Formal Specification of the Cardano Ledger for the Conway era

Andre Knispel
andre.knispel@iohk.io

William DeMeo
william.demeo@iohk.io

Joosep Jääger
joosep.jaager@iohk.io

Carlos Tomé Cortiñas
carlos.tome-cortinas@iohk.io

Abstract

This document presents the modifications to the previous specifications of the Cardano ledger (see [1], [2], [3], [4]) for the Conway era. The additions mostly relate to the implementation of the governance framework described in CIP-1694 [5].

List of Contributors

Alasdair Hill, Ulf Norell, Orestis Melkonian, Jared Corduan, Alexey Kuleshevich

Contents

1 Introduction

This is the specification of the Conway era of the Cardano ledger. As with previous specifications, this document is an incremental specification, so everything that isn't defined here refers to the most recent definition from an older specification.

1.1 A Note on Agda

This specification is written using the Agda programming language and proof assistant [6]. We have made a considerable effort to ensure that this document is readable by 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 Secs. A and 2, but there is no guarantee that those sections are complete. If the meaning of an expression is confusing or unclear, please open an issue in the formal ledger repository with the 'notation' label.

1.2 Separation of Concerns

The *Cardano Node* consists of three pieces,

- a *networking layer* responsible for sending messages across the internet,
- a *consensus layer* establishing a common order of valid blocks, and
- a *ledger layer* which determines whether a sequence of blocks is valid.

Because of this separation, the ledger can be modeled as 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 roughly 25 (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 state transition rules need to be applied as many times as possible to arrive at a final state. Since we use this pattern multiple times, we define a closure operation which takes a transition rule and applies it as many times as possible.

The closure $\_\vdash\_\rightharpoonup⟦\_⟧*\_$ of a relation $\_\vdash\_\rightharpoonup⟦\_⟧\_$ is defined in Fig. 1. In the remainder of the text, the closure operation is called `ReflexiveTransitiveClosure`.

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 Fig. 2 which is parametrized over the state transition relation.

```
Closure type

  data _⊢_⇀⟦_⟧*_ : C → S → List Sig → S → Type where

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

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

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

Figure 2: Computational relations

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

This has two further implications:

- Since `compute` is a function, the 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.

When we need to convert a list `l` to its set of elements, we write `fromList l`.

```
_⊆_ : {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 × B → Type` or `A → B → Type`.

These relations are typically required to be decidable, which means that there is a boolean-valued function that computes whether the predicate holds or not. This means that it is generally safe to think of predicates simply returning a boolean value instead.

2 Notation

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

Propositions, sets and types. See Sec. 1.5. Note that 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` and use ℙ for our set type.

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 ⊎ B`, and their product is denoted by `A × B`. The projection functions from products are denoted `proj₁` and `proj₂`, and the injections are denoted `inj₁` and `inj₂` respectively. The properties whether an element of a coproduct is in the left or right component are called `isInj₁` and `isInj₂`.

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 Sec. A.

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

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

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

Left-biased union. For maps m and m', we write $m \cup^{l} m'$ 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 $\{f\ x \mid x \in S \text{ and } f \text{ is defined at } x\}$.

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 symbol `~` denotes (pseudo)equality of two values x and y of type `Maybe X`: if x is of the form `just` x' and y is of the form `just` y', then x' and y' have to be equal. Otherwise, they are considered "equal".

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.

2.1 Superscripts and Other Special Notations

In the current version of this specification, superscript letters are sometimes 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]

[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.

3 Protocol Parameters

This section is part of the `Ledger.Conway.PParams` module of the formal ledger specification., in which we define the adjustable protocol parameters of the Cardano ledger.

Protocol 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.

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

Figure 3: Protocol parameter group definition

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

record PoolThresholds : Type where
  field
    Q1 Q2a Q2b Q4 Q5 : ℚ
```

Figure 4: Protocol parameter threshold definitions

`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 purpose of these groups is to determine voting thresholds for proposals aiming to change parameters. Given a proposal to change a certain set of parameters, we look at which groups those parameters fall into and from this we determine the voting threshold for that proposal. (The voting threshold calculation is described in detail in Sec. 12.1; in particular, the definition of the `threshold` function appears in Fig. 42.)

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 CIP-1694 there is no `SecurityGroup`, but there is the concept of security-relevant protocol parameters (see Corduan et al. [5]). The difference between these notions is only social, so we implement security-relevant protocol parameters as a group.

The new protocol parameters are declared in Fig. 5 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 Q5;
- 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.

Fig. 5 also defines the function paramsWellFormed which performs some sanity checks on protocol parameters. Fig. 7 defines types and functions to update parameters. These consist of an abstract type UpdateT and two functions applyUpdate and updateGroups. The type UpdateT is to be instantiated by a type that

- can be used to update parameters, via the function applyUpdate
- can be queried about what parameter groups it updates, via the function updateGroups

An element of the type UpdateT is well formed if it updates at least one group and applying the update preserves well-formedness.

```
record PParams : Type where
  field
```

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
    monetaryExpansion           : UnitInterval -- formerly: rho
    treasuryCut                 : UnitInterval -- formerly: tau
    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
```

Security group

maxBlockSize maxTxSize maxHeaderSize maxValSize maxBlockExUnits a b minFeeRefScriptCoinsPerByte coinsPerUTxOByte govActionDeposit

Figure 5: Protocol parameter definitions

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

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

Figure 6: 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 7: Abstract type for parameter updates

```
scriptsCost : (pp : PParams) → ℕ → Coin
scriptsCost pp scriptSize
  = scriptsCostAux 0ℚ minFeeRefScriptCoinsPerByte scriptSize
  where
    minFeeRefScriptCoinsPerByte = PParams.minFeeRefScriptCoinsPerByte pp
    refScriptCostMultiplier = PParams.refScriptCostMultiplier pp
    refScriptCostStride = PParams.refScriptCostStride pp
    scriptsCostAux : ℚ       -- accumulator
                   → ℚ       -- current tier price
                   → (n : ℕ) -- remaining script size
                   → Coin
    scriptsCostAux acl curTierPrice n
 = case n ≤? fromℕ⁺ refScriptCostStride of
 (yes _) → ∣ floor (acl + (fromℕ n * curTierPrice)) ∣
 (no p)  → scriptsCostAux
             (acl + (fromℕ (fromℕ⁺ refScriptCostStride) * curTierPrice))
             (refScriptCostMultiplier * curTierPrice)
             (n - fromℕ⁺ refScriptCostStride)
```

Figure 8: Calculation of fees for reference scripts

4 Fee Calculation

This section is part of the `Ledger.Conway.Fees` module of the formal ledger specification., where we define the functions used to compute the fees associated with reference scripts.

The function `scriptsCost` (Fig. 8) calculates the fee for reference scripts in a transaction. It takes as input the total size of the reference scripts in bytes—which can be calculated using `refScriptsSize` (Fig. 17)—and uses a function (`scriptsCostAux`) that is piece-wise linear in the size, where the linear constant multiple grows with each `refScriptCostStride` bytes. In addition, `scriptsCost` depends on the following constants (which are bundled with the protocol parameters; see Fig. 5):

- `refScriptCostMultiplier`, a rational number, the growth factor or step multiplier that determines how much the price per byte increases after each increment;
- `refScriptCostStride`, an integer, the size in bytes at which the price per byte grows linearly;
- `minFeeRefScriptCoinsPerByte`, a rational number, the base fee or initial price per byte.

For background on this particular choice of fee calculation, see [7].

5 Governance Actions

This section is part of the `Ledger.Conway.GovernanceActions` module of the formal ledger specification..

We introduce the following distinct bodies with 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`).

Fig. 9 defines several data types used to represent governance actions. The type `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.

- `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 Fig. 10 for definitions);

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.

5.1 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 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 Fig. 11.

[2]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.

5.2 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 contains:

- a pointer to the previous action if required (see Sec. 5.1),
- 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 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
  field
    url  : String
    hash : DocHash

data GovActionType : Type where
  NoConfidence     : GovActionType
  UpdateCommittee : GovActionType
  NewConstitution : GovActionType
  TriggerHF        : GovActionType
  ChangePParams    : GovActionType
  TreasuryWdrl     : GovActionType
  Info             : GovActionType

GovActionData : GovActionType → Type
GovActionData NoConfidence     = ⊤
GovActionData UpdateCommittee = (Credential ⇀ Epoch) × ℙ Credential × ℚ
GovActionData NewConstitution = DocHash × Maybe ScriptHash
GovActionData TriggerHF        = ProtVer
GovActionData ChangePParams    = PParamsUpdate
GovActionData TreasuryWdrl     = RwdAddr ⇀ Coin
GovActionData Info             = ⊤

record GovAction : Type where
  field
    gaType : GovActionType
    gaData : GovActionData gaType
```

Figure 9: Governance actions

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`[2]	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 10: Types of governance actions

```
NeedsHash : GovActionType → 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 11: NeedsHash and HashProtected types

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

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

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

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

Figure 12: Vote and proposal types

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

proposedCC : GovAction → ℙ Credential
proposedCC ⟦ UpdateCommittee , (x , _ , _) ⟧ᵍᵃ = dom x
proposedCC _                                  = ∅
```

Figure 13: Governance helper function

6 Transactions

This section is part of the `Ledger.Conway.Transaction` module of the formal ledger specification., where we define transactions.

A transaction consists of a transaction body, a collection of witnesses and some optional auxiliary data.

Ingredients of the transaction body introduced in the Conway era are the following:

- `txvote`, the list of votes for goverance actions;
- `txprop`, the list of governance proposals;
- `txdonation`, amount of `Coin` to donate to treasury, e.g., to return money to the treasury after a governance action;
- `curTreasury`, the current value of the treasury. This field serves as a precondition to executing Plutus scripts that access the value of the treasury.

Abstract types

```
Ix TxId AuxiliaryData : Type
```

Transaction types

```
record TxBody : Type where
  field
    txins         : ℙ TxIn
    refInputs     : ℙ TxIn
    txouts        : Ix ⇀ TxOut
    txfee         : Coin
    mint          : Value
    txvldt        : Maybe Slot × Maybe Slot
    txcerts       : List DCert
    txwdrls       : Wdrl
    txvote        : List GovVote
    txprop        : List GovProposal
    txdonation    : Coin
    txup          : Maybe Update
    txADhash      : Maybe ADHash
    txNetworkId   : Maybe Network
    curTreasury   : Maybe Coin
    txsize        : ℕ
    txid          : TxId
    collateral    : ℙ TxIn
    reqSigHash    : ℙ KeyHash
    scriptIntHash : Maybe ScriptHash
```

Figure 14: Transactions and related types

7 UTxO

This section is part of the `Ledger.Conway.Utxo` module of the formal ledger specification., where we define types and functions needed for the UTxO transition system.

7.1 Accounting

Figs. 15 to 17 define types and functions needed for the UTxO transition system.

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. See also [8].

Similar to `ScriptPurpose`, `DepositPurpose` carries the information what the deposit is being made for. The deposits are stored in the `deposits` field of `UTxOState` (the type `Deposits` is defined in Fig. 29). `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. (The handling of deposits in the Shelley era is discussed in Corduan et al. [1, Sec 8] and IOHK Formal Methods Team [9, Sec B.2].)

UTxO states

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

Figure 15: UTxO transition-system types

As seen in Fig. 17, 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.

Fig. 16 defines the function `minfee`. In Conway, `minfee` includes the cost for reference scripts. This is calculated using `scriptsCost` (see Fig. 8).

Fig. 16 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

```
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
```

Figure 16: Functions used in UTxO rules

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

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.

```
Scripts-Yes :
  let pp    = Γ .pparams
      sLst = collectPhaseTwoScriptInputs pp tx utxo
    in
     • ValidCertDeposits pp deposits txcerts
     • evalScripts tx sLst ≡ isValid
     • isValid ≡ true
       ────────────────────────────────
       Γ ⊢ ⟦ utxo , fees , deposits , donations ⟧ ⇀⦇ tx ,UTXOS⦈ ⟦ (utxo ∣ txins ᶜ) ∪ˡ (outs txb) , fees
Scripts-No :
  let pp    = Γ .pparams
      sLst = collectPhaseTwoScriptInputs pp tx utxo
    in
     • evalScripts tx sLst ≡ isValid
     • isValid ≡ false
       ────────────────────────────────
       Γ ⊢ ⟦ utxo , fees , deposits , donations ⟧ ⇀⦇ tx ,UTXOS⦈ ⟦ utxo ∣ collateral ᶜ , fees + cbalance
```

Figure 18: UTXOS rule

Fig. 19 ties all the pieces of the UTXO rule together. The symbol ≡? is explained in Sec. 2.

```
UTXO-inductive :
  let pp       = Γ .pparams
      slot     = Γ .slot
      treasury = Γ .treasury
      utxo     = s .UTxOState.utxo
      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 ∣ ]
      inject ((overhead + utxoEntrySize txout) * coinsPerUTxOByte pp) ≤ᵗ getValueʰ txout
  • ∀[ (_ , txout) ∈ ∣ txoutsʰ ∣ ]
      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 ]          NetworkIdOf a ≡ NetworkId
  • txNetworkId  ~ just NetworkId
  • curTreasury ~ just treasury
  • Γ ⊢ s ⇀⦇ tx ,UTXOS⦈ s'
  ────────────────────────────────
  Γ ⊢ s ⇀⦇ tx ,UTXO⦈ s'
```

Figure 19: UTXO inference rules

7.2 Witnessing

This section is part of the `Ledger.Conway.Utxow` module of the formal ledger specification., in which we define witnessing.

The purpose of witnessing is make sure the intended action is authorized by the holder of the signing key. (For details see Corduan et al. [1, Sec 8.3].) Fig. 20 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.

7.3 Plutus script context

CIP-0069 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-0069 are not included here. To provide a CIP-0069-conformant implementation of Plutus to this specification, an additional step processing the `List Data` argument we provide would be required.

In Fig. 22, the line *inputHashes* ⊆ *txdatsHashes* compares two inhabitants of `ℙ DataHash`. In the Alonzo spec, these two terms would have inhabited `ℙ (Maybe DataHash)`, where a `nothing` is thrown out [3, Sec 3.1].

```
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
  = mapˢ (λ (i , o)  → (Spend i , payCred (proj₁ o))) ((utxo ∣ (txins ∪ collateral)) ˢ)
  ∪ mapˢ (λ a        → (Rwrd  a , stake a)) (dom ∣ txwdrls ∣)
  ∪ mapPartial (λ c → (Cert  c ,_) <$> cwitness c) (fromList txcerts)
  ∪ mapˢ (λ x        → (Mint  x , ScriptObj x)) (policies mint)
  ∪ mapˢ (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 ∘ mapˢ proj₂ ∘ credsNeeded

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

Figure 20: Functions used for witnessing

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

Figure 21: UTxOW transition-system types

```
UTXOW-inductive :
  let utxo              = s .utxo
      witsKeyHashes     = mapˢ hash (dom vkSigs)
      witsScriptHashes  = mapˢ 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)
  in
  • ∀[ (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 22: UTXOW inference rules

Derived types

```
GovState = List (GovActionID × GovActionState)

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

Figure 23: Types used in the GOV transition system

8 Governance

This section is part of the `Ledger.Conway.Gov` module of the formal ledger specification., where we define the types required for ledger 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 in Sec. 11.

- `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.

Fig. 26 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.

The function `govActionPriority` assigns a priority to the various types of governance actions. This is useful for ordering lists of governance actions (see `insertGovAction` in Fig. 24). Priority is also used to check if two actions `Overlap`; that is, they would modify the same piece of `EnactState`.

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

Overlap : GovActionType → GovActionType → Type
Overlap NoConfidence UpdateCommittee = ⊤
Overlap UpdateCommittee NoConfidence = ⊤
Overlap a            a'              = a ≡ a'

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

mkGovStatePair : Epoch → GovActionID → RwdAddr → (a : GovAction) → NeedsHash (a .gaType)
                 → 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 .gaType)
          → 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 Γ (r , c) = case r of
CC          → just c ∈ range (gState .ccHotKeys)
DRep        → c ∈ dom (gState .dreps)
SPO         → c ∈ mapˢ KeyHashObj (dom (pState .pools))
  where
          open CertState (GovEnv.certState Γ) using (gState; pState)

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

Figure 24: Functions used in the GOV transition system

Transition relation types

```
_⊢_⇀⦇_,GOV⦈_  : GovEnv × ℕ → GovState → GovVote ⊎ GovProposal → GovState → Type
_⊢_⇀⦇_,GOVS⦈_ : GovEnv → GovState → List (GovVote ⊎ GovProposal) → GovState → Type
```

Figure 25: Type signature of the transition relation of the GOV transition system

```
enactable : EnactState → List (GovActionID × GovActionID)
          → GovActionID × GovActionState → Type
enactable e aidPairs = λ (aidNew , as) → case getHashES e (action as .gaType) 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 → GovActionType → Type
hasParentE e aid gaTy = case getHashES e gaTy of
  nothing   → ⊤
  (just id) → id ≡ aid

hasParent : EnactState → GovState → (gaTy : GovActionType) → NeedsHash gaTy → Type
hasParent e s gaTy aid = case getHash aid of
nothing      → ⊤
(just aid') → hasParentE e aid' gaTy
              ⊎ Any (λ (gid , gas) → gid ≡ aid' × Overlap (gas .action .gaType) gaTy) s
```

Figure 26: Enactability predicate

```
actionValid : ℙ Credential → Maybe ScriptHash → Maybe ScriptHash → Epoch → GovAction → Type
actionValid rewardCreds p ppolicy epoch ⟦ ChangePParams , _ ⟧ᵍᵃ =
  p ≡ ppolicy
actionValid rewardCreds p ppolicy epoch ⟦ TreasuryWdrl , x ⟧ᵍᵃ =
  p ≡ ppolicy × mapˢ RwdAddr.stake (dom x) ⊆ rewardCreds
actionValid rewardCreds p ppolicy epoch ⟦ UpdateCommittee , (new , rem , q) ⟧ᵍᵃ =
  p ≡ nothing × (∀[ e ∈ range new ] epoch < e) × (dom new ∩ rem ≡ᵉ ∅)
actionValid rewardCreds p ppolicy epoch _ =
  p ≡ nothing

actionWellFormed : GovAction → Type
actionWellFormed ⟦ ChangePParams , x ⟧ᵍᵃ = ppdWellFormed x
actionWellFormed ⟦ TreasuryWdrl , x ⟧ᵍᵃ =
  (∀[ a ∈ dom x ] NetworkIdOf a ≡ NetworkId) × (∃[ v ∈ range x ] ¬ (v ≡ 0))
actionWellFormed _ = ⊤
```

Figure 27: Validity and wellformedness predicates

Fig. 27 defines predicates used in the `GOV-Propose` case of the GOV rule to ensure that a governance action is valid and well-formed.

- `actionValid` ensures that the proposed action is valid given the current state of the system:
  - a `ChangePParams` action is valid if the proposal policy is provided;
  - a `TreasuryWdrl` action is valid if the proposal policy is provided and the reward stake credential is registered;
  - an `UpdateCommittee` action is valid if credentials of proposed candidates have not expired, and the action does not propose to both add and remove the same candidate.
- `actionWellFormed` ensures that the proposed action is well-formed:
  - a `ChangePParams` action must preserves well-formedness of the protocol parameters;
  - a `TreasuryWdrl` action is well-formed if the network ID is correct and there is at least one non-zero withdrawal amount in the given `RwdAddr ⇀ Coin` map.

```
data _⊢_⇀⦇_,GOV⦈_ where
  GOV-Vote :
    • (aid , ast) ∈ fromList s
    • canVote (Γ .pparams) (action ast) (proj₁ voter)
    • isRegistered Γ voter
    • ¬ expired (Γ .epoch) ast
      ────────────────────────────────
      (Γ , k) ⊢ s ⇀⦇ inj₁ ⟦ aid , voter , v , machr ⟧ ,GOV⦈ addVote s aid voter v

  GOV-Propose :
    let pp          = Γ .pparams
        e           = Γ .epoch
        enactState  = Γ .enactState
        rewardCreds = Γ .rewardCreds
        prop        = record { returnAddr = addr ; action = a ; anchor = achr
                             ; policy = p ; deposit = d ; prevAction = prev }
    in
    • actionWellFormed a
    • actionValid rewardCreds p (Γ .ppolicy) e a
    • d ≡ pp .govActionDeposit
    • validHFAction prop s enactState
    • hasParent enactState s (a .gaType) prev
    • NetworkIdOf addr ≡ NetworkId
    • CredentialOf addr ∈ rewardCreds
      ────────────────────────────────
      (Γ , k) ⊢ s ⇀⦇ inj₂ prop ,GOV⦈ addAction s (pp .govActionLifetime +ᵉ e)
                                                  (Γ .txid , k) addr a prev

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

Figure 28: Rules for the GOV transition system

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

For `GOV-Vote`, we check that the governance action being voted on exists; that the voter's role is allowed to vote (see `canVote` in Fig. 42); and that the voter's credential is actually associated with their role (see `isRegistered` in Fig. 25).

For `GOV-Propose`, we check the correctness of the deposit along with some and some conditions that ensure the action is well-formed and valid; naturally, these checks depend on the type of action being proposed (see Fig. 27).

9 Certificates

This section is part of the Ledger.Conway.Certs module of the formal ledger specification..

Derived types

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

Deposits = DepositPurpose ⇀ Coin
Rewards  = Credential ⇀ Coin
DReps    = Credential ⇀ Epoch
```

Figure 29: Deposit types

9.1 Changes Introduced in Conway Era

9.1.1 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.

9.1.2 Removal of Pointer Addresses, Genesis Delegations and MIR Certificates

Support for pointer addresses, genesis delegations and MIR certificates is removed (see CIP-1694 and Corduan et al. [5]). 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 (see Corduan et al. [1, Sec 9.2]).

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.

9.1.3 Explicit Deposits

Registration and deregistration of staking credentials are now required to explicitly state the deposit that is being paid or refunded. This deposit is used for checking correctness of transactions with certificates. Including the deposit aligns 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.

Figure 30: Stake pool parameter definitions

```
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
```

Figure 31: Delegation definitions

9.2 Governance Certificate Rules

The rules for transition systems dealing with individual certificates are defined in Figs. 34 and 35.GOVCERT deals with the new certificates relating to DReps and the constitutional committee.

- `GOVCERT-regdrep` registers (or re-registers) a DRep. In case of registration, a deposit needs to be paid. Either way, the activity period of the DRep is reset.
- `GOVCERT-deregdrep` deregisters a DRep.
- `GOVCERT-ccreghot` registers a "hot" credential for constitutional committee members.[3] 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.

Fig. 36 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.

[3] 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.

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

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

record GState : Type where
  field
    dreps     : DReps
    ccHotKeys : Credential ⇀ Maybe Credential

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

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

GovCertEnv = CertEnv
PoolEnv    = PParams
```

Figure 32: Types used for CERTS transition system

```
_⊢_⇀⦇_,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
```

Figure 33: Types for the transition systems relating to certificates

```
DELEG-delegate :
  let Γ = ⟦ pp , pools , delegatees ⟧
  in
  • (c ∉ dom rwds → d ≡ pp .keyDeposit)
  • (c ∈ dom rwds → d ≡ 0)
  • mv ∈ mapˢ (just ∘ credVoter DRep) delegatees ∪
      fromList ( nothing ∷ just abstainRep ∷ just noConfidenceRep ∷ [] )
  • mkh ∈ mapˢ just (dom pools) ∪ ❴ nothing ❵
    ────────────────────────────────
    Γ ⊢ ⟦ 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 ⟧

DELEG-reg :
  • c ∉ dom rwds
  • d ≡ pp .keyDeposit ⊎ d ≡ 0
    ────────────────────────────────
    ⟦ pp , pools , delegatees ⟧ ⊢
      ⟦ vDelegs , sDelegs , rwds ⟧ ⇀⦇ reg c d ,DELEG⦈
      ⟦ vDelegs , sDelegs , rwds ∪ˡ ❴ c , 0 ❵ ⟧
```

Figure 34: Auxiliary DELEG transition system

```
GOVCERT-regdrep :
  let Γ = ⟦ e , pp , vs , wdrls , cc ⟧
  in
  • (d ≡ pp .drepDeposit × c ∉ dom dReps) ⊎ (d ≡ 0 × c ∈ dom dReps)
    ────────────────────────────────
    Γ ⊢ ⟦ dReps , ccKeys ⟧ ⇀⦇ regdrep c d an ,GOVCERT⦈
        ⟦ ❴ c , e + pp .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 cc
```

Figure 35: 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 refresh          = mapPartial getDRepVote (fromList vs)
        refreshedDReps   = mapValueRestricted (const (e + pp .drepActivity)) dReps refresh
        wdrlCreds        = mapˢ stake (dom wdrls)
        validVoteDelegs = voteDelegs ∣^ ( mapˢ (credVoter DRep) (dom dReps)
                                          ∪ fromList (noConfidenceRep ∷ abstainRep ∷ []) )
    in
    • filter isKeyHash wdrlCreds ⊆ dom voteDelegs
    • mapˢ (map₁ stake) (wdrls ˢ) ⊆ rewards ˢ
      ────────────────────────────────
      ⟦ e , pp , vs , wdrls , cc ⟧ ⊢
        ⟦ ⟦ voteDelegs , stakeDelegs , rewards ⟧
        , stᵖ
        , ⟦ dReps , ccHotKeys ⟧
        ⟧ ⇀⦇ _ ,CERTBASE⦈
        ⟦ ⟦ validVoteDelegs , stakeDelegs , constMap wdrlCreds 0 ∪ˡ rewards ⟧
        , stᵖ
        , ⟦ refreshedDReps , ccHotKeys ⟧
        ⟧
```

Figure 36: CERTS rules

10 Ledger

This section is part of the `Ledger.Conway.Ledger` module of the formal ledger specification., where 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.

As there is nothing new to the Conway era in this part of the ledger, we do not present any details of the Agda formalization.

```
record LEnv : Type where
  field
    slot        : Slot
    ppolicy     : Maybe ScriptHash
    pparams     : PParams
    enactState  : EnactState
    treasury    : Coin


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

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

rmOrphanDRepVotes : CertState → GovState → GovState
rmOrphanDRepVotes cs govSt = L.map (map₂ go) govSt
  where
    ifDRepRegistered : Voter → Type
    ifDRepRegistered (r , c) = r ≡ DRep → c ∈ dom (cs .gState .dreps)

    go : GovActionState → GovActionState
    go gas = record gas { votes = filterKeys ifDRepRegistered (gas .votes) }

allColdCreds : GovState → EnactState → ℙ Credential
allColdCreds govSt es =
  ccCreds (es .cc) ∪ concatMapˢ (λ (_ , st) → proposedCC (st .action)) (fromList govSt)
```

Figure 37: Types and functions for the LEDGER transition system

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

  LEDGER-V :
    let txb      = tx .body
        rewards = certState .dState .rewards
    in
    • isValid tx ≡ true
    • ⟦ slot , pp , treasury ⟧ ⊢ utxoSt ⇀⦇ tx ,UTXOW⦈ utxoSt'
    • ⟦ epoch slot , pp , txvote , txwdrls , allColdCreds govSt enactState ⟧ ⊢ certState ⇀⦇ txcerts ,CE
    • ⟦ txid , epoch slot , pp , ppolicy , enactState , certState' , dom rewards ⟧ ⊢ rmOrphanDRepVotes ce
      ────────────────────────────────
      ⟦ slot , ppolicy , pp , enactState , treasury ⟧ ⊢ ⟦ utxoSt , govSt , certState ⟧ ⇀⦇ tx ,LEDGER⦈ ⟦ u

  LEDGER-I :
    • isValid tx ≡ false
    • ⟦ slot , pp , treasury ⟧ ⊢ utxoSt ⇀⦇ tx ,UTXOW⦈ utxoSt'
      ────────────────────────────────
      ⟦ slot , ppolicy , pp , enactState , treasury ⟧ ⊢ ⟦ utxoSt , govSt , certState ⟧ ⇀⦇ tx ,LEDGER⦈ ⟦ u
```

Figure 38: LEDGER transition system

11 Enactment

This section is part of the `Ledger.Conway.Enact` module of the formal ledger specification.

Fig. 39 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 the `withdrawals` field of `EnactState`, but this information is applied and reset in EPOCH (see Fig. 51). Also, enacting these governance actions is the *only* way of modifying `EnactState`.

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

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

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

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

getHash : ∀ {a} → NeedsHash a → Maybe GovActionID
getHash {NoConfidence}     h = just h
getHash {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 → GovActionType → 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 39: Types and function used for the ENACT transition system

Figs. 40 and 41 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-UpdComm : let old     = maybe proj₁ ∅ (s .cc .proj₁)
                      maxTerm = ccMaxTermLengthOf s +ᵉ e
                  in
    ∀[ term ∈ range new ] term ≤ maxTerm
    ──────────────────────────────────────────
                         ( UpdateCommittee )
[gid , t , e ⟧ ⊢ s ⇀⦇    (   (new          ) ,ENACT⦈
                         (   rem           )
                         (   q)            )

  record s { cc = just ((new ∪ˡ old) ∣ rem ᶜ , q) , gid }

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

Figure 40: ENACT transition system

```
Enact-HF :
  ────────────────────────────────────────
[ gid , t , e ] ⊢ s ⇀⦇ ( TriggerHF
                            v        ) ,ENACT⦈ record s { pv = v , gid }

Enact-PParams :
  ────────────────────────────────────────
[gid , t , e ] ⊢ s ⇀⦇ ( ChangePParams
                              up       ) ,ENACT⦈

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

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

Enact-Info :
  ────────────────────────────────────────
[ gid , t , e ] ⊢ s ⇀⦇ ( Info
                           _   ) ,ENACT⦈ s
```

Figure 41: ENACT transition system (continued)

12 Ratification

This section is part of the `Ledger.Conway.Ratify` module of the formal ledger specification.

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.

12.1 Ratification Requirements

Fig. 42 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 fraction of stake associated with yes votes with respect to that associated with yes and no 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 Fig. 42 are protocol parameters.

12.2 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 ga =
  case ga ↓ of
       (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 Q5 )

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

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

Figure 42: Functions related to voting

12.3 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.

Fig. 43 defines some types and functions used in the RATIFY transition system. `CCData` is simply an alias to define some functions more easily.

Fig. 44 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
  field
    stakeDistr : VDeleg ⇀ Coin

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

record RatifyState : Type where
  field
    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 43: 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 Sec. 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.

Fig. 45 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.

Fig. 46 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 RATIFIES transition system is defined as the reflexive-transitive closure of RATIFY, which is defined via three rules, defined in Fig. 48.

- 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 → GovActionType
            → (GovRole × Credential ⇀ Vote) → (VDeleg ⇀ Vote)
actualVotes Γ pparams cc gaTy votes
  =  mapKeys (credVoter CC) actualCCVotes ∪ˡ actualPDRepVotes gaTy
  ∪ˡ actualDRepVotes                      ∪ˡ actualSPOVotes gaTy
  where
  roleVotes : GovRole → VDeleg ⇀ Vote
  roleVotes r = mapKeys (uncurry credVoter) (filter (λ (x , _) → r ≡ proj₁ x) votes)

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

  getCCHotCred : Credential × Epoch → Maybe Credential
  getCCHotCred (c , e) = if currentEpoch > e then nothing
    else case lookupᵐ? ccHotKeys c of
       (just (just c')) → just c'
       _                → nothing -- no hot key or resigned

  SPODefaultVote : GovActionType → VDeleg → Vote
  SPODefaultVote gaT (credVoter SPO (KeyHashObj kh)) = case lookupᵐ? pools kh of
       nothing → Vote.no
       (just p) → case lookupᵐ? delegatees (PoolParams.rewardAccount p) , gaTy 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 ≤ lengthˢ (mapFromPartialFun getCCHotCred (m ˢ))
                        then mapWithKey actualCCVote m
                        else constMap (dom m) Vote.no

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

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

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

Figure 44: Vote counting

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

acceptedStakeRatio : GovRole → ℙ VDeleg → StakeDistrs → (VDeleg ⇀ Vote) → ℚ
acceptedStakeRatio r cc dists votes = acceptedStake /₀ totalStake
  where
    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 (gaType 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 , _) → lengthˢ 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 45: Functions related to ratification

```
verifyPrev : (a : GovActionType) → 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 : GovActionType → Bool
delayingAction NoConfidence    = true
delayingAction UpdateCommittee = true
delayingAction NewConstitution = true
delayingAction TriggerHF       = true
delayingAction ChangePParams   = false
delayingAction TreasuryWdrl    = false
delayingAction Info            = false

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

acceptConds : RatifyEnv → RatifyState → GovActionID × GovActionState → Type
acceptConds Γ stʳ (id , st) =
    accepted Γ es st
  × ¬ delayed (gaType action) prevAction es delay
  × ∃[ es' ] ⟦ id , treasury , currentEpoch ⟧ ⊢ es ⇀⦇ action ,ENACT⦈ es'
```

Figure 46: Functions related to ratification, continued

```
_⊢_⇀⦇_,RATIFY⦈_ : RatifyEnv → RatifyState → GovActionID × GovActionState → RatifyState → Type

_⊢_⇀⦇_,RATIFIES⦈_ : RatifyEnv → RatifyState → List (GovActionID × GovActionState)
                    → RatifyState → Type
```

Figure 47: Types of the RATIFY and RATIFIES transition systems

```
RATIFY-Accept :
  let treasury      = Γ .treasury
      e             = Γ .currentEpoch
      (gaId , gaSt) = a
      action        = gaSt .action
  in
  • acceptConds Γ ⟦ es , removed , d ⟧ a
  • ⟦ gaId , treasury , e ⟧ ⊢ es ⇀⦇ action ,ENACT⦈ es'
    ────────────────────────────────
    Γ ⊢ ⟦ es , removed , d ⟧ ⇀⦇ a ,RATIFY⦈
        ⟦ es' , ❴ a ❵ ∪ removed , delayingAction (gaType action) ⟧

RATIFY-Reject :
  let e             = Γ .currentEpoch
      (gaId , gaSt) = a
  in
  • ¬ acceptConds Γ ⟦ es , removed , d ⟧ a
  • expired e gaSt
    ────────────────────────────────
    Γ ⊢ ⟦ es , removed , d ⟧ ⇀⦇ a ,RATIFY⦈ ⟦ es , ❴ a ❵ ∪ removed , d ⟧

RATIFY-Continue :
  let e             = Γ .currentEpoch
      (gaId , gaSt) = a
  in
  • ¬ acceptConds Γ ⟦ es , removed , d ⟧ a
  • ¬ expired e gaSt
    ────────────────────────────────
    Γ ⊢ ⟦ es , removed , d ⟧ ⇀⦇ a ,RATIFY⦈ ⟦ es , removed , d ⟧

_⊢_⇀⦇_,RATIFIES⦈_ = ReflexiveTransitiveClosure {sts = _⊢_⇀⦇_,RATIFY⦈_}
```

Figure 48: The RATIFY transition system

13 Epoch Boundary

This section is part of the Ledger.Conway.Epoch module of the formal ledger specification.

```
record EpochState : Type where
field
  acnt : Acnt
  ss   : Snapshots
  ls   : LState
  es   : EnactState
  fut  : RatifyState
```

Figure 49: Definitions for the EPOCH and NEWEPOCH transition systems

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

mkStakeDistrs : Snapshot → GovState → Deposits → (Credential ⇀ VDeleg) → StakeDistrs
mkStakeDistrs ss govSt ds delegations .StakeDistrs.stakeDistr =
  aggregateBy ∣ delegations ∣ (Snapshot.stake ss ∪⁺ gaDepositStake govSt ds)
```

Figure 50: Functions for computing stake distributions

Fig. 51 defines the EPOCH transition rule. Currently, this incorporates logic that was previously handled by POOLREAP in the Shelley specification [1, Sec 11.6]; POOLREAP is not implemented here.

The EPOCH rule now also needs to invoke RATIFIES and properly deal with its results by carrying out each of the following tasks.

- 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      = RatifyState.es fut
  es       = record esW { withdrawals = ∅ }
  tmpGovSt = filter (λ x → proj₁ x ∉ mapˢ proj₁ removed) govSt
  orphans  = fromList (getOrphans es tmpGovSt)
  removed' = removed ∪ orphans
  removedGovActions = flip concatMapˢ removed' λ (gaid , gaSt) →
    mapˢ (returnAddr gaSt ,_) ((utxoSt .deposits ∣ ❴ GovActionDeposit gaid ❵) ˢ)
  govActionReturns = aggregate₊ (mapˢ (λ (a , _ , d) → a , d) removedGovActions ᶠˢ)

  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 ∉ mapˢ proj₁ removed') govSt

  dState' = ⟦ dState .voteDelegs , dState .stakeDelegs , dState .rewards ∪⁺ refunds ⟧

  pState' = ⟦ pState .pools ∣ retired ᶜ , pState .retiring ∣ retired ᶜ ⟧

  gState' = ⟦ (if null govSt' then mapValues (1 +_) (gState .dreps) else (gState .dreps))
            , gState .ccHotKeys ∣ ccCreds (es .cc) ⟧

  certState' : CertState
  certState' = ⟦ dState' , pState' , gState' ⟧

  utxoSt' = ⟦ utxoSt .utxo , utxoSt .fees , utxoSt .deposits ∣ mapˢ (proj₁ ∘ proj₂) removedGovActions ᶜ

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

Figure 51: EPOCH transition system

References

[1] 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, LaTeXsource: https://github.com/intersectmbo/cardano-ledger/tree/master/eras/shelley/formal-spec/shelley-ledger.tex; Accessed: 2024-07-30, 2019. [Online]. Available: https://github.com/intersectmbo/cardano-ledger/releases/latest/download/shelley-ledger.pdf.

[2] P. Vinogradova and A. Knispel, *A Formal Specification of the Cardano Ledger with a Native Multi-Asset Implementation*, https://github.com/intersectmbo/cardano-ledger/releases/latest/download/mary-ledger.pdf, LaTeXsource: https://github.com/intersectmbo/cardano-ledger/tree/master/eras/shelley-ma/formal-spec/mary-ledger.tex; Accessed: 2024-07-30, 2019. [Online]. Available: https://github.com/intersectmbo/cardano-ledger/releases/latest/download/mary-ledger.pdf.

[3] P. Vinogradova and A. Knispel, *A Formal Specification of the Cardano Ledger integrating Plutus Core*, https://github.com/intersectmbo/cardano-ledger/releases/latest/download/alonzo-ledger.pdf, Accessed: 2024-07-30, 2021. [Online]. Available: https://github.com/intersectmbo/cardano-ledger/releases/latest/download/alonzo-ledger.pdf.

[4] A. Knispel and J. Corduan, *Formal Specification of the Cardano Ledger for the Babbage era*, https://github.com/intersectmbo/cardano-ledger/releases/latest/download/babbage-ledger.pdf, Accessed: 2024-07-15, 2022. [Online]. Available: https://github.com/intersectmbo/cardano-ledger/releases/latest/download/babbage-ledger.pdf.

[5] 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.

[6] Agda development team, *Agda 2.7.0 documentation*, https://agda.readthedocs.io/en/v2.7.0/, Dec. 2024.

[7] A. Kuleshevich, *Changes to the fee calculation due to Reference Scripts*, https://github.com/IntersectMBO/cardano-ledger/blob/master/docs/adr/2024-08-14_009-refscripts-fee-change.md, 2024. [Online]. Available: https://github.com/IntersectMBO/cardano-ledger/blob/master/docs/adr/2024-08-14_009-refscripts-fee-change.md.

[8] J. Corduan, *Track individual deposits*, https://github.com/IntersectMBO/cardano-ledger/blob/master/docs/adr/2022-12-05_003-track-individual-deposits.md, 2022. [Online]. Available: https://github.com/IntersectMBO/cardano-ledger/blob/master/docs/adr/2022-12-05_003-track-individual-deposits.md.

[9] IOHK Formal Methods Team, *Design Specification for Delegation and Incentives in Cardano, IOHK Deliverable SL-D1*, 2018. [Online]. Available: https://github.com/intersectmbo/cardano-ledger/releases/latest/download/shelley-delegation.pdf.

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. For more details, the reader is encouraged to consult the official Agda documentation [6].

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 agda.readthedocs.io/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 CIP-1694, 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 [9]. 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.