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 avoid. 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. Thus each such relation represents the transition rule of the state machine; X is simply a placeholder for the name of the transition rule.

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 → Type
```

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 nodes 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 state transition relation.

```
record Computational (_⊢_⇀⦇_,X⦈_ : C → S → Sig → S → Type) : Type where
  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 state transition relation is necessarily a (partial) function; 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 necessarily defined everywhere—equivalently, "left-unique" relations) and we use the harpoon arrow ⇀ to distinguish such maps from standard Agda functions which use →. The figure below also gives notation for the powerset operation, ℙ, used to form a type of sets with elements in a given type, as well as the subset relation and the equality relation for sets.

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

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

Rel : Type → Type → Type
Rel A B = ℙ (A × B)

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

_⇀_ : Type → Type → Type
A ⇀ B = r ∈ Rel A B , left-unique r
```

1.6 Propositions as Types, Properties and Relations

In type theory we represent propositions as types and proofs of a proposition as elements of the corresponding type. A unary predicate is a function that takes each x (of some type A) and returns a proposition $P(x)$. Thus, a predicate is a function of type A → `Type`. A *binary relation* R between A and B is a function that takes a pair of values x and y and returns a proposition asserting that the relation R holds between x and y. Thus, such a relation is a function of type $A \times B$ → `Type` or $A \to B$ → `Type`.

1.7 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.[1]

Finally, there are some ? and ¿ operations. These relate to decision procedures and can also safely be ignored.[2]

[1] 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.

[2] We plan on refactoring the code so that these special symbols will also disappear from this document.

2 Notation

This section introduces some of the notation we use in this document and in our Agda formalization.

Propositions, sets and types. In this document the abstract notions of "set" and "type" are essentially the same, despite having different formal definitions in our Agda code. We represent sets as a special type, which we denote by `Set` A, for A an arbitrary type. (See Section 1.5 for details and [4, Chapter 19] for background.) Agda denotes the primitive notion of type by `Set`. To avoid confusion, throughout this document and in our Agda code we call this primitive `Type`, reserving the name `Set` for our set type. All of our sets are finite, and when we need to convert a list l to its set of elements, we write `fromList` l.

Lists We use the notation a `::` as for the list with *head* a and *tail* as; `[]` denotes the empty list, and l `::ʳ` x appends the element x to the end of the list l.

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

Addition of map values. The expression `∑[ x ← m ] f x` denotes the sum of the values obtained by applying the function `f` to the values of the map `m`.

Record types are explained in Appendix A.

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

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

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

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

Map addition. For maps m and m', we write $m \cup^{+} m'$ for their union, where keys that appear in both maps have their corresponding values added.

Mapping a partial function. A *partial function* is a function on A which may not be defined for all elements of A. We denote such a function by $f : A \rightharpoonup B$. If we happen to know that the function is *total* (defined for all elements of A), then we write $f : A \to B$. The `mapPartial` operation takes such a function f and a set S of elements of A and applies f to the elements of S at which it is defined; the result is the set $\{f\ x \mid x\ \in\ S$ and f is defined at $x\}$.

The `Maybe` type represents an optional value and can either be `just` x (indicating the presence of a value, x) or `nothing` (indicating the absence of a value). If x has type `X`, then `just` x has type `Maybe X`.

The unit type `⊤` has a single inhabitant `tt` and may be thought of as a type that carries no information; it is useful for signifying the completion of an action, the presence of a trivial value, a trivially satisfied requirement, etc.

3 Cryptographic Primitives

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

Types & functions

```
SKey VKey Sig Ser : Type
isKeyPair        : SKey → VKey → Type
isSigned         : VKey → Ser → Sig → Type
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 : Type₁ where
    Value : Set
    ⦃ Value-CommutativeMonoid ⦄ : CommutativeMonoid 0ℓ 0ℓ Value

    coin                        : Value → Coin
    inject                      : Coin → Value
    policies                    : Value → ℙ PolicyId
    size                        : Value → MemoryEstimate
    _≤ᵗ_                        : Value → Value → Type
    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 script (`isScript`).

N.B. in the Shelley era the type of the `stake` field of the `BaseAddr` record was `Credential`; to specify an address with no stake, we would use an "enterprise" address. In contrast, the type of `stake` in the Conway era is `Maybe Credential`, so we can now use `BaseAddr` to specify an address with no stake by setting `stake` to `nothing`.

Abstract types

```
    Network
    KeyHash
    ScriptHash
```

Derived types

```
  data Credential : Type where
    KeyHashObj : KeyHash → Credential
    ScriptObj : ScriptHash → Credential

  record BaseAddr : Type where
    field net   : Network
          pay   : Credential
          stake : Maybe Credential

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

  record RwdAddr : Type 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
  stakeCred    : Addr → Maybe Credential
  netId        : Addr → Network
  isVKeyAddr   : Addr → Type
  isScriptAddr : Addr → Type

  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 : Type 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 → Type 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 : Type where
    treasury reserves : Coin

ProtVer : Type
ProtVer = ℕ × ℕ

instance
  Show-ProtVer : Show ProtVer
  Show-ProtVer = Show-×

data pvCanFollow : ProtVer → ProtVer → Type 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;
- `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.

```
data PParamGroup : Type where
  NetworkGroup    : PParamGroup
  EconomicGroup   : PParamGroup
  TechnicalGroup  : PParamGroup
  GovernanceGroup : PParamGroup
  SecurityGroup   : PParamGroup

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

record PoolThresholds : Type where
    Q1 Q2a Q2b Q4 Q5e : ℚ

record PParams : Type where
```

Network group

```
    maxBlockSize                 : ℕ
    maxTxSize                    : ℕ
    maxHeaderSize                : ℕ
    maxTxExUnits                 : ExUnits
    maxBlockExUnits              : ExUnits
    maxValSize                   : ℕ
    maxCollateralInputs          : ℕ
```

Economic group

```
    a                            : ℕ
    b                            : ℕ
    keyDeposit                   : Coin
    poolDeposit                  : Coin
    coinsPerUTxOByte             : Coin
    prices                       : Prices
    minFeeRefScriptCoinsPerByte  : ℚ
    maxRefScriptSizePerTx        : ℕ
    maxRefScriptSizePerBlock     : ℕ
    refScriptCostStride          : ℕ
    refScriptCostMultiplier      : ℚ
```

Technical group

```
    Emax                         : Epoch
    nopt                         : ℕ
    a0                           : ℚ
    collateralPercentage         : ℕ
    costmdls                     : CostModel
```

Governance group

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

Figure 9: Protocol parameter definitions

```
positivePParams : PParams → List ℕ
positivePParams pp = ( maxBlockSize ∷ maxTxSize ∷ maxHeaderSize ∷ maxValSize ∷ refScriptCostStride
                     ∷ coinsPerUTxOByte ∷ poolDeposit ∷ collateralPercentage ∷ ccMaxTermLength
                     ∷ govActionLifetime ∷ govActionDeposit ∷ drepDeposit ∷ [] )

paramsWellFormed : PParams → Type
paramsWellFormed pp = 0 ∉ fromList (positivePParams pp)
```

Figure 10: Protocol parameter well-formedness

Abstract types & functions

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

Well-formedness condition

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

Figure 11: 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 : Type where
  CC DRep SPO : GovRole

Voter       = GovRole × Credential
GovActionID = TxId × ℕ

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

record Anchor : Type where
    url  : String
    hash : DocHash

data GovAction : Type where
  NoConfidence    :                                            GovAction
  UpdateCommittee : (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 → Type
actionWellFormed (ChangePParams x) = ppdWellFormed x
actionWellFormed (TreasuryWdrl x)  =
  (∀[ a ∈ dom x ] RwdAddr.net a ≡ NetworkId)
  × (∃[ v ∈ range x ] ¬ (v ≡ 0))
actionWellFormed _                 = ⊤
```

Figure 12: Governance actions

Figure 12 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 13 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.

The governance actions carry the following information:

- `UpdateCommittee`: a map of credentials and terms to add and a set of credentials to remove from the committee;
- `NewConstitution`: a hash of the new constitution document and an optional proposal policy;
- `TriggerHF`: the protocol version of the epoch to hard fork into;
- `ChangePParams`: the updates to the parameters; and
- `TreasuryWdrl`: a map of withdrawals.

Action	Description
`NoConfidence`	a motion to create a *state of no-confidence* in the current constitutional committee
`UpdateCommittee`	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`[3]	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 13: Types of governance actions

9.1 Hash Protection

For some governance actions, in addition to obtaining the necessary votes, enactment requires that the following condition is also satisfied: the state obtained by enacting the proposal is in fact the state that was intended when the proposal 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

[3]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 attach a bit more meaning to the definition 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 rendering a node obsolete if it is unable to handle the upgrade.

enacted proposal modifying the same piece of state. NoConfidence and UpdateCommittee 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 14.

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

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

Figure 14: NeedsHash and HashProtected types

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.

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 voting stake (but not the block production 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 deposit 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 : Type where
  yes no abstain : Vote

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

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

record GovActionState : Type where
    votes       : Voter ⇀ Vote
    returnAddr : RwdAddr
    expiresIn  : Epoch
    action      : GovAction
    prevAction : NeedsHash action
```

Figure 15: Vote and proposal types

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

proposedCC : GovAction → ℙ Credential
proposedCC (UpdateCommittee x _ _) = dom x
proposedCC _ = ∅
```

Figure 16: Governance helper function

10 Transactions

Transactions are defined in Figure 17. 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 `txins` 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 `txouts` 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 `txsize` and the hash `txid` of the serialized form of the transaction that was included in the block.

Abstract types

```
Ix TxId AuxiliaryData : Type
```

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 : Type where
  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
  txNetworkId    : Maybe Network
  curTreasury    : Maybe Coin
  txsize         : ℕ
  txid           : TxId
  collateral     : ℙ TxIn
  reqSigHash     : ℙ KeyHash
  scriptIntHash : Maybe ScriptHash
record TxWitnesses : Type where
  vkSigs  : VKey ⇀ Sig
  scripts : ℙ Script
  txdats  : DataHash ⇀ Datum
  txrdmrs : RdmrPtr ⇀ Redeemer × ExUnits

scriptsP1 : ℙ P1Script
scriptsP1 = mapPartial isInj₁ scripts

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

Figure 17: 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 → List Script
refScripts tx utxo =
  mapMaybe (proj₂ ∘ proj₂ ∘ proj₂) $ setToList (range (utxo ∣ (txins ∪ refInputs)))
  where open Tx; open TxBody (tx .body)

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

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

Figure 18: Functions related to transactions

11 UTxO

11.1 Accounting

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


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

  getInputHashes : Tx → UTxO → ℙ DataHash
  getInputHashes tx utxo = getDataHashes
    (filter (λ (a , _ ) → isTwoPhaseScriptAddress′ tx utxo a)
            (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 19: Functions supporting UTxO rules

Figures 19, 21, and 22 define functions needed for the UTxO transition system. Note the special multiplication symbol `*↓` used in Figure 21: it means multiply and take the absolute value of the result, rounded down to the nearest integer.

Figure 20 defines the types needed for the UTxO transition system. The UTxO transition system is given in Figure 25.

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

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

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

There are two equivalent ways to introduce this tracking of the deposits. One option would be to populate the `deposits` field of `UTxOState` with the correct keys and values that can be extracted from the state of the previous era at the transition into the Conway era. Alternatively, we can effectively treat the old handling of deposits as an erratum in the Shelley specification, which we fix by implementing the new deposits logic in older eras and then replaying the chain.

UTxO environment

```
record UTxOEnv : Type where
    slot     : Slot
    pparams  : PParams
    treasury : Coin
```

UTxO states

```
record UTxOState : Type where
    utxo      : UTxO
    fees      : Coin
    deposits  : Deposits
    donations : Coin
```

UTxO transitions

```
  _⊢_⇀⦇_,UTXO⦈_ : UTxOEnv → UTxOState → Tx → UTxOState → Type
```

Figure 20: UTxO transition-system types

As seen in Figures 21 and 23, we redefine `depositRefunds` and `newDeposits` via `depositsChange`, which computes the difference between the total deposits before and after their application. This simplifies their definitions and some correctness proofs. We then add the absolute value of `depositsChange` to `consumed` or `produced` depending on its sign. This is done via `negPart` and `posPart`, which satisfy the key property that their difference is the identity function.

Figures 21 also shows the signature of `ValidCertDeposits`. Inhabitants of this type are constructed in one of eight ways, corresponding to seven certificate types plus one for an empty list of certificates. Suffice it to say that `ValidCertDeposits` is used to check the validity of the deposits in a transaction so that the function `updateCertDeposits` can correctly register and deregister deposits in the UTxO state based on the certificates in the transaction.

Figure 25 ties all the pieces of the UTXO rule together. (The `_≡?_` symbol that appears in the figure denotes a special equality where the value on the left-handside is optional; equality holds if and only if the value on the left is present and equal to the value on the right.)

```
  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)
  refScriptsSize : UTxO → Tx → ℕ
  refScriptsSize utxo tx = sum $ map scriptSize (refScripts tx utxo)

  minfee : PParams → UTxO → Tx → Coin
  minfee pp utxo tx = pp .a * tx .body .txsize + pp .b
                    + txscriptfee (pp .prices) (totExUnits tx)
                    + scriptsCost pp (refScriptsSize utxo tx)

certDeposit : DCert → PParams → Deposits
certDeposit (delegate c _ _ v) _ = ❴ CredentialDeposit c , v ❵
certDeposit (reg c _)       pp   = ❴ CredentialDeposit c , pp .keyDeposit ❵
certDeposit (regpool kh _)  pp   = ❴ PoolDeposit kh , 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 _               = ∅

data ValidCertDeposits (pp : PParams) (deps : Deposits) : List DCert → Set

updateCertDeposits : PParams → List DCert → Deposits → Deposits
updateCertDeposits pp [] deposits = deposits
updateCertDeposits pp (delegate c vd khs v :: certs) deposits
  = updateCertDeposits pp certs (deposits ∪⁺ certDeposit (delegate c vd khs v) pp)
updateCertDeposits pp (regpool kh p :: certs) deposits
  = updateCertDeposits pp certs (deposits ∪⁺ certDeposit (regpool kh p) pp)
updateCertDeposits pp (regdrep c v a :: certs) deposits
  = updateCertDeposits pp certs (deposits ∪⁺ certDeposit (regdrep c v a) pp)
updateCertDeposits pp (dereg c v :: certs) deposits
  = updateCertDeposits pp certs (deposits | certRefund (dereg c v)ᶜ)
updateCertDeposits pp (deregdrep c v :: certs) deposits
  = updateCertDeposits pp certs (deposits | certRefund (deregdrep c v)ᶜ)
updateCertDeposits pp (_ :: certs) deposits
  = updateCertDeposits pp certs deposits

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

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

depositsChange : PParams → TxBody → Deposits → ℤ
depositsChange pp txb deposits =
```

```
data inInterval (slot : Slot) : (Maybe Slot × Maybe Slot) → Type 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 → Type
feesOK pp tx utxo = ( minfee pp utxo tx ≤ txfee × (txrdmrs  ≢ ∅
                      → ( All (λ (addr , _) → isVKeyAddr addr) collateralRange
                        × isAdaOnly bal
                        × coin bal * 100 ≥ txfee * pp .collateralPercentage
                        × 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 22: Functions used in UTxO rules, continued

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

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

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

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

```
_⊢_⇀⦇_,UTXOS⦈_ : UTxOEnv → UTxOState → Tx → UTxOState → Type

Scripts-Yes :
  ∀ {Γ} {s} {tx}
  → let open Tx tx renaming (body to txb); open TxBody txb
        open UTxOEnv Γ renaming (pparams to pp)
        open UTxOState s
        sLst = collectPhaseTwoScriptInputs pp tx utxo
    in
      • ValidCertDeposits pp deposits txcerts
      • evalScripts tx sLst ≡ isValid
      • isValid ≡ true
      ────────────────────────────────
Γ ⊢ s ⇀⦇ tx ,UTXOS⦈ ⟦ (utxo ∣ txins ᶜ) ∪ˡ (outs txb)
                     , fees + txfee
                     , updateDeposits pp txb deposits
                     , donations + txdonation ⟧

Scripts-No :
  ∀ {Γ} {s} {tx}
  → let open Tx tx renaming (body to txb); open TxBody txb
        open UTxOEnv Γ renaming (pparams to pp)
        open UTxOState s
        sLst = collectPhaseTwoScriptInputs pp tx utxo
    in
      • evalScripts tx sLst ≡ isValid
      • isValid ≡ false
      ────────────────────────────────
Γ ⊢ s ⇀⦇ tx ,UTXOS⦈ ⟦ utxo ∣ collateral ᶜ
                     , fees + cbalance (utxo ∣ collateral)
                     , deposits
                     , donations ⟧
```

Figure 24: UTXOS rule

```
UTXO-inductive :
  let open Tx tx renaming (body to txb); open TxBody txb
      open UTxOEnv Γ renaming (pparams to pp)
      open UTxOState s
      txoutsʰ = (mapValues txOutHash txouts)
      overhead = 160
  in
  • txins ≢ ∅                  • txins ∪ refInputs ⊆ dom utxo
  • txins ∩ refInputs ≡ ∅ • inInterval slot txvldt
  • feesOK pp tx utxo          • consumed pp s txb ≡ produced pp s txb
  • coin mint ≡ 0              • txsize ≤ maxTxSize pp
  • refScriptsSize utxo tx ≤ pp .maxRefScriptSizePerTx

  • ∀[ (_ , txout) ∈ txoutsʰ .proj₁ ]
      inject ((overhead + utxoEntrySize txout) * coinsPerUTxOByte pp) ≤ᵗ getValueʰ txout
  • ∀[ (_ , txout) ∈ txoutsʰ .proj₁ ]
      serSize (getValueʰ txout) ≤ maxValSize pp
  • ∀[ (a , _) ∈ range txoutsʰ ]
      Sum.All (const ⊤) (λ a → a .BootstrapAddr.attrsSize ≤ 64) a
  • ∀[ (a , _) ∈ range txoutsʰ ] netId a          ≡ NetworkId
  • ∀[ a ∈ dom txwdrls ]         a .RwdAddr.net ≡ NetworkId
  • txNetworkId ≡? NetworkId
  • curTreasury ≡? treasury
  • Γ ⊢ s ⇀⦇ tx ,UTXOS⦈ s'
    ────────────────────────────────
    Γ ⊢ s ⇀⦇ tx ,UTXO⦈ s'
```

Figure 25: UTXO inference rules

11.2 Witnessing

The purpose of witnessing is make sure the intended action is authorized by the holder of the signing key. (For details see the Formal Ledger Specification for the Shelley Era [3, Sec. 8.3].) Figure 26 defines functions used for witnessing. `witsVKeyNeeded` and `scriptsNeeded` are now defined by projecting the same information out of `credsNeeded`. Note that the last component of `credsNeeded` adds the script in the proposal policy only if it is present.

`allowedLanguages` has additional conditions for new features in Conway. If a transaction contains any votes, proposals, a treasury donation or asserts the treasury amount, it is only allowed to contain Plutus V3 scripts. Additionally, the presence of reference scripts or inline scripts does not prevent Plutus V1 scripts from being used in a transaction anymore. Only inline datums are now disallowed from appearing together with a Plutus V1 script.

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

allowedLanguages : Tx → UTxO → ℙ Language
allowedLanguages tx utxo =
  if (∃[ o ∈ os ] isBootstrapAddr (proj₁ o))
    then ∅
  else if UsesV3Features txb
    then fromList (PlutusV3 ∷ [])
  else if ∃[ o ∈ os ] HasInlineDatum o
    then fromList (PlutusV2 ∷ PlutusV3 ∷ [])
  else
    fromList (PlutusV1 ∷ PlutusV2 ∷ PlutusV3 ∷ [])
  where
    txb = tx .Tx.body; open TxBody txb
    os = range (outs txb) ∪ range (utxo ∣ (txins ∪ refInputs))

getScripts : ℙ Credential → ℙ ScriptHash
getScripts = mapPartial isScriptObj

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

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

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

Figure 26: Functions used for witnessing

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

Figure 27: UTxOW transition-system types

```
UTXOW-inductive :
⟦ utxo , fees , deps , dons ⟧ᵘ      = s
⟦ utxo' , fees' , deps' , dons' ⟧ᵘ = s'
witsKeyHashes                       = map hash (dom vkSigs)
witsScriptHashes                    = map hash scripts
inputHashes                         = getInputHashes tx utxo
refScriptHashes                     = fromList $ map hash (refScripts tx utxo)
neededHashes                        = scriptsNeeded utxo txb
txdatsHashes                        = dom txdats
allOutHashes                        = getDataHashes (range txouts)
nativeScripts                       = mapPartial isInj₁ (txscripts tx utxo)
• ∀[ (vk , σ) ∈ vkSigs ] isSigned vk (txidBytes txid) σ
• ∀[ s ∈ nativeScripts ] (hash s ∈ neededHashes → validP1Script witsKeyHashes txvldt s)
• witsVKeyNeeded utxo txb ⊆ witsKeyHashes
• neededHashes \ refScriptHashes ≡ᵉ witsScriptHashes
• inputHashes ⊆ txdatsHashes
• txdatsHashes ⊆ inputHashes ∪ allOutHashes ∪ getDataHashes (range (utxo ∣ refInputs))
• languages tx utxo ⊆ allowedLanguages tx utxo
• txADhash ≡ map hash txAD
• Γ ⊢ s ⇀⦇ tx ,UTXO⦈ s'
  ────────────────────────────────
  Γ ⊢ s ⇀⦇ tx ,UTXOW⦈ s'
```

Figure 28: UTXOW inference rules

11.3 Plutus script context

CIP-69 unifies the arguments given to all types of Plutus scripts currently available (spending, certifying, rewarding, minting, voting, proposing).

The formal specification permits running spending scripts in the absence datums in the Conway era. However, since the interface with Plutus is kept abstract in this specification, changes to the representation of the script context which are part of CIP-69 are not included here. To provide a CIP-69-conformant implementation of Plutus to this specification, an additional step processing the `List Data` argument we provide would be required.

In Figure 28, the line `inputHashes ⊆ txdatsHashes` compares two inhabitants of `ℙ DataHash`. In the original Alonzo spec, these two terms would have inhabited `ℙ (Maybe DataHash)`, where a `nothing` is thrown out. In original spec, however, the right-hand side (`txdatsHashes`) could never contain `nothing`, hence the left-hand side (`inputHashes`) could never contain `nothing`.

12 Governance

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

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

Figure 30 shows some of the functions used to determine whether certain actions are enactable in a given state. Specifically, allEnactable passes the GovState to getAidPairsList to obtain a list of GovActionID-pairs which is then passed to enactable. The latter uses the _connects_to_ function to check whether the list of GovActionID-pairs connects the proposed action to a previously enacted one.

Additionally, govActionPriority assigns a priority to the various governance action types. This is useful for ordering lists of governance actions as well as grouping governance actions by constructor. In particular, the relations _~_ and _≈_ defined in Figure 30 are used for determining whether two actions are of the same "kind" in the following sense: either the actions arise from the same constructor, or one action is NoConfidence and the other is an UpdateCommittee action.

The GOV transition system is now given as the reflexive-transitive closure of the system GOV', described in Figure 31.

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 47. Note that there are no checks on whether the credential is actually associated with the role. This means that anyone can vote for, e.g., the CC role. However, during ratification those votes will only carry weight if they are properly associated with members of the constitutional committee.

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

- for ChangePParams or TreasuryWdrl, the proposal policy needs to be provided;
- for UpdateCommittee, 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.

Derived types

```
GovState = List (GovActionID × GovActionState)

record GovEnv : Type where
  txid        : TxId
  epoch       : Epoch
  pparams     : PParams
  ppolicy     : Maybe ScriptHash
  enactState  : EnactState
  certState   : CertState
```

Functions used in the GOV rules

```
govActionPriority : GovAction → ℕ
govActionPriority NoConfidence              = 0
govActionPriority (UpdateCommittee _ _ _) = 1
govActionPriority (NewConstitution _ _)   = 2
govActionPriority (TriggerHF _)            = 3
govActionPriority (ChangePParams _)        = 4
govActionPriority (TreasuryWdrl _)         = 5
govActionPriority Info                     = 6

_∼_ : ℕ → ℕ → Type
n ∼ m = (n ≡ m) ⊎ (n ≡ 0 × m ≡ 1) ⊎ (n ≡ 1 × m ≡ 0)

_≈ᵍ_ : GovAction → GovAction → Type
a ≈ᵍ a' = (govActionPriority a) ∼ (govActionPriority a')

insertGovAction : GovState → GovActionID × GovActionState → GovState
insertGovAction [] gaPr = [ gaPr ]
insertGovAction ((gaID₀ , gaSt₀) ∷ gaPrs) (gaID₁ , gaSt₁)
  = if (govActionPriority (action gaSt₀)) ≤? (govActionPriority (action gaSt₁))
    then (gaID₀ , gaSt₀) ∷ insertGovAction gaPrs (gaID₁ , gaSt₁)
    else (gaID₁ , gaSt₁) ∷ (gaID₀ , gaSt₀) ∷ gaPrs

mkGovStatePair : Epoch → GovActionID → RwdAddr → (a : GovAction) → NeedsHash a
                 → GovActionID × GovActionState
mkGovStatePair e aid addr a prev = (aid , record
  { votes = ∅ ; returnAddr = addr ; expiresIn = e ; action = a ; prevAction = prev })

addAction : GovState
          → Epoch → GovActionID → RwdAddr → (a : GovAction) → NeedsHash a
          → GovState
addAction s e aid addr a prev = insertGovAction s (mkGovStatePair e aid addr a prev)
addVote : GovState → GovActionID → Voter → Vote → GovState
addVote s aid voter v = map modifyVotes s
  where modifyVotes : GovActionID × GovActionState → GovActionID × GovActionState
        modifyVotes = λ (gid , s') → gid , record s'
          { votes = if gid ≡ aid then insert (votes s') voter v else votes s'}

isRegistered : GovEnv → Voter → Type
isRegistered ⟦ _ , _ , _ , _ , _ , ⟦ _ , pState , gState ⟧ᶜ ⟧ᵍ (r , c) = case r of λ where
  CC    → just c ∈ range (gState .ccHotKeys)
  DRep  → c ∈ dom (gState .dreps)
  SPO   → c ∈ map KeyHashObj (dom (pState .pools))

validHFAction : GovProposal → GovState → EnactState → Type
validHFAction (record { action = TriggerHF v ; prevAction = prev }) s e =
```

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

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

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

hasParent : EnactState → GovState → (a : GovAction) → NeedsHash a → Type
hasParent e s a aid with getHash aid
... | just aid' = hasParentE e aid' a
                  ⊎ Any (λ (gid , gas) → gid ≡ aid' × action gas ≈ᵍ a) s
... | nothing = ⊤
```

Figure 30: Enactability predicate

```
GOV-Vote : ∀ {x ast} → let
    open GovEnv Γ
    vote = record { gid = aid ; voter = voter ; vote = v ; anchor = x }
  in
  • (aid , ast) ∈ fromList s
  • canVote pparams (action ast) (proj₁ voter)
  • isRegistered Γ voter
    ────────────────────────────────
    (Γ , k) ⊢ s ⇀⦇ inj₁ vote ,GOV'⦈ addVote s aid voter v

GOV-Propose : ∀ {x} → let
    open GovEnv Γ; open PParams pparams hiding (a)
    prop = record { returnAddr = addr ; action = a ; anchor = x
                  ; policy = p ; deposit = d ; prevAction = prev }
    s' = addAction s (govActionLifetime +ᵉ epoch) (txid , k) addr a prev
  in
  • actionWellFormed a
  • d ≡ govActionDeposit
  • (∃[ u ] a ≡ ChangePParams u ⊎ ∃[ w ] a ≡ TreasuryWdrl w → p ≡ ppolicy)
  • (¬ (∃[ u ] a ≡ ChangePParams u ⊎ ∃[ w ] a ≡ TreasuryWdrl w) → p ≡ nothing)
  • (∀ {new rem q} → a ≡ UpdateCommittee new rem q
     → ∀[ e ∈ range new ] epoch < e × dom new ∩ rem ≡ᵉ ∅)
  • validHFAction prop s enactState
  • hasParent enactState s a prev
  • addr .RwdAddr.net ≡ NetworkId
    ────────────────────────────────
    (Γ , k) ⊢ s ⇀⦇ inj₂ prop ,GOV'⦈ s'

_⊢_⇀⦇_,GOV⦈_ = ReflexiveTransitiveClosureᵢ {sts = _⊢_⇀⦇_,GOV'⦈_}
```

Figure 31: Rules for the GOV transition system

13 Certificates

Derived types

```
data DepositPurpose : Type where
  CredentialDeposit : Credential  → DepositPurpose
  PoolDeposit       : KeyHash     → DepositPurpose
  DRepDeposit       : Credential  → DepositPurpose
  GovActionDeposit  : GovActionID → DepositPurpose

Deposits = DepositPurpose ⇀ Coin
```

Figure 32: Deposit types

```
record PoolParams : Type where
    rewardAddr : Credential
data DCert : Type where
  delegate   : Credential → Maybe VDeleg → Maybe KeyHash → Coin → DCert
  dereg      : Credential → Maybe Coin → DCert
  regpool    : KeyHash → PoolParams → DCert
  retirepool : KeyHash → Epoch → DCert
  regdrep    : Credential → Coin → Anchor → DCert
  deregdrep  : Credential → Coin → DCert
  ccreghot   : Credential → Maybe Credential → DCert
cwitness : DCert → Maybe Credential
cwitness (delegate c _ _ _) = just c
cwitness (dereg c _)         = just c
cwitness (regpool kh _)      = just $ KeyHashObj kh
cwitness (retirepool kh _)  = just $ KeyHashObj kh
cwitness (regdrep c _ _)     = just c
cwitness (deregdrep c _)     = just c
cwitness (ccreghot c _)      = just c
```

Figure 33: Delegation definitions

13.1 Removal of Pointer Addresses, Genesis Delegations and MIR Certificates

In the Conway era, support for pointer addresses, genesis delegations and MIR certificates is removed. In `DState`, this means that the four fields relating to those features are no longer present, and `DelegEnv` contains none of the fields it used to in the Shelley era.

Note that pointer addresses are still usable, only their staking functionality has been retired. So all funds locked behind pointer addresses are still accessible, they just don't count towards the stake distribution anymore. Genesis delegations and MIR certificates have been superceded by the new governance mechanisms, in particular the `TreasuryWdrl` governance action in case of the MIR certificates.

```
record CertEnv : Type where
  epoch     : Epoch
  pp        : PParams
  votes     : List GovVote
  wdrls     : RwdAddr ⇀ Coin
  coldCreds : ℙ Credential

record DState : Type where
  voteDelegs  : Credential ⇀ VDeleg
  stakeDelegs : Credential ⇀ KeyHash
  rewards     : Credential ⇀ Coin

record PState : Type where
  pools    : KeyHash ⇀ PoolParams
  retiring : KeyHash ⇀ Epoch

record GState : Type where
  dreps     : Credential ⇀ Epoch
  ccHotKeys : Credential ⇀ Maybe Credential

record CertState : Type where
  dState : DState
  pState : PState
  gState : GState

record DelegEnv : Type where
  pparams     : PParams
  pools       : KeyHash ⇀ PoolParams
  delegatees  : ℙ Credential

GovCertEnv = CertEnv
PoolEnv    = PParams
```

Figure 34: Types used for CERTS transition system

13.2 Explicit Deposits

Registration and deregistration of staking credentials are now required to explicitly state the deposit that is being paid or refunded. This aligns them better with other design decisions such as having explicit transaction fees and helps make this information visible to light clients and hardware wallets. While not shown in the figures, the old certificates without explicit deposits will still be supported for some time for backwards compatibility.

13.3 Delegation

Registered credentials can now delegate to a DRep as well as to a stake pool. This is achieved by giving the `delegate` certificate two optional fields, corresponding to a DRep and stake pool. Stake can be delegated for voting and block production simultaneously, since these are two separate features. In fact, preventing this could weaken the security of the chain, since security relies on high participation of honest stake holders.

13.4 Governance Certificate Rules

The rules for transition systems dealing with individual certificates are defined in Figures 36, 37 and 38. 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.[4] We check that the cold key did not previously resign from the committee. We allow this delegation for any cold credential that is either part of `EnactState` or is is a proposal. This allows a newly elected member of the constitutional committee to 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.

```
  _⊢_⇀⦇_,DELEG⦈_    : DelegEnv → DState → DCert → DState → Type
  _⊢_⇀⦇_,POOL⦈_     : PoolEnv → PState → DCert → PState → Type
  _⊢_⇀⦇_,GOVCERT⦈_  : GovCertEnv → GState → DCert → GState → Type
  _⊢_⇀⦇_,CERT⦈_     : CertEnv → CertState → DCert → CertState → Type
  _⊢_⇀⦇_,CERTBASE⦈_ : CertEnv → CertState → ⊤ → CertState → Type

_⊢_⇀⦇_,CERTS⦈_ : CertEnv → CertState → List DCert → CertState → Type
_⊢_⇀⦇_,CERTS⦈_ = ReflexiveTransitiveClosureᵇ' {_⊢_⇀⟦_⟧ᵇ_ = _⊢_⇀⦇_,CERTBASE⦈_} {_⊢_⇀⦇_,CERT⦈_}
```

Figure 35: Types for the transition systems relating to certificates

Figure 39 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.

[4] By "hot" and "cold" credentials we mean the following: a cold credential is used to register a hot credential, and then the hot credential is used for voting. The idea is that the access to the cold credential is kept in a secure location, while the hot credential is more conveniently accessed. If the hot credential is compromised, it can be changed using the cold credential.

```
DELEG-delegate : let open PParams pp in
  • (c ∉ dom rwds → d ≡ keyDeposit)
  • (c ∈ dom rwds → d ≡ 0)
  • mv ∈ map (just ∘ credVoter DRep) delegatees ∪
      fromList ( nothing ∷ just abstainRep ∷ just noConfidenceRep ∷ [] )
  • mkh ∈ map just (dom pools) ∪ ❴ nothing ❵
    ────────────────────────────────
  ⟦ pp , pools , delegatees ⟧ ⊢ ⟦ vDelegs , sDelegs , rwds ⟧
  ⇀⦇ delegate c mv mkh d ,DELEG⦈
  ⟦ insertIfJust c mv vDelegs , insertIfJust c mkh sDelegs , rwds ∪ˡ ❴ c , 0 ❵ ⟧

DELEG-dereg :
  • (c , 0) ∈ rwds
    ────────────────────────────────
  ⟦ pp , pools , delegatees ⟧ ⊢ ⟦ vDelegs , sDelegs , rwds ⟧ ⇀⦇ dereg c md ,DELEG⦈ ⟦ vDelegs ∣ ❴ c ❵ ᶜ , sDelegs ∣ ❴ c ❵ ᶜ , rwds ∣ ❴ c ❵ ᶜ ⟧
  ⟦ pp , pools , delegatees ⟧ ⊢ ⟦ vDelegs , sDelegs , rwds ⟧ ⇀⦇ reg c d ,DELEG⦈ ⟦ vDelegs , sDelegs , rwds ∪ˡ ❴ c , 0 ❵ ⟧
```

Figure 36: Auxiliary DELEG transition system

```
POOL-regpool :
  • kh ∉ dom pools
    ────────────────────────────────
  pp ⊢ ⟦ pools , retiring ⟧ ⇀⦇ regpool kh poolParams ,POOL⦈ ⟦ ❴ kh , poolParams ❵ ∪ˡ pools , retiring ⟧

POOL-retirepool :
    ────────────────────────────────
  pp ⊢ ⟦ pools , retiring ⟧ ⇀⦇ retirepool kh e ,POOL⦈ ⟦ pools , ❴ kh , e ❵ ∪ˡ retiring ⟧
```

Figure 37: Auxiliary POOL transition system

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

GOVCERT-deregdrep :
  • c ∈ dom dReps
  ────────────────────────────────
  ⟦ e , pp , vs , wdrls , cc ⟧ ⊢ ⟦ dReps , ccKeys ⟧ ⇀⦇ deregdrep c d ,GOVCERT⦈ ⟦ dReps ∣ ❴ c ❵ ᶜ , ccKeys ⟧

GOVCERT-ccreghot :
  • (c , nothing) ∉ ccKeys
  • c ∈ cc
  ────────────────────────────────
  ⟦ e , pp , vs , wdrls , cc ⟧ ⊢ ⟦ dReps , ccKeys ⟧ ⇀⦇ ccreghot c mc ,GOVCERT⦈ ⟦ dReps , ❴ c , mc ❵ ∪ˡ ccKeys ⟧
```

Figure 38: Auxiliary GOVCERT transition system

CERT transitions

```
CERT-deleg :
  • ⟦ pp , PState.pools stᵖ , dom (GState.dreps stᵍ) ⟧ ⊢ stᵈ ⇀⦇ dCert ,DELEG⦈ stᵈ'
    ────────────────────────────────
    ⟦ e , pp , vs , wdrls , cc ⟧ ⊢ ⟦ stᵈ , stᵖ , stᵍ ⟧ ⇀⦇ dCert ,CERT⦈ ⟦ stᵈ' , stᵖ , stᵍ ⟧

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

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

CERTBASE transition

```
CERT-base : let
    open PParams pp
    refresh          = mapPartial getDRepVote (fromList vs)
    refreshedDReps   = mapValueRestricted (const (e + drepActivity)) dReps refresh
    wdrlCreds        = map stake (dom wdrls)
    validVoteDelegs  = voteDelegs ∣^ ( map (credVoter DRep) (dom dReps)
                                      ∪ fromList (noConfidenceRep ∷ abstainRep ∷ []) )
  in
  • filter isKeyHash wdrlCreds ⊆ dom voteDelegs
  • map (map₁ stake) (wdrls ) ⊆ rewards
    ────────────────────────────────
    ⟦ e , pp , vs , wdrls , cc ⟧ ⊢ ⟦ ⟦ voteDelegs , stakeDelegs , rewards ⟧ , stᵖ , ⟦ dReps , ccHotKeys ⟧ ⟧
      ⇀⦇ _ ,CERTBASE⦈
    ⟦ ⟦ validVoteDelegs , stakeDelegs , constMap wdrlCreds 0 ∪ˡ rewards ⟧ , stᵖ , ⟦ refreshedDReps , ccHotKeys ⟧ ⟧
```

Figure 39: 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 : Type where
    slot        : Slot
    ppolicy     : Maybe ScriptHash
    pparams     : PParams
    enactState  : EnactState
    treasury    : Coin

record LState : Type where
    utxoSt    : UTxOState
    govSt     : GovState
    certState : CertState

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

isUnregisteredDRep : CertState → Voter → Type
isUnregisteredDRep ⟦ _ , _ , gState ⟧ (r , c) = r ≡ DRep × c ∉ dom (gState .dreps)

removeOrphanDRepVotes : CertState → GovActionState → GovActionState
removeOrphanDRepVotes certState gas = record gas { votes = votes′ }
  where
    votes′ = filterKeys (¬_ ∘ isUnregisteredDRep certState) (votes gas)

_|ᵒ_ : GovState → CertState → GovState
govSt |ᵒ certState = L.map (map₂ (removeOrphanDRepVotes certState)) govSt
allColdCreds : GovState → EnactState → ℙ Credential
allColdCreds govSt es =
  ccCreds (es .cc) ∪ concatMap (λ (_ , st) → proposedCC (st .action)) (fromList govSt)
```

Figure 40: Types and functions for the LEDGER transition system

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

Figure 41: 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 , allColdCreds govSt enactState ⟧ ⊢ certState ⇀⦇ txcerts ,CERTS⦈ certState'
  • ⟦ txid , epoch slot , pparams , ppolicy , enactState , certState' ⟧ ⊢ govSt |ᵒ certState' ⇀⦇ txgov txb ,GOV⦈ govSt'
  ────────────────────────────────
  Γ ⊢ s ⇀⦇ tx ,LEDGER⦈ ⟦ utxoSt' , govSt' , certState' ⟧

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 ⟧
```

Figure 42: LEDGER transition system

```
_⊢_⇀⦇_,LEDGERS⦈_ : LEnv → LState → List Tx → LState → Type
_⊢_⇀⦇_,LEDGERS⦈_ = ReflexiveTransitiveClosure {sts = _⊢_⇀⦇_,LEDGER⦈_}
```

Figure 43: LEDGERS transition system

15 Enactment

Figure 44 contains some definitions required to define the ENACT transition system. EnactEnv is the environment and EnactState the state of ENACT, which enacts a governance action. All governance actions except TreasuryWdrl and Info modify EnactState permanently, which of course can have further consequences. TreasuryWdrl accumulates withdrawal temporarily in EnactState, but this information is applied and discarded immediately in EPOCH. Also, enacting these governance actions is the *only* way of modifying EnactState. The withdrawals field of EnactState is special in that it is ephemeral—ENACT accumulates withdrawals there which are paid out at the next epoch boundary where this field will be reset.

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

```
record EnactEnv : Type where
    gid      : GovActionID
    treasury : Coin
    epoch    : Epoch

record EnactState : Type where
    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 {UpdateCommittee _ _ _} h = just h
getHash {NewConstitution _ _}   h = just h
getHash {TriggerHF _}            h = just h
getHash {ChangePParams _}        h = just h
getHash {TreasuryWdrl _}         _ = nothing
getHash {Info}                   _ = nothing

getHashES : EnactState → GovAction → Maybe GovActionID
getHashES es NoConfidence              = just (es .cc .proj₂)
getHashES es (UpdateCommittee _ _ _) = 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
```

Type of the ENACT transition system

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

Figure 44: Types and function used for the ENACT transition system

Figures 45 and 46 define 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 `UpdateCommittee` and `TreasuryWdrl`:

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

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

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

Figure 45: ENACT transition system

```
Enact-HF :

    ──────────────────────────────────────────
⎛ gid ⎞
⎜  t  ⎟ ⊢ s ⇀⦇ TriggerHF v ,ENACT⦈ record s { pv = v , gid }
⎝  e  ⎠

Enact-PParams :

    ──────────────────────────────────────────
⎛ gid ⎞
⎜  t  ⎟ ⊢ s ⇀⦇ ChangePParams up ,ENACT⦈
⎝  e  ⎠

record s { pparams = applyUpdate (s .pparams .proj₁) up , gid }

Enact-Wdrl : let newWdrls = s .withdrawals ∪⁺ wdrl in
    ∑[ x ← newWdrls ] x ≤ t
    ──────────────────────────────────────────
⎛ gid ⎞
⎜  t  ⎟ ⊢ s ⇀⦇ TreasuryWdrl wdrl ,ENACT⦈ record s { withdrawals = newWdrls }
⎝  e  ⎠

Enact-Info :

    ──────────────────────────────────────────
⎛ gid ⎞
⎜  t  ⎟ ⊢ s ⇀⦇ Info ,ENACT⦈ s
⎝  e  ⎠
```

Figure 46: ENACT transition system (continued)

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 47 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: For an action to pass, the stake associated with the yes votes must exceed 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 `UpdateCommittee` and `ChangePParams`.

For `UpdateCommittee`, 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.

Finally, each of the $\mathtt{P}_x$ and $\mathtt{Q}_x$ in Figure 47 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 ∣
  (UpdateCommittee _ _ _) → ∣ ─  ∥ 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 (map (proj₁ ∘ pparamThreshold) (updateGroups ppu))
             , maxThreshold (map (proj₂ ∘ pparamThreshold) (updateGroups ppu))

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

Figure 47: 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 48 defines some types and functions used in the RATIFY transition system. CCData is simply an alias to define some functions more easily.

Figure 49 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 : Type where
  stakeDistr : VDeleg ⇀ Coin

record RatifyEnv : Type where
  stakeDistrs  : StakeDistrs
  currentEpoch : Epoch
  dreps        : Credential ⇀ Epoch
  ccHotKeys    : Credential ⇀ Maybe Credential
  treasury     : Coin
  pools        : KeyHash ⇀ PoolParams
  delegatees   : Credential ⇀ VDeleg

record RatifyState : Type where
  es      : EnactState
  removed : ℙ (GovActionID × GovActionState)
  delay   : Bool

CCData : Type
CCData = Maybe ((Credential ⇀ Epoch) × ℚ)

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

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

Figure 48: 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.

Let us discuss the last item above—the way SPO votes are counted—as the ledger specification's handling of this has evolved in response to community feedback. Previously, if an SPO did not vote, then it would be counted as having voted `abstain` by default. Members of the SPO community found this behavior counterintuitive and requested that non-voters be assigned a `no` vote by default, with the caveat that an SPO could change its default setting by delegating its reward account credential to an `AlwaysNoConfidence` DRep or an `AlwaysAbstain` DRep. (This change applies only after the bootstrap period; during the bootstrap period the logic is unchanged; see Appendix Section C.) To be precise, the agreed upon specification is the following: an SPO that did not vote is assumed to have vote `no`, except under the following circumstances:

- if the SPO has delegated its reward credential to an `AlwaysNoConfidence` DRep, then their default vote is `yes` for `NoConfidence` proposals and `no` for other proposals;

- if the SPO has delegated its reward credential to an `AlwaysAbstain` DRep, then its default vote is `abstain` for all proposals.

It is important to note that the credential that can now be used to set a default voting behavior is the credential used to withdraw staking rewards, which is not (in general) the same as the credential used for voting.

Figure 50 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.

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

- `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 51 defines functions that deal with delays and the acceptance criterion for ratification. 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 52.

- `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.

```
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 = filter 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

  SPODefaultVote : GovAction → VDeleg → Vote
  SPODefaultVote ga (credVoter SPO (KeyHashObj kh)) = case lookupᵐ? pools kh of
       nothing → Vote.no
       (just p) → case lookupᵐ? delegatees (PoolParams.rewardAddr p) , ga of
             (_                    , TriggerHF _  ) → Vote.no
             (just noConfidenceRep , NoConfidence ) → Vote.yes
             (just abstainRep      , _            ) → Vote.abstain
             _                                      → Vote.no
  SPODefaultVote _ _ = Vote.no

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

  actualCCVotes : Credential ⇀ Vote
  actualCCVotes = case cc of
       nothing         → ∅
       (just (m , q)) → if ccMinSize ≤ length (mapFromPartialFun getCCHotCred (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 (map (credVoter DRep) activeDReps) Vote.no

  actualSPOVotes : GovAction → VDeleg ⇀ Vote
  actualSPOVotes a = roleVotes SPO ∪ˡ mapFromFun (SPODefaultVote a) spos
```

Figure 49: Vote counting

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

acceptedStakeRatio : GovRole → ℙ VDeleg → StakeDistrs → (VDeleg ⇀ Vote) → ℚ
acceptedStakeRatio r cc dists votes = acceptedStake /₀ totalStake
  where
    dist : VDeleg ⇀ Coin
    dist = getStakeDist r cc dists
    acceptedStake totalStake : Coin
    acceptedStake = ∑[ x ← dist ∣ votes ⁻¹ Vote.yes ] x
    totalStake    = ∑[ x ← dist ∣ dom (votes ∣^ (❴ Vote.yes ❵ ∪ ❴ Vote.no ❵)) ] x

acceptedBy : RatifyEnv → EnactState → GovActionState → GovRole → Type
acceptedBy Γ (record { cc = cc , _; pparams = pparams , _ }) gs role =
  let open GovActionState gs; open PParams pparams
      votes' = actualVotes Γ pparams cc action votes
      mbyT   = threshold pparams (proj₂ <$> cc) action role
      t      = maybe id 0ℚ mbyT
  in acceptedStakeRatio role (dom votes') (stakeDistrs Γ) votes' ≥ t
    ∧ (role ≡ CC → maybe (λ (m , _) → length m) 0 cc ≥ ccMinSize ⊎ Is-nothing mbyT)

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

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

Figure 50: Functions used in RATIFY rules, without delay

```
verifyPrev : (a : GovAction) → NeedsHash a → EnactState → Type
verifyPrev NoConfidence            h es = h ≡ es .cc .proj₂
verifyPrev (UpdateCommittee _ _ _) 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 (UpdateCommittee _ _ _) = true
delayingAction (NewConstitution _ _)   = true
delayingAction (TriggerHF _)           = true
delayingAction (ChangePParams _)       = false
delayingAction (TreasuryWdrl _)        = false
delayingAction Info                    = false

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

acceptConds : RatifyEnv → RatifyState → GovActionID × GovActionState → Type

               ⎛   es    ⎞
acceptConds Γ  ⎜ removed ⎟ (id , st) = let open RatifyEnv Γ; open GovActionState st in
               ⎝   d     ⎠


    accepted Γ es st
  × ¬ delayed action prevAction es d

          ⎛      id      ⎞
× ∃[ es' ] ⎜   treasury   ⎟ ⊢ es ⇀⦇ action ,ENACT⦈ es'
          ⎝ currentEpoch ⎠
```

Figure 51: Functions related to ratification

```
RATIFY-Accept : ∀ {Γ} {es} {removed} {d} {a} {es'} → let open RatifyEnv Γ; st = a .proj₂; open GovAction

  • acceptConds Γ ⎛ es      ⎞ a
                  ⎜ removed ⎟
                  ⎝ d       ⎠
  • ⎛ a .proj₁     ⎞ ⊢ es ⇀⦇ action ,ENACT⦈ es'
    ⎜ treasury     ⎟
    ⎝ currentEpoch ⎠

  ──────────────────────────────
Γ ⊢ ⎛ es      ⎞ ⇀⦇ a ,RATIFY'⦈ ⎛ es'                    ⎞
    ⎜ removed ⎟                ⎜ ❴ a ❵ ∪ removed        ⎟
    ⎝ d       ⎠                ⎝ delayingAction action  ⎠

RATIFY-Reject : ∀ {Γ} {es} {removed} {d} {a} → let open RatifyEnv Γ; st = a .proj₂ in

  • ¬ acceptConds Γ ⎛ es      ⎞ a
                    ⎜ removed ⎟
                    ⎝ d       ⎠

  • expired currentEpoch st
  ──────────────────────────────
Γ ⊢ ⎛ es      ⎞ ⇀⦇ a ,RATIFY'⦈ ⎛ es              ⎞
    ⎜ removed ⎟                ⎜ ❴ a ❵ ∪ removed ⎟
    ⎝ d       ⎠                ⎝ d               ⎠

RATIFY-Continue : ∀ {Γ} {es} {removed} {d} {a} → let open RatifyEnv Γ; st = a .proj₂ in

  • ¬ acceptConds Γ ⎛ es      ⎞ a
                    ⎜ removed ⎟
                    ⎝ d       ⎠

  • ¬ expired currentEpoch st
  ──────────────────────────────
Γ ⊢ ⎛ es      ⎞ ⇀⦇ a ,RATIFY'⦈ ⎛ es      ⎞
    ⎜ removed ⎟                ⎜ removed ⎟
    ⎝ d       ⎠                ⎝ d       ⎠

_⊢_⇀⦇_,RATIFY⦈_ : RatifyEnv → RatifyState → List (GovActionID × GovActionState)
                → RatifyState → Type
_⊢_⇀⦇_,RATIFY⦈_ = ReflexiveTransitiveClosure {sts = _⊢_⇀⦇_,RATIFY'⦈_}
```

Figure 52: The RATIFY transition system

17 Epoch Boundary

```
record RewardUpdate : Set where
  constructor ⟦_,_,_,_⟧ʳᵘ
  field
    Δt Δr Δf : ℤ
    rs : Credential ⇀ Coin
```

```
record Snapshot : Set where
  constructor ⟦_,_⟧
  field
    stake       : Credential ⇀ Coin
    delegations : Credential ⇀ KeyHash
    -- poolParameters : KeyHash ⇀ PoolParam

record Snapshots : Set where
  constructor ⟦_,_,_,_⟧
  field
    mark set go  : Snapshot
    feeSS        : Coin

record EpochState : Type where
    acnt : Acnt
    ss   : Snapshots
    ls   : LState
    es   : EnactState
    fut  : RatifyState
record NewEpochState : Type where
    lastEpoch  : Epoch
    epochState : EpochState
    ru         : Maybe RewardUpdate
```

Figure 53: Definitions for the EPOCH and NEWEPOCH transition systems

```
applyRUpd : RewardUpdate → EpochState → EpochState
```

```
applyRUpd ⟦ Δt , Δr , Δf , rs ⟧
  ⟦ ⟦ treasury , reserves ⟧ , ss ,
    ⟦ ⟦ utxo , fees , deposits , donations ⟧ , govSt ,
      ⟦ ⟦ voteDelegs , stakeDelegs , rewards ⟧ , pState , gState ⟧ ⟧ , es , fut ⟧
  = ⟦ ⟦ posPart (ℤ.+ treasury ℤ.+ Δt ℤ.+ ℤ.+ unregRU')
      , posPart (ℤ.+ reserves ℤ.+ Δr) ⟧ , ss ,
      ⟦ ⟦ utxo , posPart (ℤ.+ fees ℤ.+ Δf) , deposits , donations ⟧ , govSt ,
        ⟦ ⟦ voteDelegs , stakeDelegs , rewards ∪⁺ regRU ⟧ , pState , gState ⟧ ⟧ , es , fut ⟧
```

```
  where
    regRU     = rs ∣ dom rewards
    unregRU   = rs ∣ dom rewards ᶜ
    unregRU' = ∑[ x ← unregRU ] x

getOrphans : EnactState → GovState → GovState
getOrphans es govSt = proj₁ $ iterate step ([] , govSt) (length govSt)
  where
    step : GovState × GovState → GovState × GovState
    step (orps , govSt) =
      let
        isOrphan? a prev = ¬? (hasParent? es govSt a prev)
        (orps' , govSt') = partition
          (λ (_ , record {action = a ; prevAction = prev}) → isOrphan? a prev) govSt
      in
        (orps ++ orps' , govSt')
```

```
data _⊢_⇀⦇_,SNAP⦈_ : LState → Snapshots → ⊤ → Snapshots → Type where
  SNAP : let open LState lstate; open UTxOState utxoSt; open CertState certState
             stake = stakeDistr utxo dState pState
    in
    lstate ⊢ ⟦ mark , set , go , feeSS ⟧ ⇀⦇ tt ,SNAP⦈ ⟦ stake , mark , set , fees ⟧
```

```
data _⊢_⇀⦇_,EPOCH⦈_ : ⊤ → EpochState → Epoch → EpochState → Type where
```

Figure 55 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 by carrying out each of the following tasks.

```
stakeDistr : UTxO → DState → PState → Snapshot
stakeDistr utxo ⟦ _ , stakeDelegs , rewards ⟧ pState = ⟦ aggregate₊ (stakeRelation ᶠ) , stakeDelegs ⟧
  where
    m = map (λ a → (a , cbalance (utxo ∣^' λ i → getStakeCred i ≡ just a))) (dom rewards)
    stakeRelation = m ∪ proj₁ rewards

gaDepositStake : GovState → Deposits → Credential ⇀ Coin
gaDepositStake govSt ds = aggregateBy
  (map (λ (gaid , addr) → (gaid , addr) , stake addr) govSt')
  (mapFromPartialFun (λ (gaid , _) → lookupᵐ? ds (GovActionDeposit gaid)) govSt')
  where govSt' = map (map₂ returnAddr) (fromList govSt)

mkStakeDistrs : Snapshot → GovState → Deposits → (Credential ⇀ VDeleg) → StakeDistrs
mkStakeDistrs ⟦ stake , _ ⟧ govSt ds delegations .StakeDistrs.stakeDistr =
  aggregateBy (proj₁ delegations) (stake ∪⁺ gaDepositStake govSt ds)
```

Figure 54: Functions for computing stake distributions

- 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
( esW  removed  _ )ᵀ = fut ; ( utxoSt  govSt  ( dState
                                                  pState
                                                  gState ) )ᵀ = ls

    es         = record esW { withdrawals = ∅ }
    tmpGovSt = filter (λ x → ¿ proj₁ x ∉ map proj₁ removed ¿) govSt
    orphans   = fromList $ getOrphans es tmpGovSt
    removed' = removed ∪ orphans
    removedGovActions = flip concatMap removed' λ (gaid , gaSt) →
      map (returnAddr gaSt ,_) ((utxoSt .deposits ∣ ❴ GovActionDeposit gaid ❵) )
    govActionReturns = aggregate₊ (map (λ (a , _ , d) → a , d) removedGovActions ᶠ)

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

    retired   = (pState .retiring) ⁻¹ e
    payout    = govActionReturns ∪⁺ trWithdrawals
    refunds   = pullbackMap payout toRwdAddr (dom (dState .rewards))
    unclaimed = getCoin payout - getCoin refunds

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

    certState' =
(               record dState { rewards = dState .rewards ∪⁺ refunds }
                         (  (pState .pools) ∣ retired ᶜ     )
                         ( (pState .retiring) ∣ retired ᶜ   )
 ( if null govSt' then mapValues (1 +_) (gState .dreps) else (gState .dreps) )
                     (gState .ccHotKeys) ∣ ccCreds (es .cc)              )
                                   utxoSt .utxo
                                   utxoSt .fees
utxoSt' = (  utxoSt .deposits ∣ map (proj₁ ∘ proj₂) removedGovActions ᶜ  )
                                       0

    acnt' = record acnt
      { treasury = acnt .treasury ∸ totWithdrawals + utxoSt .donations + unclaimed }
  in
  record { currentEpoch = e
         ; stakeDistrs = mkStakeDistrs (Snapshots.mark ss') govSt'
                                         (utxoSt' .deposits) (voteDelegs dState)
         ; treasury = acnt .treasury ; GState gState
         ; pools = pState .pools ; delegatees = dState .voteDelegs }
⊢ ( es  ∅  false )ᵀ ⇀⦇ govSt' ,RATIFY⦈ fut'

    → ls ⊢ ss ⇀⦇ tt ,SNAP⦈ ss'
  ────────────────────────────────
                                     acnt'
                                      ss'
   ( acnt )                      ( utxoSt'  )
   (  ss  )                      ( govSt'   )
_ ⊢ (  ls  ) ⇀⦇ e ,EPOCH⦈ (    ( certState' ) )
   ( es₀  )                          es
   ( fut  )                         fut'
```


Figure 55: EPOCH transition system

```
_⊢_⇀⦇_,NEWEPOCH⦈_ : ⊤ → NewEpochState → Epoch → NewEpochState → Type

NEWEPOCH-New : let
    eps' = applyRUpd ru eps
  in
  • e ≡ lastEpoch + 1
  • _ ⊢ eps' ⇀⦇ e ,EPOCH⦈ eps''
    ────────────────────────────────
    _ ⊢ ⟦ lastEpoch , eps , just ru ⟧ ⇀⦇ e ,NEWEPOCH⦈ ⟦ e , eps'' , nothing ⟧

NEWEPOCH-Not-New :
  • e ≢ lastEpoch + 1
    ────────────────────────────────
    _ ⊢ ⟦ lastEpoch , eps , mru ⟧ ⇀⦇ e ,NEWEPOCH⦈ ⟦ lastEpoch , eps , mru ⟧

NEWEPOCH-No-Reward-Update :
  • e ≡ lastEpoch + 1
  • _ ⊢ eps ⇀⦇ e ,EPOCH⦈ eps'
    ────────────────────────────────
    _ ⊢ ⟦ lastEpoch , eps , nothing ⟧ ⇀⦇ e ,NEWEPOCH⦈ ⟦ e , eps' , nothing ⟧
```

Figure 56: NEWEPOCH transition system

18 Blockchain Layer

```
record ChainState : Type where
  newEpochState : NewEpochState

record Block : Type where
  ts   : List Tx
  slot : Slot
```

Figure 57: Definitions CHAIN transition system

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

Figure 58: Type of the CHAIN transition system

```
CHAIN :
totalRefScriptsSize ls ts ≤ maxRefScriptSizePerBlock
→ _ ⊢ newEpochState ⇀⦇ epoch slot ,NEWEPOCH⦈ nes
  ⎛              slot              ⎞
  ⎜ constitution .proj₁ .proj₂     ⎟
→ ⎜               pp               ⎟ ⊢ ls ⇀⦇ ts ,LEDGERS⦈ ls'
  ⎜               es               ⎟
  ⎝      Acnt.treasury acnt        ⎠

────────────────────────────────────
_ ⊢ s ⇀⦇ b ,CHAIN⦈ record s { newEpochState =
                               record nes { epochState =
                                             record epochState { ls = ls'} } }
```

Figure 59: 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 60: Computing the UTXO transition system

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

```
txid ∉ map proj₁ (dom utxo)
```

and

```
Γ ⊢ ⟦ utxo  , fees  , deposits  , donations  ⟧ᵘ ⇀⦇ tx ,UTXO⦈
    ⟦ utxo' , fees' , deposits' , donations' ⟧ᵘ
```

then

```
getCoin ⟦ utxo , fees , deposits , donations ⟧ᵘ + φ(getCoin txwdrls , isValid)
≡ 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, A. Knispel, M. Benkort, K. Hammond, C. Hoskinson, and S. Leathers. A first step towards on-chain decentralized governance. https://cips.cardano.org/cip/CIP-1694, 2023.

[3] J. Corduan, P. Vinogradova, and M. Güdemann. A Formal Specification of the Cardano Ledger. https://github.com/intersectmbo/cardano-ledger/releases/latest/download/shelley-ledger.pdf, 2019. Accessed: 2024-07-30.

[4] B. Nordström, K. Petersson, and J. M. Smith. Programming in Martin-Löf's type theory: An introduction. https://www.cse.chalmers.se/research/group/logic/book/book.pdf, July 1990. Previously published as [5].

[5] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*. International series of monographs on computer science. Clarendon Press; Oxford University Press, July 1990.

A 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 : Type) : Type 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.

C Bootstrapping the Governance System

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

- Transactions containing any proposal except `TriggerHF`, `ChangePParams` or `Info` will be rejected.
- Transactions containing a vote other than a `CC` vote, a `SPO` vote on a `TriggerHF` action or any vote on an `Info` action will be rejected.
- `Q4`, `P5` and `Q5e` are set to 0.
- An SPO that does not vote is assumed to have voted `abstain`.

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