Agda essentials

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

A.1 Record types

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

Example.

```
record Pair (A B : Set) : Set where
  constructor ⟪_,_⟫
  field
    fst : A
    snd : B
```

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

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

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

```
p23' : Pair ℕ ℕ
p23' = ⟪ 2 , 3 ⟫
```

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

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

This results a new record, `p24`, which denotes the pair `⟪ 2 , 4 ⟫`.

See also https://agda.readthedocs.io/en/v2.6.4/language/record-types.

B Bootstrapping EnactState

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

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

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

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