Contents

1 Introduction

This is the work-in-progress specification of the Cardano ledger. The Agda source code with which we formalize the ledger specification and which generates this pdf document is open source and resides at the following

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

The current status of each individual era is described in Table 1.

Era	Figures	Prose	Cleanup
Shelley [1]	Partial	Partial	Not started
Shelley-MA [2]	Partial	Partial	Not started
Alonzo [3]	Partial	Partial	Not started
Babbage [4]	Not started	Not started	Not started
Conway [5]	Complete	Partial	Partial

Table 1: Specification progress

1.1 Overview

This document describes, in a precise and executable way, the behavior of the Cardano ledger that can be updated in response to a series of events. Because of the precise nature of the document, it can be dense and difficult to read at times, and it can be helpful to have a high-level understanding of what it is trying to describe, which we present below. Keep in mind that this section focuses on intuition, using terms (set in *italics*) which may be unfamiliar to some readers, but rest assured that later sections of the document will make the intuition and italicized terms precise.

1.2 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. 2 and B, 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.

1.3 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. (See Fig. 1.)Thus each such relation represents the transition rule of the state machine; X is simply a placeholder for the name of the transition rule.

1.4 Ledger State Transition Rules

By a *ledger* we mean a structure that contains information about how funds in the system are distributed accross accounts—that is, account balances, how such balances should be adjusted when transactions and proposals are processed, the ADA currently held in the treasury reserve, a list of *stake pools* operating the network, and so on.

The ledger can be updated in response to certain events, such as receiving a new transaction, time passing and crossing an *epoch boundary*, enacting a *governance proposal*, to name a few. This document defines, as part of the behaior of the ledger, a set of rules that determine which events are valid and exactly how the state of the ledger should be updated in response to those events. The primary aim of this document is to provide a precise description of this system—the ledger state, valid events and the rules for processing them.

We will model this via a number of *state transition systems* (STS) which from now on we refer to as "transition rules" or just "rules." These rules describe the different behaviors that determine how the whole system evolves and, taken together, they comprise a full description of the ledger protocol. Each transition rule consists of the following components:

- an *environment* consisting of data, read from the ledger state or the outside world, which should be considered constant for the purposes of the rule;
- an *initial state*, consisting of the subset of the full ledger state that is relevant to the rule and which the rule can update;
- a *signal* or *event*, with associated data, that the rule can receive or observe;
- a set of *preconditions* that must be met in order for the transition to be valid;
- a new state that results from the transition rule.

For example, the UTXOW transition rule defined in Fig. 34 of Sec. 12.2 checks that, among other things, a given transaction is signed by the appropriate parties.

The transition rules can be composed in the sense that they may require other transition rules to hold as part of their preconditions. For example, the UTXOW rule mentioned above requires the UTXO rule, which checks that the inputs to the transaction exist, that the transaction is balanced, and several other conditions.

A brief description of each transition rule is provided below, with a link to an Agda module and reference to a section where the rule is formally defined.

- `CHAIN` is the top level transition in response to a new block that applies the NEWEPOCH transition when crossing an epoch boundary, and the LEDGERS transition on the list of transactions in the body (Sec. 20).
- `NEWEPOCH` computes the new state as of the start of a new epoch; includes the previous EPOCH transition (Sec. 19).
- `EPOCH` computes the new state as of the end of an epoch; includes the ENACT, RATIFY, and SNAP transition rules (Sec. 19).
- `RATIFY` decides whether a pending governance action has reached the thresholds it needs to be ratified (Sec. 17).

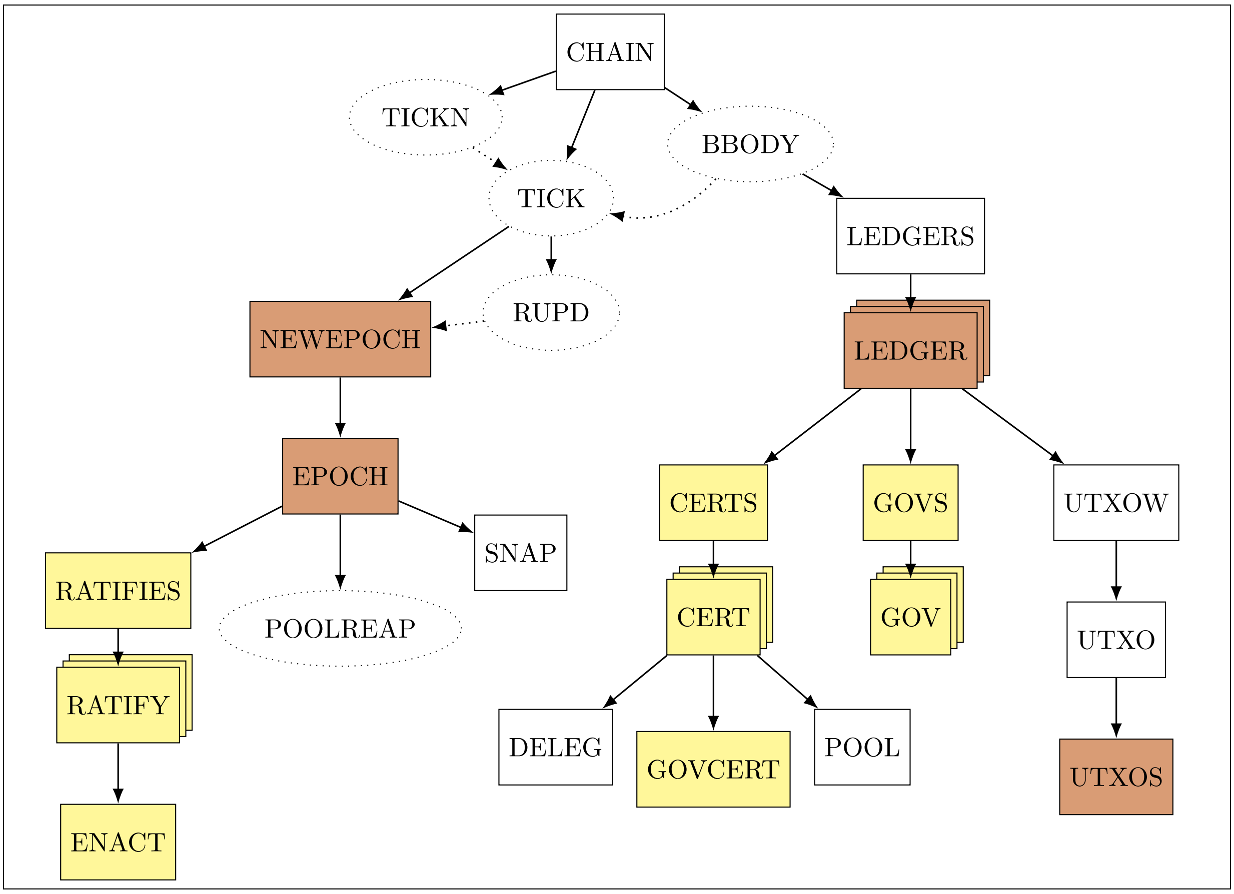


Figure 1: State transition rules of the ledger specification, presented as a directed graph; each node represents a transition rule; an arrow from rule A to rule B indicates that B appears among the premises of A; a dotted arrow represents a dependency in the sense that the output of the target node is an input to the source node, either as part of the source state, the environment or the event (■ rules added in Conway; ■ rules modified in Conway; dotted ellipses represent rules that are not yet formalized in Agda).

- `ENACT` applies the result of a previously ratified governance action, such as triggering a hard fork or updating the protocol parameters (Sec. 16).
- `SNAP` computes new stake distribution snapshots (Sec. 19).
- `LEDGERS` applies LEDGER repeatedly as needed, for each transaction in a list of transactions (Sec. 15).
- `LEDGER` is the full state update in response to a single transaction; it includes the UTXOW, GOV, and CERTS rules (Sec. 15).
- `CERTS` applies CERT repeatedly for each certificate in the transaction (Sec. 14).
- `CERT` combines DELEG, POOL, GOVCERT transition rules, as well as some additional rules shared by all three (Sec. 14).
- `DELEG` handles registering stake addresses and delegating to a stake pool (Sec. 14).
- `GOVCERT` handles registering and delegating to DReps (Sec. 14).
- `POOL` handles registering and retiring stake pools (Sec. 14).
- `GOV` handles voting and submitting governance proposals (Sec. 13).
- `UTXOW` checks that a transaction is witnessed correctly with the appropriate signatures, datums, and scripts; includes the UTXO transition rule (Sec. 12.2).

- UTXO checks core invariants for an individual transaction to be valid, such as the transaction being balanced, fees being paid, etc; include the UTXOS transition rule (Sec. 12).
- UTXOS checks that any relevant scripts needed by the transaction evaluate to true (Sec. 12).

1.5 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 _⊢_⇀⟦_⟧*_ of a relation _⊢_⇀⟦_⟧_ is defined in Fig. 2. In the remainder of the text, the closure operation is called `ReflexiveTransitiveClosure`.

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 2: Reflexive transitive closure

1.6 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. 3 which is parametrized over the state transition relation.

```
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 3: 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.

1.7 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
```

1.8 Propositions as Types, Properties and Relations

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

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. 1.7. 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 $\mathbb{P}$ 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 $A \uplus B$, and their product is denoted by $A \times B$. 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. B.

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 `|` A, and the restriction to the complement of A is written f `|` 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 Cryptographic Primitives

This section is part of the `Ledger.Conway.Crypto` module of the formal ledger specification., in which we rely on a public key signing scheme for verification of spending.

Fig. 4 shows some of the types, functions and properties of this scheme.

Types & functions

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

KeyPair = Σ[ sk ∈ SKey ] Σ[ vk ∈ VKey ] isKeyPair sk vk
```

Property of signatures

```
isSigned-correct : ((sk , vk , _) : KeyPair) (d : Ser) (σ : Sig)
                 → sign sk d ≡ σ → isSigned vk d σ
```

Figure 4: Definitions for the public key signature scheme

4 Base Types

This section is part of the `Ledger.Conway.BaseTypes` module of the formal ledger specification., in which we define some of the most basic types used throughout the ledger.

```
Coin  = ℕ
Slot  = ℕ
Epoch = ℕ
```

Figure 5: Some basic types used in many places in the ledger

```
UnitInterval = [ x ∈ ℚ ∣ (0 ≤ x) × (x ≤ 1) ]
```

Figure 6: Refinement types used in some places in the ledger

5 Token Algebras

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

Abstract types

```
PolicyId
```

Derived types

```
record TokenAlgebra : Type₁ where
  field
    Value : Set
    ⦃ Value-CommutativeMonoid ⦄ : CommutativeMonoid 0ℓ 0ℓ Value

    coin                      : Value → Coin
    inject                    : Coin → Value
    policies                  : Value → ℙ PolicyId
    size                      : Value → MemoryEstimate
    _≤ᵗ_                      : Value → Value → Type
    coin∘inject≗id            : coin ∘ inject ≗ id
    coinIsMonoidHomomorphism  : IsMonoidHomomorphism coin
```

Helper functions

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

Figure 7: Token algebras, used for multi-assets

6 Addresses

This section is part of the `Ledger.Conway.Address` module of the formal ledger specification., in which we define credentials and various types of addresses here.

A credential contains a hash, either of a verifying (public) key (`isVKey`) or of a script (`isScript`).

N.B. in the Shelley era the type of the `stake` field of the `BaseAddr` record was `Credential` (see Corduan et al. [1, Sec 4]); to specify an address with no stake, we would use an "enterprise" address. In contrast, the type of `stake` in the Conway era is `Maybe Credential`, so we can now use `BaseAddr` to specify an address with no stake by setting `stake` to `nothing`.

Abstract types

```
Network
KeyHash
ScriptHash
```

Derived types

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

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

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

record RwdAddr : Type where
  field net   : Network
        stake : Credential

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

Addr       = BaseAddr       ⊎ BootstrapAddr
VKeyAddr   = VKeyBaseAddr   ⊎ VKeyBootstrapAddr
ScriptAddr = ScriptBaseAddr ⊎ ScriptBootstrapAddr
```

Helper functions

```
payCred      : Addr → Credential
stakeCred    : Addr → Maybe Credential
netId        : Addr → Network
isVKeyAddr   : Addr → Type
isScriptAddr : Addr → Type

isVKeyAddr      = isVKey ∘ payCred
isScriptAddr    = isScript ∘ payCred
isScriptRwdAddr = isScript ∘ CredentialOf
```

Figure 8: Definitions used in Addresses

7 Scripts

This section is part of the `Ledger.Conway.Script` module of the formal ledger specification., in which we define `Timelock` scripts.

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

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


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

Figure 9: Timelock scripts and their evaluation

8 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.

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

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

record Hastreasury {a} (A : Type a) : Type a where
  field treasuryOf : A → Coin
open Hastreasury ⦃...⦄ public

record Hasreserves {a} (A : Type a) : Type a where
  field reservesOf : A → Coin
open Hasreserves ⦃...⦄ public

ProtVer : Type
ProtVer = ℕ × ℕ

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

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

Figure 10: Definitions related to protocol parameters

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

Figure 11: Protocol parameter group definition

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;

```
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 12: Protocol parameter threshold definitions

- 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. 17.1; in particular, the definition of the threshold function appears in Fig. 56.)

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. 13 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. 13 also defines the function paramsWellFormed which performs some sanity checks on protocol parameters. Fig. 15 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 13: 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 14: 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 15: 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 16: Calculation of fees for reference scripts

9 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. 16) 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. 29)—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. 13):

- `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].

10 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. 17 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. 18 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.

10.1 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. 19.

[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.

10.2 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. 10.1),
- 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 17: 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 18: 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 19: 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 20: 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 21: Governance helper function

11 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.

Some key ingredients in the transaction body are:

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

Abstract types

```
Ix TxId AuxiliaryData : Type
```

Derived types

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

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

Transaction types

```
record TxBody : Type where
  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
record TxWitnesses : Type where
  field
    vkSigs  : VKey ⇀ Sig
    scripts : ℙ Script
    txdats  : DataHash ⇀ Datum
    txrdmrs : RdmrPtr ⇀ Redeemer × ExUnits

  scriptsP1 : ℙ P1Script
  scriptsP1 = mapPartial isInj₁ scripts

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


Figure 22: Transactions and related types

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

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

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

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

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

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

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

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

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

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

Figure 23: Functions related to transactions

12 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.

12.1 Accounting

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


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

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

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

Figure 24: Functions supporting UTxO rules

Figs. 24 to 28 define functions needed for the UTxO transition system.

Fig. 25 defines the types needed for the UTxO transition system. The UTxO transition system is given in Fig. 31.

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

The deposits have been reworked since the original Shelley design. We now track the amount of every deposit individually. This fixes an issue in the original design: An increase in deposit amounts would allow an attacker to make lots of deposits before that change and refund them after the change. The additional funds necessary would have been provided by the treasury. Since changes to protocol parameters were (and still are) known publicly and guaranteed before they are enacted, this comes at zero risk for an attacker. This means the deposit amounts could realistically never be increased. This issue is gone with the new design. 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. 41). `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 environment

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

UTxO states

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

UTxO transitions

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

Figure 25: UTxO transition-system types

As seen in Fig. 29, 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. 26 defines the function `minfee`. In Conway, `minfee` includes the cost for reference scripts. This is calculated using `scriptsCost` (see Fig. 16).

Fig. 26 also shows the signature of `ValidCertDeposits`. Inhabitants of this type are constructed in one of eight ways, corresponding to seven certificate types plus one for an empty list of certificates. Suffice it to say that `ValidCertDeposits` is used to check the validity of the deposits in a transaction so that the function `updateCertDeposits` can correctly register and deregister deposits in the UTxO state based on the certificates in the transaction.

Fig. 31 ties all the pieces of the UTXO rule together. The symbol `≡?` is explained in Sec. 2.

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

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

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

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

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

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

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

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

Figure 26: Functions used in UTxO rules

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

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

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

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

Figure 27: Functions used in UTxO rules, continued

```
data inInterval (slot : Slot) : (Maybe Slot × Maybe Slot) → Type where
  both  : ∀ {l r}   → l ≤ slot × slot ≤ r → inInterval slot (just l  , just r)
  lower : ∀ {l}     → l ≤ slot            → inInterval slot (just l  , nothing)
  upper : ∀ {r}     → slot ≤ r            → inInterval slot (nothing , just r)
  none  :                                   inInterval slot (nothing , nothing)
feesOK : PParams → Tx → UTxO → Type
feesOK pp tx utxo = ( minfee pp utxo tx ≤ txfee × (txrdmrs ˢ ≢ ∅
                    → ( All (λ (addr , _) → isVKeyAddr addr) collateralRange
                      × isAdaOnly bal
                      × coin bal * 100 ≥ txfee * pp .collateralPercentage
                      × collateral ≢ ∅
                      )
                    )
                  )
  where
    open Tx tx; open TxBody body; open TxWitnesses wits; open PParams pp
    collateralRange = range    ((mapValues txOutHash utxo) | collateral)
    bal             = balance (utxo | collateral)
```

Figure 28: Functions used in UTxO rules, continued

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

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

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

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

Figure 29: Functions used in UTxO rules, continued

```
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 30: UTXOS rule

```
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 31: UTXO inference rules

12.2 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. 32 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.

12.3 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. 34, 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 32: Functions used for witnessing

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

Figure 33: 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 34: 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 35: Types used in the GOV transition system

13 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. 16.

- `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. 38 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. 36). 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 36: 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 37: 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 38: 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 39: Validity and wellformedness predicates

Fig. 39 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 40: Rules for the GOV transition system

The GOVS transition system is now given as the reflexive-transitive closure of the system GOV, described in Fig. 40.

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. 56); and that the voter's credential is actually associated with their role (see `isRegistered` in Fig. 37).

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. 39).

14 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 41: Deposit types

14.1 Changes Introduced in Conway Era

14.1.1 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.

14.1.2 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.

```
record PoolParams : Type where
  field
    owners        : ℙ KeyHash
    cost          : Coin
    margin        : UnitInterval
    pledge        : Coin
    rewardAccount : Credential
```

Figure 42: 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
cwitness : DCert → Maybe Credential
cwitness (delegate c _ _ _) = just c
cwitness (dereg c _)        = just c
cwitness (regpool kh _)     = just $ KeyHashObj kh
cwitness (retirepool kh _)  = just $ KeyHashObj kh
cwitness (regdrep c _ _)    = just c
cwitness (deregdrep c _)    = just c
cwitness (ccreghot c _)     = just c
```

Figure 43: Delegation definitions

14.1.3 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.

14.2 Governance Certificate Rules

The rules for transition systems dealing with individual certificates are defined in Figs. 46 to 48.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.

[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.

Fig. 49 assembles the CERTS transition system by bundling the previously defined pieces together into the CERT system, and then taking the reflexive-transitive closure of CERT together with CERTBASE as the base case. CERTBASE does the following:

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

```
record CertEnv : 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 PState : Type where
  field
    pools    : KeyHash ⇀ PoolParams
    retiring : KeyHash ⇀ Epoch

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 44: 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 45: 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 46: Auxiliary DELEG transition system

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

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

Figure 47: Auxiliary POOL 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 48: 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 49: CERTS rules

15 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.

```
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 50: 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 51: LEDGER transition system

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

Figure 52: LEDGERS transition system

16 Enactment

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

Fig. 53 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. 78). 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 53: Types and function used for the ENACT transition system

Figs. 54 and 55 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 54: 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 55: ENACT transition system (continued)

17 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.

17.1 Ratification Requirements

Fig. 56 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. 56 are protocol parameters.

17.2 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 56: Functions related to voting

17.3 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. 57 defines some types and functions used in the RATIFY transition system. CCData is simply an alias to define some functions more easily.

Fig. 58 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 57: 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. D.) 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. 59 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. 60 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. 62.

- 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 58: 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 59: 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 60: Functions related to ratification, continued

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

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

Figure 61: 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 62: The RATIFY transition system

18 Rewards

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

This section defines how rewards for stake pools and their delegators are calculated and paid out. This calculation has two main aspects:

- The *amount* of rewards to be paid out. This is defined in Sec. 18.2.
- The *time* when rewards are paid out. This is defined in Sec. 18.3.

18.1 Rewards Motivation

In order to operate, any blockchain needs to attract parties that are willing to spend computational and network resources on processing transactions and producing new blocks. These parties, called *block producers*, are incentivized by monetary *rewards*.

Cardano is a proof-of-stake (PoS) blockchain: through a random lottery, one block producer is selected to produce one particular block. The probability for being select depends on their *stake* of Ada, that is the amount of Ada that they (and their delegators) own relative to the total amount of Ada. (We will explain delegation below.) After successful block production, the block producer is eligible for a share of the rewards.

The rewards for block producers come from two sources: during an initial period, rewards are paid out from the *reserve*, which is an initial allocation of Ada created for this very purpose. Over time, the reserve is depleted, and rewards are sourced from transaction fees.

Rewards are paid out epoch by epoch.

Rewards are collective, but depend on performance: after every epoch, a fraction of the available reserve and the transaction fees accumulated during that epoch are added together. This sum is paid out to the block producers proportionally to how many blocks they have created each. In order to avoid perverse incentives, block producers do not receive individual rewards that depend on the content of their blocks.

Not all people can or want to set up and administer a dedicated computer that produces blocks. However, these people still own Ada, and their stake is relevant for block production. Specifically, these people have the option to *delegate* their stake to a *stake pool*, which belongs to a block producer. This stake counts towards the stake of the pool in the block production lottery. In turn, the protocol distributes the rewards for produced blocks to the stake pool owner and their delegators. The owner receives a fixed fee ("cost") and a share of the rewards ("margin"). The remainder is distributed among delegators in proportion to their stake. By design, delegation and ownership are separate — delegation counts towards the stake of the pool, but delegators remain in full control of their Ada, stake pools cannot spend delegated Ada.

Stake pools compete for delegators based on fees and performance. In order to achieve stable blockchain operation, the rewards are chosen such that they incentivize the system to evolve into a large, but fixed number of stake pools that attract most of the stake. For more details about the design and rationale of the rewards and delegation system, see IOHK Formal Methods Team [9].

18.2 Amount of Rewards to be Paid Out

18.2.1 Precision of Arithmetic Operations

When computing rewards, all intermediate results are computed using rational numbers, $\mathbb{Q}$, and converted to `Coin` using the `floor` function at the very end of the computation.

Note for implementors: Values in $\mathbb{Q}$ can have arbitrarily large nominators and denominators. Please use an appropriate type that represents rational numbers as fractions of unbounded nominators and denominators. Types such as `Double`, `Float`, `BigDecimal` (Java Platform), or

`Fixed` (fixed-precision arithmetic) do *not* faithfully represent the rational numbers, and are *not* suitable for computing rewards according to this specification!

We use the following arithmetic operations besides basic arithmetic:

- `fromℕ`: Interpret a natural number as a rational number.
- `floor`: Round a rational number to the next smaller integer.
- `posPart`: Convert an integer to a natural number by mapping all negative numbers to zero.
- `÷`: Division of rational numbers.
- `÷₀`: Division operator that returns zero when the denominator is zero.
- `/`: Division operator that maps integer arguments to a rational number.
- `/₀`: Like `÷₀`, but with integer arguments.

18.2.2 Rewards Distribution Calculation

This section defines the amount of rewards that are paid out to stake pools and their delegators.

Fig. 63 defines the function `maxPool` which gives the maximum reward a stake pool can receive in an epoch. Relevant quantities are:

- *rewardPot*: Total rewards to be paid out after the epoch.
- *stake*: Relative stake of the pool.
- *pledge*: Relative stake that the pool owner has pledged themselves to the pool.
- `z0`: Relative stake of a fully saturated pool.
- `nopt`: Protocol parameter, planned number of block producers.
- `a0`: Protocol parameter that incentivizes higher pledges.
- `rewardℚ`: Pool rewards as a rational number.
- `rewardℕ`: Pool rewards after rounding to a natural number of lovelace.

```
maxPool : PParams → Coin → UnitInterval → UnitInterval → Coin
maxPool pparams rewardPot stake pledge = rewardℕ
  where
    a0    = 0 ⊔ℚ pparams .PParams.a0
    1+a0 = 1 + a0
    nopt = 1 ⊔ pparams .PParams.nopt
    z0      = 1 / nopt
    stake'  = fromUnitInterval stake ⊓ z0
    pledge' = fromUnitInterval pledge ⊓ z0
    rewardℚ =
      fromℕ rewardPot ÷ 1+a0
      * (stake' + pledge' * a0 * (stake' - pledge' * (z0 - stake') ÷ z0) ÷ z0)
    rewardℕ = posPart (floor rewardℚ)
```

Figure 63: Function maxPool used for computing a Reward Update

Fig. 64 defines the function `mkApparentPerformance` which computes the apparent performance of a stake pool. Relevant quantities are:

- *stake*: Relative active stake of the pool.

- *poolBlocks*: Number of blocks that the pool added to the chain in the last epoch.
- *totalBlocks*: Total number of blocks added in the last epoch.

```
mkApparentPerformance : UnitInterval → ℕ → ℕ → ℚ
mkApparentPerformance stake poolBlocks totalBlocks = ratioBlocks ÷₀ stake'
  where
    stake' = fromUnitInterval stake
    ratioBlocks = (pos poolBlocks) / (1 ⊔ totalBlocks)
```

Figure 64: Function mkApparentPerformance used for computing a Reward Update

Fig. 65 defines the functions `rewardOwners` and `rewardMember`. Their purpose is to divide the reward for one pool between pool owners and individual delegators by taking into account a fixed pool cost, a relative pool margin, and the stake of each member. The rewards will be distributed as follows:

- *rewardOwners*: These funds will go to the `rewardAccount` specified in the pool registration certificate.
- *rewardMember*: These funds will go to the reward accounts of the individual delegators.

Relevant quantities for the functions are:

- *rewards*: Rewards paid out to this pool.
- *pool*: Pool parameters, such as cost and margin.
- *ownerStake*: Stake of the pool owners relative to the total amount of Ada.
- *memberStake*: Stake of the pool member relative to the total amount of Ada.
- *stake*: Stake of the whole pool relative to the total amount of Ada.

Fig. 66 defines the function `rewardOnePool` which calculates the rewards given out to each member of a given pool. Relevant quantities are:

- *rewardPot*: Total rewards to be paid out for this epoch.
- *n*: Number of blocks produced by the pool in the last epoch.
- *N*: Expectation value of the number of blocks to be produced by the pool.
- *stakeDistr*: Distribution of stake, as mapping from `Credential` to `Coin`.
- *σ*: Total relative stake controlled by the pool.
- *σa*: Total active relative stake controlled by the pool, used for selecting block producers.
- *tot*: Total amount of Ada in circulation, for computing the relative stake.
- `mkRelativeStake`: Compute stake relative to the total amount in circulation.
- `ownerStake`: Total amount of stake controlled by the stake pool operator and owners.
- `maxP`: Maximum rewards the pool can claim if the pledge is met, and zero otherwise.
- `poolReward`: Actual rewards to be paid out to this pool.

Fig. 67 defines the function `poolStake` which filters the stake distribution to one stake pool. Relevant quantities are:

```
rewardOwners : Coin → PoolParams → UnitInterval → UnitInterval → Coin
rewardOwners rewards pool ownerStake stake = if rewards ≤ cost
  then rewards
  else cost + posPart (floor (
         (fromℕ rewards - fromℕ cost) * (margin + (1 - margin) * ratioStake)))
  where
    ratioStake = fromUnitInterval ownerStake ÷₀ fromUnitInterval stake
    cost       = pool .PoolParams.cost
    margin     = fromUnitInterval (pool .PoolParams.margin)


rewardMember : Coin → PoolParams → UnitInterval → UnitInterval → Coin
rewardMember rewards pool memberStake stake = if rewards ≤ cost
  then 0
  else posPart (floor (
         (fromℕ rewards - fromℕ cost) * ((1 - margin) * ratioStake)))
  where
    ratioStake = fromUnitInterval memberStake ÷₀ fromUnitInterval stake
    cost       = pool .PoolParams.cost
    margin     = fromUnitInterval (pool .PoolParams.margin)
```

Figure 65: Functions rewardOwners and rewardMember

- *hk*: `KeyHash` of the stake pool to be filtered by.
- *delegs*: Mapping from `Credentials` to stake pool that they delegate to.
- *stake*: Distribution of stake for all `Credentials`.

Fig. 68 defines the function `reward` which applies `rewardOnePool` to each registered stake pool. Relevant quantities are:

- `uncurryᵐ`: Helper function to rearrange a nested mapping.
- *blocks*: Number of blocks produced by pools in the last epoch, as a mapping from pool `KeyHash` to number.
- *poolParams*: Parameters of all known stake pools.
- *stake*: Distribution of stake, as mapping from `Credential` to `Coin`.
- *delegs*: Mapping from `Credentials` to stake pool that they delegate to.
- *total*: Total stake = amount of Ada in circulation, for computing the relative stake.
- `active`: Active stake = amount of Ada that was used for selecting block producers.
- `Σ_/total`: Sum of stake divided by total stake.
- `Σ_/active`: Sum of stake divided by active stake.
- `N`: Total number of blocks produced in the last epoch.
- `pdata`: Data needed to compute rewards for each pool.

```
Stake = Credential ⇀ Coin

rewardOnePool : PParams → Coin → ℕ → ℕ → PoolParams
  → Stake → UnitInterval → UnitInterval → Coin → (Credential ⇀ Coin)
rewardOnePool pparams rewardPot n N pool stakeDistr σ σa tot = rewards
  where
    mkRelativeStake = λ coin → clamp (coin /₀ tot)
    owners = mapˢ KeyHashObj (pool .PoolParams.owners)
    ownerStake = ∑[ c ← stakeDistr ∣ owners ] c
    pledge = pool .PoolParams.pledge
    maxP = if pledge ≤ ownerStake
      then maxPool pparams rewardPot σ (mkRelativeStake pledge)
      else 0
    apparentPerformance = mkApparentPerformance σa n N
    poolReward = posPart (floor (apparentPerformance * fromℕ maxP))
    memberRewards =
      mapValues (λ coin → rewardMember poolReward pool (mkRelativeStake coin) σ)
        (stakeDistr ∣ owners ᶜ)
    ownersRewards =
      ❴ pool .PoolParams.rewardAccount
      , rewardOwners poolReward pool (mkRelativeStake ownerStake) σ ❵ᵐ
    rewards = memberRewards ∪⁺ ownersRewards
```

Figure 66: Function rewardOnePool used for computing a Reward Update

```
Delegations = Credential ⇀ KeyHash

poolStake : KeyHash → Delegations → Stake → Stake
poolStake hk delegs stake = stake ∣ dom (delegs ∣^ ❴ hk ❵)
```

Figure 67: Function poolStake

18.2.3 Reward Update

This section defines the `RewardUpdate` type, which records the net flow of Ada due to paying out rewards after an epoch. This type is defined in Fig. 69. The update consists of four net flows:

- `Δt`: The change to the treasury. This will be a positive value.
- `Δr`: The change to the reserves. We typically expect this to be a negative value.
- `Δf`: The change to the fee pot. This will be a negative value.
- `rs`: The map of new individual rewards, to be added to the existing rewards.

We require these net flows to satisfy certain constraints that are also stored in the `RewardUpdate` data type. Specifically, `flowConservation` states that all four net flows add up to zero, and we state the directions of `Δt` and `Δf`.

The function `createRUpd` calculates the `RewardUpdate`, but requires the definition of the type `EpochState`, so we have to defer the definition of this function to Sec. 19.

Fig. 70 captures the potential movement of funds in the entire system that can happen during one transition step as described in this document. Exception: Withdrawals from the "Treasury"

```
BlocksMade = KeyHash ⇀ ℕ

uncurryᵐ :
  A ⇀ (B ⇀ C) → (A × B) ⇀ C

reward : PParams → BlocksMade → Coin → (KeyHash ⇀ PoolParams)
  → Stake → Delegations → Coin → (Credential ⇀ Coin)
reward pp blocks rewardPot poolParams stake delegs total = rewards
  where
    active = ∑[ c ← stake ] c
    Σ_/total = λ st → clamp ((∑[ c ← st ] c) /₀ total)
    Σ_/active = λ st → clamp ((∑[ c ← st ] c) /₀ active)
    N = ∑[ m ← blocks ] m
    mkPoolData = λ hk p →
      map (λ n → (n , p , poolStake hk delegs stake)) (lookupᵐ? blocks hk)
    pdata = mapMaybeWithKeyᵐ mkPoolData poolParams

    results : (KeyHash × Credential) ⇀ Coin
    results = uncurryᵐ (mapValues (λ (n , p , s)
      → rewardOnePool pp rewardPot n N p s (Σ s /total) (Σ s /active) total)
      pdata)
    rewards = aggregateBy
      (mapˢ (λ (kh , cred) → (kh , cred) , cred) (dom results))
      results
```

Figure 68: Function reward used for computing a Reward Update

are not shown in this diagram, they can move funds into "Reward accounts". Value is moved between accounting pots, but the total amount of value in the system remains constant. In particular, the red subgraph represents the inputs and outputs to the `rewardPot`, a temporary variable used during the reward update calculation in the function `createRUpd`. Each red arrow corresponds to one field of the `RewardUpdate` data type. The blue arrows represent the movement of funds after they have passed through the `rewardPot`.

18.2.4 Stake Distribution Calculation

This section defines the calculation of the stake distribution for the purpose of calculating rewards.

Fig. 71 defines the type `Snapshot` that represents a stake distribution snapshot. Such a snapshot contains the essential data needed to compute rewards:

- `stake` A stake distribution, that is a mapping from credentials to coin.
- `delegations`: A delegation map, that is a mapping from credentials to the stake pools that they delegate to.
- `poolParameters`: A mapping that stores the pool parameters of each stake pool.

Fig. 72 defines the calculation of the stake distribution from the data contained in a ledger state. Here,

- `aggregate₊` takes a relation $R \subset A \times V$, where V is any monoid with operation $+$, and returns a mapping $A \rightharpoonup B$ such that any item $a \in A$ is mapped to the sum (using the operation $+$) of all $b \in B$ such that $(a , b) \in R$.

```
record RewardUpdate : Set where
  field
    Δt Δr Δf          : ℤ
    rs                : Credential ⇀ Coin
    flowConservation : Δt + Δr + Δf + pos (∑[ c ← rs ] c) ≡ 0
    Δt-nonnegative   : 0 ≤ Δt
    Δf-nonpositive   : Δf ≤ 0
```

Figure 69: RewardUpdate type

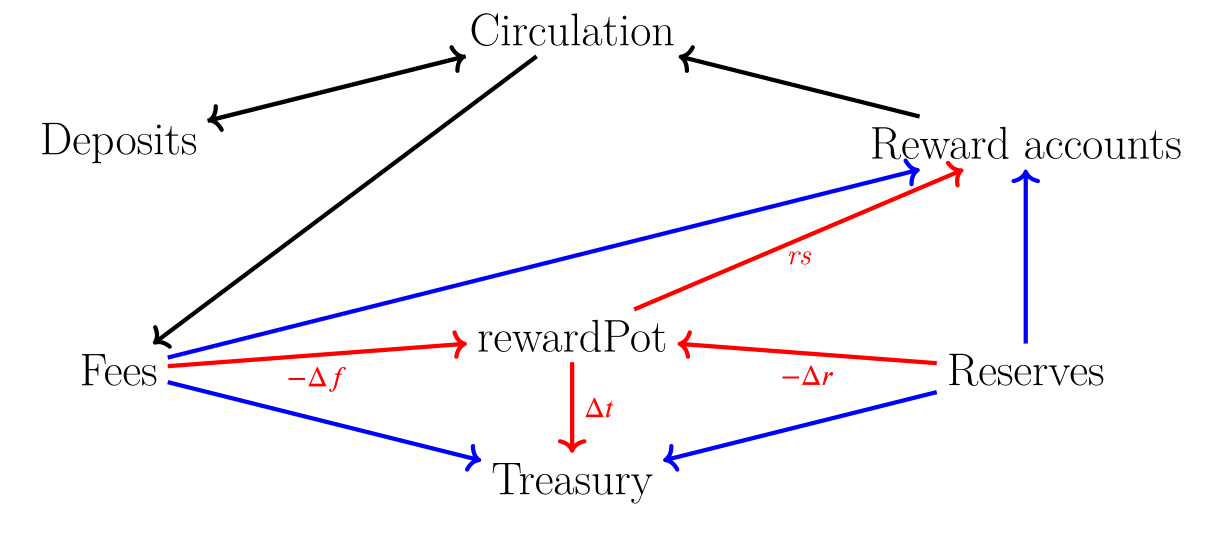


Figure 70: Preservation of funds and rewards

```
record Snapshot : Set where
  field
    stake          : Credential ⇀ Coin
    delegations    : Credential ⇀ KeyHash
    poolParameters : KeyHash ⇀ PoolParams
```

Figure 71: Definitions of the Snapshot type

- `m` is the stake relation computed from the UTxO set.
- `stakeRelation` is the total stake relation obtained by combining the stake from the UTxO set with the stake from the reward accounts.

18.3 Timing when Rewards are Paid Out

18.3.1 Timeline of the Rewards Calculation

As described in Sec. 18.1, the probability of producing a block depends on the stake delegated to the block producer. However, the stake distribution changes over time, as funds are transferred between parties. This raises the question: What is the point in time from which we take the

```
stakeDistr : UTxO → DState → PState → Snapshot
stakeDistr utxo stᵈ pState =
    ⟦ aggregate₊ (stakeRelation ᶠˢ) , stakeDelegs , poolParams ⟧
  where
    poolParams = pState .PState.pools
    open DState stᵈ using (stakeDelegs; rewards)
    m = mapˢ (λ a → (a , cbalance (utxo ∣^' λ i → getStakeCred i ≡ just a))) (dom rewards)
    stakeRelation = m ∪ ∣ rewards ∣
```

Figure 72: Functions for computing stake distributions

stake distribution? Right at the moment of producing a block? Some time in the past? How do we deal with the fact that the blockchain is only *eventually consistent*, i.e. blocks can be rolled back before a stable consensus on the chain is formed?

On Cardano, the answer to these questions is to group time into *epochs*. An epoch is long enough such that at the beginning of a new epoch, the beginning of the previous epoch has become stable. An epoch is also long enough for human users to react to parameter changes, such as stake pool costs or performance. But an epoch is also short enough so that changes to the stake distribution will be reflected in block production within a reasonable timeframe.

The rewards for the blocks produced during a given epoch e_i involve the two epochs surrounding it. In particular, the stake distribution will come from the previous epoch and the rewards will be calculated in the following epoch. At each epoch boundary, one snapshot of the stake distribution is taken; changes to the stake distribution within an epoch are not considered until the next snapshot is taken. More concretely:

(A) A stake distribution snapshot is taken at the begining of epoch e_{i-1}.

(B) The randomness for leader election is fixed during epoch e_{i-1}

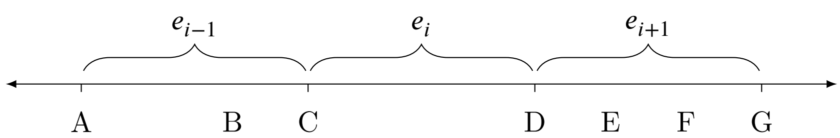
(C) Epoch e_i begins, blocks are produced using the snapshot taken at (A).

(D) Epoch e_i ends. A snapshot is taken of the stake pool performance during epoch e_i. A snapshot is also taken of the fee pot.

(E) The snapshots from (D) are stable and the reward calculation can begin.

(F) The reward calculation is finished and an update to the ledger state is ready to be applied.

(G) Rewards are given out.



In order to specify this logic, we store the last three snapshots of the stake distributions. The mnemonic "mark, set, go" will be used to keep track of the snapshots, where the label "mark" refers to the most recent snapshot, and "go" refers to the snapshot that is ready to be used in the reward calculation.

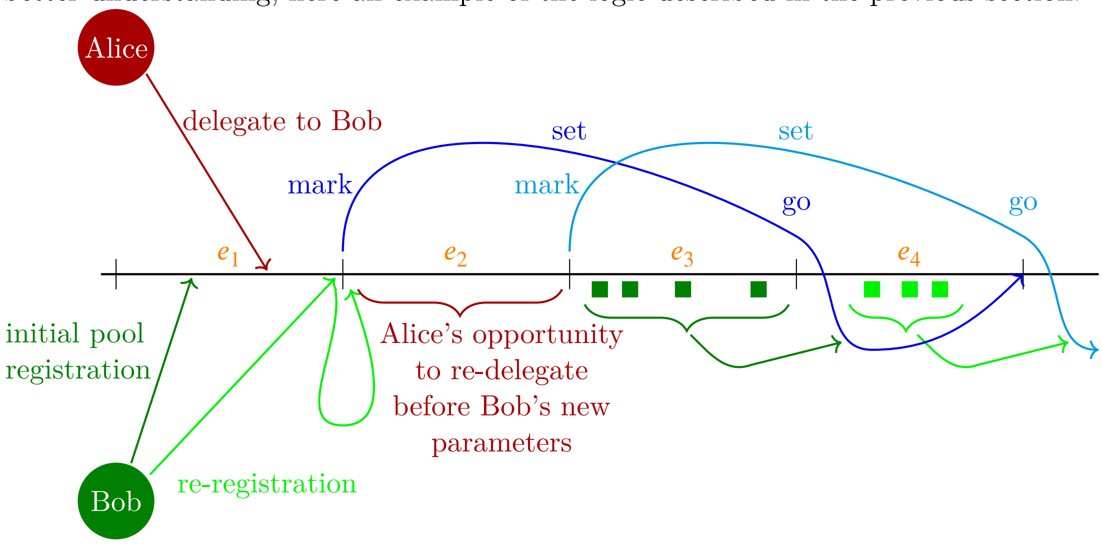
In the above diagram, the snapshot taken at (A) is labeled "mark" during epoch e_{i-1}, "set" during epoch e_i and "go" during epoch e_{i+1}. At (G) the snapshot taken at (A) is no longer needed and will be discarded.

In other words, blocks will be produced using the snapshot labeled "set", whereas rewards are computed from the snapshot labeled "go".

> **NOTE**
>
> Between time D and E we are concerned with chain growth and stability. Therefore this duration can be stated as 2k blocks (to state it in slots requires details about the particular version of the Ouroboros protocol). The duration between F and G is also 2k blocks. Between E and F a single honest block is enough to ensure a random nonce.

18.3.2 Example Illustration of the Reward Cycle

For better understanding, here an example of the logic described in the previous section:



Bob registers his stake pool in epoch e_1. Alice delegates to Bob's stake pool in epoch e_1. Just before the end of epoch e_1, Bob submits a stake pool re-registration, changing his pool parameters. The change in parameters is not immediate, as shown by the curved arrow around the epoch boundary.

A snapshot is taken on the e_1/e_2 boundary. It is labeled "mark" initially. This snapshot includes Alice's delegation to Bob's pool, and Bob's pool parameters and listed in the initial pool registration certificate.

If Alice changes her delegation choice any time during epoch e_2, she will never be effected by Bob's change of parameters.

A new snapshot is taken on the e_2/e_3 boundary. The previous (darker blue) snapshot is now labeled "set", and the new one labeled "mark". The "set" snapshot is used for leader election in epoch e_3.

On the e_3/e_4 boundary, the darker blue snapshot is labeled "go" and the lighter blue snapshot is labeled "set". Bob's stake pool performance during epoch e_3 (he produced 4 blocks) will be used with the darker blue snapshot for the rewards which will be handed out at the beginning of epoch e_5.

18.3.3 Stake Distribution Snapshots

This section defines the SNAP transition rule for stake distribution snapshots.

Fig. 73 defines the type `Snapshots` that contains the data that needs to be saved at the end of an epoch. This data is:

- `mark`, `set`, `go`: Three stake distribution snapshots as explained in Sec. 18.3.1.
- `feeSS`: stores the fees which are added to the reward pot during the next reward update calculation, which is then subtracted from the fee pot on the epoch boundary.

```
record Snapshots : Set where
  field
    mark set go : Snapshot
    feeSS       : Coin
```

Figure 73: Definitions for the SNAP transition system

Fig. 74 defines the snapshot transition rule. This transition has no preconditions and results in the following state change:

- The oldest snapshot is replaced with the penultimate one.
- The penultimate snapshot is replaced with the newest one.
- The newest snapshot is replaced with one just calculated.
- The current fees pot is stored in `feeSS`. Note that this value will not change during the epoch, unlike the `fees` value in the UTxO state.

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

Figure 74: SNAP transition system

19 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


record NewEpochState : Type where
  field
    lastEpoch  : Epoch
    epochState : EpochState
    ru         : Maybe RewardUpdate
```

Figure 75: Definitions for the EPOCH and NEWEPOCH transition systems

Fig. 76 defines the function createRUpd which creates a RewardUpdate, i.e. the net flow of Ada due to paying out rewards after an epoch. Relevant quantities are:

- prevPp: Previous protocol parameters, which correspond to the parameters during the epoch for which we are creating rewards.
- ActiveSlotCoeff: Global constant, equal to the probability that a party holding all the stake will be selected to be a leader for given slot. Equals 1/20 during the Shelley era on the Cardano Mainnet.
- Δr_1: Ada taken out of the reserves for paying rewards, as determined by the monetaryExpansion protocol parameter.
- rewardPot: Total amount of coin available for rewards this epoch, as described in IOHK Formal Methods Team [9, Sec 6.4].
- feeSS: The fee pot which, together with the reserves, funds the rewardPot. We use the fee pot that accumulated during the epoch for which we now compute block production rewards. Note that fees are not explicitly removed from any account: the fees come from transactions paying them and are accounted for whenever transactions are processed.
- Δt_1: The proportion of the reward pot that will move to the treasury, as determined by the treasuryCut protocol parameter. The remaining pot is called the R, just as in IOHK Formal Methods Team [9, Sec 6.5].
- pstakego: Stake distribution used for calculating the rewards. This is the oldest stake distribution snapshot, labeled go.
- rs: The rewards, as calculated by the function reward.
- Δr_2: The difference between the maximal amount of rewards that could have been paid out if pools had been optimal, and the actual rewards paid out. This difference is returned to the reserves.
- $\div_0$: Division operator that returns zero when the denominator is zero.

```
createRUpd : ℕ → BlocksMade → EpochState → Coin → RewardUpdate
createRUpd slotsPerEpoch b es total =
  record { Δt = Δt₁
         ; Δr = 0 - Δr₁ + Δr₂
         ; Δf = 0 - pos feeSS
         ; rs = rs
         }
  where
    prevPp      = PParamsOf es
    reserves    = reservesOf es
    pstakego    = es .EpochState.ss .Snapshots.go
    feeSS       = es .EpochState.ss .Snapshots.feeSS
    stake       = pstakego .Snapshot.stake
    delegs      = pstakego .Snapshot.delegations
    poolParams  = pstakego .Snapshot.poolParameters
    blocksMade  = ∑[ m ← b ] m
    ρ           = fromUnitInterval (prevPp .PParams.monetaryExpansion)
    η           = fromℕ blocksMade ÷₀ (fromℕ slotsPerEpoch * ActiveSlotCoeff)
    Δr₁         = floor (1 ⊓ η * ρ * fromℕ reserves)
    rewardPot   = pos feeSS + Δr₁
    τ           = fromUnitInterval (prevPp .PParams.treasuryCut)
    Δt₁         = floor (fromℤ rewardPot * τ)
    R           = rewardPot - Δt₁
    circulation = total - reserves
    rs          = reward prevPp b (posPart R) poolParams stake delegs circulation
    Δr₂         = R - pos (∑[ c ← rs ] c)
```

Figure 76: RewardUpdate Creation

Fig. 78 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'`.

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

⟦ ⟦ treasury
  , reserves ⟧
, ss
, ⟦ ⟦ utxo
    , fees
    , deposits
    , donations ⟧
  , govSt
  , ⟦ ⟦ voteDelegs
      , stakeDelegs
      , rewards ⟧
    , pState
    , gState ⟧ ⟧
, es
, fut ⟧ =

  ⟦ ⟦ posPart (pos treasury + Δt + pos unregRU')
    , posPart (pos reserves + Δr) ⟧
  , ss
  , ⟦ ⟦ utxo , posPart (pos fees + Δf) , deposits , donations ⟧
    , govSt
    , ⟦ ⟦ voteDelegs , stakeDelegs , rewards ∪⁺ regRU ⟧ , pState , gState ⟧ ⟧
  , es
  , fut ⟧
  where
    open RewardUpdate rewardUpdate using (Δt; Δr; Δf; rs)
    regRU    = rs ∣ dom rewards
    unregRU  = rs ∣ dom rewards ᶜ
    unregRU' = ∑[ x ← unregRU ] x

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

```
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 77: Functions for computing stake distributions

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

```
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 78: EPOCH transition system

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


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

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

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

Figure 79: NEWEPOCH transition system

20 Blockchain Layer

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

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

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

Figure 80: Definitions CHAIN transition system

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

Figure 81: Type of the CHAIN transition system

```
CHAIN : {b : Block} {nes : NewEpochState} {cs : ChainState}
let cs' = record cs { newEpochState
                     = record nes { epochState
                                    = record epochState {ls = ls'} } }
    Γ   = ⟦ slot , ∣ constitution ∣ , ∣ pp ∣ , es , treasuryOf nes ⟧
in
• totalRefScriptsSize ls ts ≤ maxRefScriptSizePerBlock
• _ ⊢ newEpochState ⇀⦇ epoch slot ,NEWEPOCH⦈ nes
• Γ ⊢ ls ⇀⦇ ts ,LEDGERS⦈ ls'
  ────────────────────────────────────
  _ ⊢ cs ⇀⦇ b ,CHAIN⦈ cs'
```

Figure 82: CHAIN transition system

21 Properties

This section presents the properties of the ledger that we have formally proved in Agda or plan to do so in the near future. We indicate in which Agda module each property is formally stated and (possibly) proved. A "Claim" is a property that is not yet proved, while a "Theorem" is one for which we have a formal proof.

21.1 Preservation of Value

Theorem 1. **(`Conway/Ledger/Properties/PoV`: `LEDGER` rule preserves value)**

- *Informally.* Let `s`, `s'` : `LState` be ledger states and let `tx` : `Tx` be a *fresh* transaction, that is, a transaction that is not already part of the `UTxOState` of `s`. If `s ⇀⦇ tx ,LEDGER⦈ s'`, then the coin values of `s` and `s'` are equal, that is, `getCoin s` ≡ `getCoin s'`.
- *Formally.*

```
LEDGER-pov : {Γ : LEnv} {s s' : LState}
  → txid ∉ mapˢ proj₁ (dom (UTxOOf s))
  → Γ ⊢ s ⇀⦇ tx ,LEDGER⦈ s' → getCoin s ≡ getCoin s'
```

- *Proof.* See the `Conway/Ledger/Properties/PoV` module in the formal ledger repository.

Lemma 2. *(`Conway/Utxo/Properties/PoV`: `UTXO` rule preserves value)*

- *Informally.* Let `s` and `s'` be `UTxOState`s, let `tx` : `Tx` be a fresh transaction with withdrawals `txwdrls`, and suppose `s ⇀⦇ tx ,UTXO⦈ s'`. If `tx` is valid, then the coin value of `s'` is equal to the sum of the coin values of `s` and `txwdrls`. If `tx` is not valid, then the coin values of `s` and `s'` are equal. We can express this concisely as follows:

  $$\texttt{getCoin}\ s + \texttt{getCoin}\ txwdrls \cdot \chi(tx\ .\texttt{isValid}) \equiv \texttt{getCoin}\ s',$$

  where χ : *Bool* → 0, 1 is the *characteristic function*, which returns 0 for false and 1 for true.
- *Formally.*

```
UTXOpov : {Γ : UTxOEnv} {tx : Tx} {s s' : UTxOState}
  → txidOf tx ∉ mapˢ proj₁ (dom (UTxOOf s))
  → Γ ⊢ s ⇀⦇ tx ,UTXO⦈ s'
  → getCoin s + getCoin (wdrlsOf tx) * χ (tx .isValid) ≡ getCoin s'
```

- *Proof.* See the `Conway/Utxo/Properties/PoV` module in the formal ledger repository.

Theorem 3. **(`Conway/Certs/Properties/PoV`: `CERTS` rule preserves value)**

- *Informally.* Let `l` be a list of `DCert`s, and let `s₁`, `sₙ` be `CertState`s such that `s₁ ⇀⦇ l ,CERTS⦈ sₙ`. Then, the value of `s₁` is equal to the value of `sₙ` plus the value of the withdrawals in `Γ`.
- *Formally.*

```
CERTS-pov : {Γ : CertEnv} {s₁ sₙ : CertState}
  → ∀[ a ∈ dom (CertEnv.wdrls Γ) ] NetworkIdOf a ≡ NetworkId
  → Γ ⊢ s₁ ⇀⦇ l ,CERTS⦈ sₙ
  → getCoin s₁ ≡ getCoin sₙ + getCoin (wdrlsOf Γ)
```

- *Proof.* See the `Conway/Certs/Properties/PoV` module in the formal ledger repository.

Lemma 4. (`Conway/Certs/Properties/PoVLemmas`*:* `CERT` *rule preserves value)*

- *Informally.* Let `s`, `s'` be `CertState`s such that `s ⇀⦇ dcert ,CERT⦈ s'` for some `dcert : DCert`. Then, `getCoin s ≡ getCoin s'`.
- *Formally.*

```
CERT-pov : {Γ : CertEnv} {s s' : CertState}
  → Γ ⊢ s ⇀⦇ dCert ,CERT⦈ s'
  → getCoin s ≡ getCoin s'
```

- *Proof.* See the `Conway/Certs/Properties/PoVLemmas` module in the formal ledger repository.

Lemma 5. (`Conway/Certs/Properties/PoVLemmas`*:* `CERTBASE` *rule preserves value)*

- *Informally.* Let `Γ : CertEnv` be a certificate environment, and let `s`, `s' : CertState` be certificate states such that `s ⇀⦇ _ ,CERTBASE⦈ s'`. Then, the value of `s` is equal to the value of `s'` plus the value of the withdrawals in `Γ`. In other terms,

  `getCoin s ≡ getCoin s' + getCoin (Γ .wdrls )`.
- *Formally.*

```
CERTBASE-pov : {Γ : CertEnv} {s s' : CertState}
  → ∀[ a ∈ dom (CertEnv.wdrls Γ) ] NetworkIdOf a ≡ NetworkId
  → Γ ⊢ s ⇀⦇ _ ,CERTBASE⦈ s'
  → getCoin s ≡ getCoin s' + getCoin (CertEnv.wdrls Γ)
```

- *Proof.* See the `Conway/Certs/Properties/PoVLemmas` module in the formal ledger repository.

Lemma 6. (`Conway/Certs/Properties/PoVLemmas`*: iteration of* `CERT` *rule preserves value)*

- *Informally.* Let `l` be a list of `DCert`s, and let s_1, s_n be `CertState`s such that, starting with s_1 and successively applying the `CERT` rule to with `DCert`s from the list `l`, we obtain s_n. Then, the value of s_1 is equal to the value of s_n.
- *Formally.*

```
sts-pov : {Γ : CertEnv} {s₁ sₙ : CertState}
  → ReflexiveTransitiveClosure {sts = _⊢_⇀⦇_,CERT⦈_} Γ s₁ l sₙ
  → getCoin s₁ ≡ getCoin sₙ
```

- *Proof.* See the `Conway/Certs/Properties/PoVLemmas` module in the formal ledger repository.

21.2 Invariance Properties

To say that a predicate `P` is an *invariant* of a transition rule means the following: if the transition rule relates states `s` and `s'` and if `P` holds at state `s`, then `P` holds at state `s'`.

Claim 7. (`Conway/Chain/Properties/CredDepsEqualDomRwds`*: Equality of credential depsoits is a* `CHAIN` *invariant)*

- *Informally.* This property concerns two quantities associated with a given `ChainState` *cs*,
  - the credential deposits of the `UTxOState` of *cs* and
  - the credential deposits of the rewards in the ledger state of *cs*.

  The predicate `credDeposits≡dom-rwds` *cs* asserts that these quantities are equal for *cs*. Formally,

  ```
  credDeposits≡dom-rwds : ChainState → Type
  credDeposits≡dom-rwds cs = filter isCredDeposit (dom (DepositsOf cs))
                                ≡ map CredentialDeposit (dom (RewardsOf cs))
  ```

  The property `credDeposits≡dom-rwds-inv` asserts that `credDeposits≡dom-rwds` is a chain invariant. That is, if *cs* and *cs'* are two `ChainState`s such that `cs ⇀⦇ tx ,CHAIN⦈ cs'`, then `credDeposits≡dom-rwds cs` only if `credDeposits≡dom-rwds cs'`.

- *Formally.*

  ```
  credDeposits≡dom-rwds-inv : Type
  credDeposits≡dom-rwds-inv = LedgerInvariant _⊢_⇀⦇_,CHAIN⦈_ credDeposits≡dom-rwds
  ```

- *Proof.* *To appear* (in the `Conway/Chain/Properties/CredDepsEqualDomRwds` module of the formal ledger repository).

Claim 8. (`Conway/Chain/Properties/PParamsWellFormed`: Well-formedness of `PParams` is a `CHAIN` invariant)

- *Informally.* We say the `PParams` of a chain state are *well-formed* if each of the following parameters is non-zero: maxBlockSize, maxTxSize, maxHeaderSize, maxValSize, refScriptCostStride, coinsPerUTxOByte, poolDeposit, collateralPercentage, ccMaxTermLength, govActionLifetime, govActionDeposit, drepDeposit. Formally,

  ```
  pp-wellFormed : ChainState → Type
  pp-wellFormed = paramsWellFormed ∘ PParamsOf
  ```

  This property asserts that `pp-wellFormed` is a chain invariant. That is, if *cs* and *cs'* are chain states such that `cs ⇀⦇ tx ,CHAIN⦈ cs'`, and if the `PParams` of *cs* are well-formed, then so are the `PParams` of *cs'*.

- *Formally.*

  ```
  pp-wellFormed-invariant : Type
  pp-wellFormed-invariant = LedgerInvariant _⊢_⇀⦇_,CHAIN⦈_ pp-wellFormed
  ```

- *Proof.* *To appear* (in the `Conway/Chain/Properties/PParamsWellFormed` module of the formal ledger repository).

21.2.1 Governance Action Deposits Match

Theorems 9 to 11 assert that a certain predicate is an invariant of the `CHAIN`, `LEDGER`, and `EPOCH` rules, respectively. Given a ledger state *s*, we focus on deposits in the `UTxOState` of *s* that are `GovActionDeposit`s and we compare that set of deposits with the `GovActionDeposit`s of the `GovState` of *s*. When these two sets are the same, we write `govDepsMatch` *s* and say the `govDepsMatch` relation holds for *s*. Formally, the `govDepsMatch` predicate is defined as follows:

```
govDepsMatch : LState → Type
govDepsMatch ls =
  filterˢ isGADeposit (dom (DepositsOf ls)) ≡ᵉ fromList (dpMap (GovStateOf ls))
```

The assertion, "the `govDepsMatch` relation is an invariant of the `LEDGER` rule," means the following: if `govDepsMatch` *s* and *s* `⇀⦇` *tx* `,LEDGER⦈` *s'*, then `govDepsMatch` *s'*.

Theorem 9. (`Conway/Chain/Properties/GovDepsMatch`: `govDepsMatch` is invariant of `CHAIN` rule)

- *Informally.* Fix a `Block` *b*, a `ChainState` *cs*, and a `NewEpochState` *nes*. Let `csLState` be the ledger state of *cs*. Recall, a `ChainState` has just one field, `newEpochState` : `NewEpochState`. Consider the chain state `cs'` defined as follows:

  ```
  cs' : ChainState
  cs' .newEpochState =
    record { lastEpoch  = nes .lastEpoch
           ; epochState = record (EpochStateOf cs) {ls = LStateOf nes}
           ; ru         = nes .ru }
  ```

  That is `cs'` is essentially *nes*, but the `EpochState` field is set to the `epochState` of *cs* with the exception of the `LState` field, which is set to that of *nes*.

  Let *utxoSt* and *utxoSt'* be the respective `UTxOState`s of the ledger states of *cs* and `cs'`, respectively, and let *govSt* and *govSt'* be their respective `GovState`s.

  Assume the following conditions hold:

  - let `removed'` : $\mathbb{P}$(`GovActionID` × `GovActionState`) be the union of
    - the governance actions in the `removed` field of the ratify state of *eps*, and
    - the orphaned governance actions in the `GovState` of *eps*.

    Let $\mathscr{G}$ be the set {`GovActionDeposit` *id* : *id* ∈ proj_1 `removed'`}. $\mathscr{G}$ is a subset of the set of deposits of the chain state *cs*; that is,

    `map (GovActionDeposit ∘ proj₁) removed' ⊆ dom (DepositsOf cs)`;
  - the total reference script size of `csLState` is not greater than the maximum allowed size per block (as specified in `PParams`);
  - *cs* `⇀⦇` *b* `,CHAIN⦈` `cs'`.

  Under these conditions, if the governance action deposits of *utxoSt* equal those of *govSt*, then the same holds for *utxoSt'* and *govSt'*. In other terms, `govDepsMatch csLState` implies `govDepsMatch nesState`.

- *Formally.*

  ```
  CHAIN-govDepsMatch :
    map (GovActionDeposit ∘ proj₁) removed' ⊆ dom (DepositsOf cs)
    → totalRefScriptsSize csLState ts ≤ maxRefScriptSizePerBlock
    → _ ⊢ cs ⇀⦇ b ,CHAIN⦈ cs'
    → govDepsMatch csLState → govDepsMatch (LStateOf nes)
  ```

- *Proof.* See the `Conway/Chain/Properties/GovDepsMatch` module in the formal ledger repository.

Lemma 10. (Conway/Ledger/Properties/GovDepsMatch: govDepsMatch is invariant of LEDGER rule)

- *Informally.* Suppose `s`, `s'` are ledger states such that `s ⇀⦇ tx ,LEDGER⦈ s'`. Let `utxoSt` and `utxoSt'` be their respective `UTxOState`s and let `govSt` and `govSt'` be their respective `GovStates`. If the governance action deposits of `utxoSt` are equal those of `govSt`, then the same holds for `utxoSt'` and `govSt'`. In other terms, if `govDepsMatch s`, then `govDepsMatch s'`.
- *Formally.*

```
LEDGER-govDepsMatch : LedgerInvariant _⊢_⇀⦇_,LEDGER⦈_ govDepsMatch
```

- *Proof.* See the Conway/Ledger/Properties/GovDepsMatch module in the formal ledger repository.

Lemma 11. (Conway/Epoch/Properties/GovDepsMatch: govDepsMatch is invariant of EPOCH rule)

- *Informally.* Let `eps`, `eps'` : `EpochState` be two epoch states and let `e` : `Epoch` be an epoch. Recall, `eps .ls` denotes the ledger state of `eps`. If `eps ⇀⦇ e ,EPOCH⦈ eps'`, then (under a certain special condition) `govDepsMatch (eps .ls)` implies `govDepsMatch (eps' .ls)`.

  The special condition under which the property holds is the same as the one in Theorem 9: let `removed'` be the union of the governance actions in the `removed` field of the ratify state of `eps` and the orphaned governance actions in the `GovState` of `eps`. Let $\mathscr{G}$ be the set {`GovActionDeposit id : id` ∈ proj_1 `removed'`}. Assume: $\mathscr{G}$ is a subset of the set of deposits of (the governance state of) `eps`.
- *Formally.*

```
EPOCH-govDepsMatch : {eps' : EpochState} {e : Epoch}
  → map (GovActionDeposit ∘ proj₁) removed' ⊆ dom (DepositsOf eps)
  → _ ⊢ eps ⇀⦇ e ,EPOCH⦈ eps'
  → govDepsMatch (eps .ls) → govDepsMatch (eps' .ls)
```

- *Proof.* See the Conway/Epoch/Properties/GovDepsMatch module in the formal ledger repository.

21.3 Minimum Spending Conditions

Theorem 12. (Conway/Utxo/Properties/MinSpend: general spend lower bound)

- *Informally.* Let `tx` : `Tx` be a valid transaction and let `txcerts` be its list of `DCert`s. Denote by `noRefundCert txcerts` the assertion that no element in `txcerts` is one of the two refund types (i.e., an element of `l` is neither a `dereg` nor a `deregdrep`).

  Let `s`, `s'` : `UTxOState` be two UTxO states. If `s ⇀⦇ tx ,UTXO⦈ s'` and if `noRefundCert txcerts`, then the coin consumed by `tx` is at least the sum of the governance action deposits of the proposals in `tx`.
- *Formally.*

```
utxoMinSpend : {Γ : UTxOEnv} {tx : Tx} {s s' : UTxOState}
  → Γ ⊢ s ⇀⦇ tx ,UTXO⦈ s'
  → noRefundCert (txcertsOf tx)
  → coin (consumed _ s (TxBodyOf tx)) ≥ length (txpropOf tx) * govActionDepositOf Γ
```

- *Proof.* See the Conway/Utxo/Properties/MinSpend module in the formal ledger repository.

Theorem 13. **(Conway/Utxo/Properties/MinSpend: spend lower bound for proposals)**

- *Preliminary remarks.*

  1. Define noRefundCert *l* and *pp* as in Theorem 12.
  2. Given a ledger state *ls* and a transaction *tx*, denote by validTxIn₂ *tx* the assertion that there exists ledger state *ls'* such that *ls* ⇀⦇ *tx* ,LEDGER⦈ *ls'*.
  3. Assume the following additive property of the ∪⁺ operator holds:

```
∑[ x ← d₁ ∪⁺ d₂ ] x ≡ ∑[ x ← d₁ ] x ◇ ∑[ x ← d₂ ] x
```

- *Informally.* Let *tx* : Tx be a valid transaction and let *cs* : ChainState be a chain state. If the condition validTxIn₂ *tx* (described above) holds, then the coin consumed by *tx* is at least the sum of the governance action deposits of the proposals in *tx*.
- *Formally.*

```
propose-minSpend : {slot : Slot} {tx : Tx} {cs : ChainState}
                   ( let pp     = PParamsOf cs
                         utxoSt = UTxOStateOf cs )
  → noRefundCert txcerts
  → validTxIn₂ cs slot tx
  → coin (consumed pp utxoSt body) ≥ length txprop * pp .govActionDeposit
```

- *Proof.* See the Conway/Utxo/Properties/MinSpend module in the formal ledger repository.

21.4 Miscellaneous Properties

Claim 14. (Conway/GovernanceActions/Properties/ChangePPGroup: PParam updates have non-empty groups)

- *Informally.* Let *p* : GovProposal be a governance proposal and suppose the GovActionType of *p* .action is ChangePParams. If the data field of *p*—that is *pu* = *p* .action .gaData—is denoted by *pu* ("parameter update"), then the set updateGroups *pu* is nonempty.
- *Formally.*

```
ChangePPHasGroup : {tx : Tx} {p : GovProposal} (pu : PParamsUpdate)
  → p ∈ Tx.body tx → p .GovProposal.action ≡ ⟦ ChangePParams , pu ⟧ᵍᵃ
  → Type
ChangePPHasGroup pu _ _ = updateGroups pu ≢ ∅
```

- *Proof.* *To appear* (in the Conway/GovernanceActions/Properties/ChangePPGroup module of the formal ledger repository).

Claim 15. (Conway/Chain/Properties/EpochStep: New enact state only if new epoch)

- *Informally.* Let *cs* and *cs'* be ChainStates and *b* a Block. If *cs* ⇀⦇ *b* ,CHAIN⦈ *cs'* and if the enact states of *cs* and *cs'* differ, then the epoch of the slot of *b* is the successor of the last epoch of *cs*.

- *Formally.*

```
enact-change⇒newEpoch : (b : Block) {cs cs' : ChainState}
  → _ ⊢ cs ⇀⦇ b ,CHAIN⦈ cs' → EnactStateOf cs ≢ EnactStateOf cs'
  → Type

enact-change⇒newEpoch b {cs} h es≢es' = epoch (b .slot) ≡ sucᵉ (LastEpochOf cs)
```

- *Proof. To appear* (in the Conway/Chain/Properties/EpochStep module of the formal ledger repository).

Claim 16. (Conway/Epoch/Properties/ConstRwds: NEWEPOCH rule leaves rewards unchanged)

- *Informally.* Rewards are left unchanged by the NEWEPOCH rule. That is, if es and es' are two NewEpochStates such that es ⇀⦇ e ,NEWEPOCH⦈ es', then the rewards of es and es' are equal.
- *Formally.*

```
dom-rwds-const : {e : Epoch} (es es' : NewEpochState)
  → _ ⊢ es ⇀⦇ e ,NEWEPOCH⦈ es' → Type

dom-rwds-const es es' step = dom (RewardsOf es) ≡ dom (RewardsOf es')
```

- *Proof. To appear* (in the Conway/Epoch/Properties/ConstRwds module of the formal ledger repository).

Claim 17. (Conway/Epoch/Properties/NoPropSameDReps: DReps unchanged if no gov proposals)

- *Informally.* If there are no governance proposals in the GovState of es, then the activeDReps of es in Epoch e are the same as the activeDReps of es' in the next epoch.
- *Formally.*

```
prop≡∅⇒activeDReps-const : Epoch → (es es' : NewEpochState) → Type
prop≡∅⇒activeDReps-const e es es' =
  GovStateOf es ≡ [] → activeDReps e es ≡ᵉ activeDReps (sucᵉ e) es'
```

- *Proof. To appear* (in the Conway/Epoch/Properties/NoPropSameDReps module of the formal ledger repository).

Claim 18. (Conway/Certs/Properties/VoteDelegsVDeleg: voteDelegs by credVoter constructor)

- *Informally.* A CertState has a DState, PState, and a GState. The DState contains a field voteDelegs which is a mapping from Credential to VDeleg.

  VDeleg is a datatype with three constructors; the one of interest to us here is credVoter, which takes two arguments, a GovRole and a Credential.

  Now suppose we have a collection C of credentials—for instance, given d : DState, take C to be the domain of the voteDelegs field of d. We could then obtain a set of VDelegs by applying credVoter DRep to each element of C.

  The present property asserts that the set of VDelegs that results from the application of credVoter DRep to the domain of the voteDelegs of d contains the range of the voteDelegs of d.

- *Formally.*

```
voteDelegsVDeleg : DState → Type
voteDelegsVDeleg d = range (voteDelegsOf d) ⊆ mapˢ (credVoter DRep) (dom (voteDelegsOf d))
```

- *Proof.* To appear in the `Conway/Certs/Properties/VoteDelegsVDeleg` module in the formal ledger repository.

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.

[10] "Cardano Docs," Accessed: Apr. 1, 2025. [Online]. Available: https://docs.cardano.org.

[11] E. Garrido and the Cardano community. "(re)introduction to cardano," Accessed: Mar. 12, 2025. [Online]. Available: https://developers.cardano.org/docs/operate-a-stake-pool/introduction-to-cardano/.

[12] O. Hryniuk. "Ouroboros chronos provides the first high-resilience, cryptographic time source based on blockchain technology," Accessed: Mar. 12, 2025. [Online]. Available: https://iohk.io/en/blog/posts/2021/10/27/ouroboros-chronos-provides-the-first-high-resilience-cryptographic-time-source-based-on-blockchain/.

[13] "Cardano Staking: How To Stake ADA," Accessed: Mar. 12, 2025. [Online]. Available: https://www.ledger.com/academy/cardano-staking-how-to-stake-ada.

[14] "Cardano Glossary," Accessed: Apr. 1, 2025. [Online]. Available: https://cardano.org/docs/glossary#cardano-glossary.

[15] "Crypto glossary," Accessed: Mar. 12, 2025. [Online]. Available: https://www.ledger.com/academy/glossary.

[16] "Glossary," Accessed: Mar. 12, 2025. [Online]. Available: https://www.essentialcardano.io/glossary?sort=alphabetical.

[17] "Time handling on Cardano," Accessed: Mar. 12, 2025. [Online]. Available: https://docs.cardano.org/about-cardano/explore-more/time.

A Definitions

To keep this document somewhat self-contained, we define some technical terms that arise when defining and describing the Cardano ledger. This is not meant to be comprehensive and the reader may wish to consult online resources to fill in any gaps. Here are a few such resources that might be helpful.

- Cardano Docs [10];
- (Re)introduction to Cardano [11];
- Ouroboros Chronos blog post [12];
- Cardano Staking: How To Stake ADA [13];
- Glossary from cardano.org [14];
- Glossary from ledger.com [15];
- Glossary from essentialcardano.io [16].

A.1 Cardano Time Handling

For more details, see the Time handling on Cardano section of [17].

In Cardano, the Ouroboros proof-of-stake (PoS) consensus protocol models the passage of physical time as an infinite sequence of time slots and epochs.

block time The actual time interval between blocks, or *block time*, is the slot length (in seconds) divided by the block coefficient f, which is the expected block frequency (blocks per second). For example, if f is 0.05, then on average 5% of slots contain blocks. If the slot length is 1 second, then the block time is 20 seconds.

epoch An *epoch* is a period of time, containing some number of slots, used to select block-producing nodes. For example, in Shelley and later eras, an epoch consists of roughly 432,000 slots (or five days if we assume a slot length of 1 second).

genesis block The *genesis block* of Cardano was created on the 23rd of September 2017. As the first block in the blockchain, it set the foundation for the network, it does not reference any previous blocks, and it generated the initial supply of Ada.

slot A *slot* is a discrete time interval in which a single block may be produced; it is the fundamental time unit within the blockchain's consensus protocol. Slots should be long enough for a new block to have a good chance to reach the next slot leader in time. For example, the slot length in the Byron era was 20 seconds, while in Shelley and later eras it is 1 second. Not every slot results in a new block. Indeed, in any given slot, one or more block-producing nodes are nominated (probabilistically based on stake distribution) to be *slot leaders* and given the opportunity to produce a new block. For example, in Shelley and later eras, on average only 0.05 of slots will produce a block (resulting in 20-second intervals between blocks). *Slot number* may refer to a slot's position within the current epoch or it may mean the absolute slot count since the genesis block. The context should make clear which meaning is intended.

The parameter values mentioned in the examples above,

- block time = 20 seconds,
- slot length = 1 second,
- block coefficient = 0.05,
- slots/epoch = 432,000,

are unlikely to change in the short-term. However, the longer term plan is to replace the current Ouroboros protocol with Ouroboros Chronos, which addresses timekeeping challenges by providing the first high-resilience cryptographic time source based on blockchain technology (see Hryniuk [12]).

B 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].

B.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.

C 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.

D 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.