2.2. Transaction Function

certificate.ak

use aiken/hash.{Blake2b\_224, Hash}

use aiken/transaction/credential.{PoolId, StakeCredential, VerificationKey}

/// An on-chain certificate attesting of some operation. Publishing

/// certificates / triggers different kind of rules; most of the time,

/// they require signatures from / specific keys.

pub type Certificate {

CredentialRegistration { delegator: StakeCredential }

CredentialDeregistration { delegator: StakeCredential }

CredentialDelegation { delegator: StakeCredential, delegatee: PoolId }

PoolRegistration { pool\_id: PoolId, vrf: Hash<Blake2b\_224, VerificationKey> }

PoolDeregistration { pool\_id: PoolId, epoch: Int }

Governance

TreasuryMovement

}

credential.ak

use aiken/builtin

use aiken/hash.{Blake2b\_224, Hash}

/// A general structure for representing an on-chain `Credential`.

///

/// Credentials are always one of two kinds: a direct public/private key

/// pair, or a script (native or Plutus).

pub type Credential {

VerificationKeyCredential(Hash<Blake2b\_224, VerificationKey>)

ScriptCredential(Hash<Blake2b\_224, Script>)

}

/// A Cardano `Address` typically holding one or two credential references.

///

/// Note that legacy bootstrap addresses (a.k.a. 'Byron addresses') are

/// completely excluded from Plutus contexts. Thus, from an on-chain

/// perspective only exists addresses of type 00, 01, ..., 07 as detailed

/// in [CIP-0019 :: Shelley Addresses](https://github.com/cardano-foundation/CIPs/tree/master/CIP-0019/#shelley-addresses).

pub type Address {

payment\_credential: PaymentCredential,

stake\_credential: Option<StakeCredential>,

}

/// Smart-constructor for an [Address](#Address) from a [verification key](#VerificationKey) hash. The resulting address has no delegation rights whatsoever.

pub fn from\_verification\_key(vk: Hash<Blake2b\_224, VerificationKey>) -> Address {

Address {

payment\_credential: VerificationKeyCredential(vk),

stake\_credential: None,

}

}

/// Smart-constructor for an [Address](#Address) from a [script](#Script) hash. The address has no delegation rights whatsoever.

pub fn from\_script(script: Hash<Blake2b\_224, Script>) -> Address {

Address {

payment\_credential: ScriptCredential(script),

stake\_credential: None,

}

}

/// Set (or reset) the delegation part of an [Address](#Address) using a [verification key](#VerificationKey) hash. This is useful when combined with [`from\_verification\_key`](#from\_verification\_key) and/or [`from\_script`](#from\_script).

pub fn with\_delegation\_key(

self: Address,

vk: Hash<Blake2b\_224, VerificationKey>,

) -> Address {

Address {

payment\_credential: self.payment\_credential,

stake\_credential: Some(Inline(VerificationKeyCredential(vk))),

}

}

/// Set (or reset) the delegation part of an [Address](#Address) using a [script](#Script) hash. This is useful when combined with [`from\_verification\_key`](#from\_verification\_key) and/or [`from\_script`](#from\_script).

pub fn with\_delegation\_script(

self: Address,

script: Hash<Blake2b\_224, Script>,

) -> Address {

Address {

payment\_credential: self.payment\_credential,

stake\_credential: Some(Inline(ScriptCredential(script))),

}

}

/// Represent a type of object that can be represented either inline (by hash)

/// or via a reference (i.e. a pointer to an on-chain location).

///

/// This is mainly use for capturing pointers to a stake credential

/// registration certificate in the case of so-called pointer addresses.

pub type Referenced<a> {

Inline(a)

Pointer { slot\_number: Int, transaction\_index: Int, certificate\_index: Int }

}

pub type VerificationKey =

ByteArray

pub type Script =

ByteArray

pub type Signature =

ByteArray

/// Verify an Ed25519 signature using the given verification key.

/// Returns `True` when the signature is valid.

pub fn verify\_signature(

key: VerificationKey,

msg: ByteArray,

sig: Signature,

) -> Bool {

builtin.verify\_ed25519\_signature(key, msg, sig)

}

/// A `StakeCredential` represents the delegation and rewards withdrawal conditions

/// associated with some stake address / account.

///

/// A `StakeCredential` is either provided inline, or, by reference using an

/// on-chain pointer.

///

/// Read more about pointers in [CIP-0019 :: Pointers](https://github.com/cardano-foundation/CIPs/tree/master/CIP-0019/#pointers).

pub type StakeCredential =

Referenced<Credential>

/// A 'PaymentCredential' represents the spending conditions associated with

/// some output. Hence,

///

/// - a `VerificationKeyCredential` captures an output locked by a public/private key pair;

/// - and a `ScriptCredential` captures an output locked by a native or Plutus script.

///

pub type PaymentCredential =

Credential

/// A unique stake pool identifier, as a hash of its owner verification key.

pub type PoolId =

Hash<Blake2b\_224, VerificationKey>

value.ak

use aiken/dict.{Dict, from\_ascending\_pairs\_with}

use aiken/hash.{Blake2b\_224, Hash}

use aiken/list

use aiken/option

use aiken/transaction/credential.{Script}

/// A type-alias for a `PolicyId`. A `PolicyId` is always 28-byte long

pub type PolicyId =

Hash<Blake2b\_224, Script>

/// Ada, the native currency, isn't associated with any `PolicyId` (it's not

/// possible to mint Ada!).

///

/// By convention, it is an empty `ByteArray`.

pub const ada\_policy\_id = #""

/// A type-alias for 'AssetName`, which are free-form byte-arrays between

/// 0 and 32 bytes.

pub type AssetName =

ByteArray

/// Ada, the native currency, isn't associated with any `AssetName` (it's not

/// possible to mint Ada!).

///

/// By convention, it is an empty `ByteArray`.

pub const ada\_asset\_name = #""

/// A multi-asset output `Value`. Contains tokens indexed by [PolicyId](#PolicyId) and [AssetName](#AssetName).

///

/// This type maintain some invariants by construction; in particular, a `Value` will never contain a

/// zero quantity of a particular token.

pub opaque type Value {

inner: Dict<PolicyId, Dict<AssetName, Int>>,

}

/// Construct an empty `Value` with nothing in it.

pub fn zero() -> Value {

Value { inner: dict.new() }

}

/// Check is a `Value` is zero. That is, it has no assets and holds no Ada/Lovelace.

pub fn is\_zero(self: Value) -> Bool {

self == zero()

}

/// Construct a `Value` from an asset identifier (i.e. `PolicyId` + `AssetName`)

/// and a given quantity.

pub fn from\_asset(

policy\_id: PolicyId,

asset\_name: AssetName,

quantity: Int,

) -> Value {

if quantity == 0 {

dict.new()

|> Value

} else {

let asset =

dict.new()

|> dict.insert(asset\_name, quantity)

dict.new()

|> dict.insert(policy\_id, asset)

|> Value

}

}

/// Construct a `Value` from a lovelace quantity.

///

/// Friendly reminder: 1 Ada = 1.000.000 Lovelace

pub fn from\_lovelace(quantity: Int) -> Value {

from\_asset(ada\_policy\_id, ada\_asset\_name, quantity)

}

/// Get a `Value` excluding Ada.

pub fn without\_lovelace(self: Value) -> Value {

dict.delete(self.inner, ada\_policy\_id)

|> Value

}

test without\_lovelace\_1() {

let v = from\_lovelace(1000000)

without\_lovelace(v) == zero()

}

test without\_lovelace\_2() {

let v = from\_lovelace(1000000)

let v2 = from\_lovelace(50000000)

without\_lovelace(v) == without\_lovelace(v2)

}

test without\_lovelace\_3() {

let v =

from\_asset(#"010203", #"040506", 100)

|> add(ada\_policy\_id, ada\_asset\_name, 100000000)

let v2 = from\_asset(#"010203", #"040506", 100)

without\_lovelace(v) == without\_lovelace(v2) && without\_lovelace(v) == v2

}

/// Negates quantities of all tokens (including Ada) in that `Value`.

///

/// ```

/// v1

/// |> value.negate

/// |> value.merge(v1)

/// |> value.is\_zero

/// // True

/// ```

pub fn negate(self: Value) -> Value {

dict.map(self.inner, fn(\_, a) { dict.map(a, fn(\_, q) { 0 - q }) })

|> Value

}

/// Combine two `Value` together.

pub fn merge(left v0: Value, right v1: Value) -> Value {

Value(

dict.union\_with(

v0.inner,

v1.inner,

fn(\_, a0, a1) {

let result =

dict.union\_with(

a0,

a1,

fn(\_, q0, q1) {

let q = q0 + q1

if q == 0 {

None

} else {

Some(q)

}

},

)

if dict.is\_empty(result) {

None

} else {

Some(result)

}

},

),

)

}

test merge\_1() {

let v1 = from\_lovelace(1)

let v2 = from\_lovelace(-1)

merge(v1, v2) == zero()

}

test merge\_2() {

let v1 = from\_asset(#"00", #"", 1)

let v2 = from\_asset(#"01", #"", 2)

let v3 = from\_asset(#"02", #"", 3)

let v =

from\_lovelace(42)

|> merge(v3)

|> merge(v1)

|> merge(v2)

flatten(v) == [

(#"", #"", 42),

(#"00", #"", 1),

(#"01", #"", 2),

(#"02", #"", 3),

]

}

test merge\_3() {

let v1 = from\_asset(#"00", #"", 1)

let v2 = from\_asset(#"00", #"", -1)

let v3 = from\_asset(#"01", #"", 1)

let v =

zero()

|> merge(v1)

|> merge(v2)

|> merge(v3)

flatten(v) == [(#"01", #"", 1)]

}

test merge\_4() {

let v1 = from\_asset(#"00", #"", 1)

let v2 = from\_asset(#"00", #"", -1)

merge(v1, v2) == zero()

}

test merge\_5() {

let v =

zero()

|> add(#"acab", #"beef", 0)

merge(zero(), v) == zero()

}

/// Add a (positive or negative) quantity of a single token to a value.

/// This is more efficient than [`merge`](#merge) for a single asset.

pub fn add(

self: Value,

policy\_id: PolicyId,

asset\_name: AssetName,

quantity: Int,

) -> Value {

if quantity == 0 {

self

} else {

let helper =

fn(\_, left, \_right) {

let inner\_result =

dict.insert\_with(

left,

asset\_name,

quantity,

fn(\_k, ql, qr) {

let q = ql + qr

if q == 0 {

None

} else {

Some(q)

}

},

)

if dict.is\_empty(inner\_result) {

None

} else {

Some(inner\_result)

}

}

Value(

dict.insert\_with(

self.inner,

policy\_id,

dict.from\_ascending\_pairs([Pair(asset\_name, quantity)]),

helper,

),

)

}

}

test add\_1() {

let v =

zero()

|> add(#"acab", #"beef", 321)

|> add(#"acab", #"beef", -321)

v == zero()

}

test add\_2() {

let v =

from\_lovelace(123)

|> add(#"acab", #"beef", 321)

|> add(#"acab", #"beef", -1 \* 321)

v == from\_lovelace(123)

}

test add\_3() {

let v =

from\_lovelace(1)

|> add(ada\_policy\_id, ada\_asset\_name, 2)

|> add(ada\_policy\_id, ada\_asset\_name, 3)

v == from\_lovelace(6)

}

test add\_4() {

let v =

zero()

|> add(#"acab", #"beef", 0)

v == zero()

}

test add\_5() {

let v =

zero()

|> add(#"acab", #"beef", 0)

|> add(#"acab", #"beef", 0)

v == zero()

}

/// Extract the quantity of a given asset.

pub fn quantity\_of(

self: Value,

policy\_id: PolicyId,

asset\_name: AssetName,

) -> Int {

self.inner

|> dict.get(policy\_id)

|> option.and\_then(dict.get(\_, asset\_name))

|> option.or\_else(0)

}

/// A specialized version of `quantity\_of` for the Ada currency.

pub fn lovelace\_of(self: Value) -> Int {

quantity\_of(self, ada\_policy\_id, ada\_asset\_name)

}

/// Get all tokens associated with a given policy.

pub fn tokens(self: Value, policy\_id: PolicyId) -> Dict<AssetName, Int> {

self.inner

|> dict.get(policy\_id)

|> option.or\_else(dict.new())

}

/// A list of all token policies in that Value with non-zero tokens.

pub fn policies(self: Value) -> List<PolicyId> {

dict.keys(self.inner)

}

/// Flatten a value as list of 3-tuple (PolicyId, AssetName, Quantity).

///

/// Handy to manipulate values as uniform lists.

pub fn flatten(self: Value) -> List<(PolicyId, AssetName, Int)> {

dict.foldr(

self.inner,

[],

fn(policy\_id, asset\_list, value) {

dict.foldr(

asset\_list,

value,

fn(asset\_name, quantity, xs) {

[(policy\_id, asset\_name, quantity), ..xs]

},

)

},

)

}

/// Flatten a value as a list of results, possibly discarding some along the way.

///

/// When the transform function returns `None`, the result is discarded altogether.

pub fn flatten\_with(

self: Value,

with: fn(PolicyId, AssetName, Int) -> Option<result>,

) -> List<result> {

dict.foldr(

self.inner,

[],

fn(policy\_id, asset\_list, value) {

dict.foldr(

asset\_list,

value,

fn(asset\_name, quantity, xs) {

when with(policy\_id, asset\_name, quantity) is {

None -> xs

Some(x) ->

[x, ..xs]

}

},

)

},

)

}

test flatten\_with\_1() {

flatten\_with(zero(), fn(p, a, q) { Some((p, a, q)) }) == []

}

test flatten\_with\_2() {

let v =

zero()

|> add("a", "1", 14)

|> add("b", "", 42)

|> add("a", "2", 42)

flatten\_with(

v,

fn(p, a, q) {

if q == 42 {

Some((p, a))

} else {

None

}

},

) == [("a", "2"), ("b", "")]

}

/// Reduce a value into a single result

///

/// ```

/// value.zero()

/// |> value.add("a", "1", 10)

/// |> value.add("b", "2", 20)

/// |> value.reduce(v, 0, fn(\_, \_, quantity, acc) { acc + quantity })

/// // 30

/// ```

pub fn reduce(

self: Value,

start: acc,

with: fn(PolicyId, AssetName, Int, acc) -> acc,

) -> acc {

dict.foldr(

self.inner,

start,

fn(policy\_id, asset\_list, result) {

dict.foldr(asset\_list, result, with(policy\_id, \_, \_, \_))

},

)

}

test reduce\_1() {

let v =

zero()

|> add("a", "1", 10)

|> add("b", "2", 20)

let result = reduce(v, 0, fn(\_, \_, quantity, acc) { acc + quantity })

result == 30

}

test reduce\_2() {

let v =

zero()

|> add("a", "1", 5)

|> add("a", "2", 15)

|> add("b", "", 10)

let result =

reduce(

v,

[],

fn(policy\_id, asset\_name, \_, acc) { [(policy\_id, asset\_name), ..acc] },

)

result == [("a", "1"), ("a", "2"), ("b", "")]

}

test reduce\_3() {

let v = zero()

let result = reduce(v, 1, fn(\_, \_, quantity, acc) { acc + quantity })

result == 1

}

/// Promote an arbitrary list of assets into a `Value`. This function fails

/// (i.e. halt the program execution) if:

///

/// - there's any duplicate amongst `PolicyId`;

/// - there's any duplicate amongst `AssetName`;

/// - the `AssetName` aren't sorted in ascending lexicographic order; or

/// - any asset quantity is null.

///

/// This function is meant to turn arbitrary user-defined `Data` into safe `Value`,

/// while checking for internal invariants.

pub fn from\_asset\_list(xs: Pairs<PolicyId, Pairs<AssetName, Int>>) -> Value {

xs

|> list.foldr(

dict.new(),

fn(inner, acc) {

expect Pair(p, [\_, ..] as x) = inner

x

|> from\_ascending\_pairs\_with(fn(v) { v != 0 })

|> dict.insert\_with(

acc,

p,

\_,

fn(\_, \_, \_) {

fail @"Duplicate policy in the asset list."

},

)

},

)

|> Value

}

test from\_asset\_list\_1() {

let v = from\_asset\_list([])

v == zero()

}

test from\_asset\_list\_2() fail {

let v = from\_asset\_list([Pair(#"33", [])])

v == zero()

}

test from\_asset\_list\_3() fail {

let v = from\_asset\_list([Pair(#"33", [Pair(#"", 0)])])

v != zero()

}

test from\_asset\_list\_4() {

let v = from\_asset\_list([Pair(#"33", [Pair(#"", 1)])])

flatten(v) == [(#"33", #"", 1)]

}

test from\_asset\_list\_5() {

let v = from\_asset\_list([Pair(#"33", [Pair(#"", 1), Pair(#"33", 1)])])

flatten(v) == [(#"33", #"", 1), (#"33", #"33", 1)]

}

test from\_asset\_list\_6() fail {

let v =

from\_asset\_list(

[

Pair(#"33", [Pair(#"", 1), Pair(#"33", 1)]),

Pair(#"33", [Pair(#"", 1), Pair(#"33", 1)]),

],

)

v != zero()

}

test from\_asset\_list\_7() fail {

let v =

from\_asset\_list(

[

Pair(#"33", [Pair(#"", 1), Pair(#"33", 1)]),

Pair(#"34", [Pair(#"", 1), Pair(#"", 1)]),

],

)

v != zero()

}

test from\_asset\_list\_8() {

let v =

from\_asset\_list(

[

Pair(#"33", [Pair(#"", 1), Pair(#"33", 1)]),

Pair(#"34", [Pair(#"31", 1)]),

Pair(#"35", [Pair(#"", 1)]),

],

)

flatten(v) == [

(#"33", #"", 1),

(#"33", #"33", 1),

(#"34", #"31", 1),

(#"35", #"", 1),

]

}

test from\_asset\_list\_9() {

let v =

from\_asset\_list(

[

Pair(#"35", [Pair(#"", 1)]),

Pair(#"33", [Pair(#"", 1), Pair(#"33", 1)]),

Pair(#"34", [Pair(#"31", 1)]),

],

)

flatten(v) == [

(#"33", #"", 1),

(#"33", #"33", 1),

(#"34", #"31", 1),

(#"35", #"", 1),

]

}

/// Convert the value into a dictionary of dictionaries.

pub fn to\_dict(self: Value) -> Dict<PolicyId, Dict<AssetName, Int>> {

self.inner

}

/// A multi-asset value that can be found when minting transaction. It always holds

/// a null quantity of \_Ada\_. Note that because of historical reasons, this is slightly

/// different from `Value` found in transaction outputs.

///

/// Note that you're never expected to construct a `MintedValue` yourself. If you need to

/// manipulate multi-asset values, use [Value](#Value)

///

/// See also [`from\_minted\_value`](#from\_minted\_value).

pub opaque type MintedValue {

inner: Dict<PolicyId, Dict<AssetName, Int>>,

}

/// Convert minted value into a dictionary of dictionaries.

pub fn minted\_to\_dict(self: MintedValue) -> Dict<PolicyId, Dict<AssetName, Int>> {

self.inner

}

/// Convert a [`MintedValue`](#MintedValue) into a [`Value`](#Value).

pub fn from\_minted\_value(self: MintedValue) -> Value {

self.inner |> dict.delete(ada\_policy\_id) |> Value

}

test from\_minted\_value\_1() {

flatten(from\_minted\_value(from\_internal\_list([]))) == []

}

test from\_minted\_value\_2() {

flatten(from\_minted\_value(from\_internal\_list([("p0", "a0", 1)]))) == [

("p0", "a0", 1),

]

}

test from\_minted\_value\_3() {

let assets =

[("p0", "a0", 1), ("p1", "a0", 1), ("p0", "a0", 1), ("p1", "a1", 1)]

let result =

[("p0", "a0", 2), ("p1", "a0", 1), ("p1", "a1", 1)]

flatten(from\_minted\_value(from\_internal\_list(assets))) == result

}

test from\_minted\_value\_4() {

let assets =

[

("", "", 0),

("p0", "a0", 1),

("p1", "a0", 1),

("p0", "a0", 1),

("p1", "a1", 1),

]

let result =

[("p0", "a0", 2), ("p1", "a0", 1), ("p1", "a1", 1)]

flatten(from\_minted\_value(from\_internal\_list(assets))) == result

}

test from\_minted\_value\_5() {

let assets =

[

("p0", "a0", 1),

("p0", "a1", 1),

("p1", "a0", 1),

("p1", "a1", 1),

("p1", "a2", 1),

("p2", "a0", 1),

("p2", "a1", 1),

("p3", "a0", 1),

("p3", "a1", 1),

("p3", "a2", 1),

("p3", "a3", 1),

("p3", "a4", 1),

("p3", "a5", 1),

("p3", "a6", 1),

("p3", "a7", 1),

]

flatten(from\_minted\_value(from\_internal\_list(assets))) == assets

}

/// Convert a [`Value`](#Value) into a [`MintedValue`](#MintedValue).

pub fn to\_minted\_value(self: Value) -> MintedValue {

self.inner

|> dict.insert(ada\_policy\_id, dict.insert(dict.new(), ada\_asset\_name, 0))

|> MintedValue

}

test to\_minted\_value\_1() {

let minted\_value = to\_minted\_value(zero())

( minted\_value.inner |> dict.to\_pairs |> list.length ) == 1

}

test to\_minted\_value\_2() {

let minted\_value = to\_minted\_value(from\_lovelace(42))

(

minted\_value.inner

|> dict.get(ada\_policy\_id)

|> option.and\_then(dict.get(\_, ada\_asset\_name))

) == Some(0)

}

/// Convert a list of tokens into a `MintedValue`.

///

/// NOTE: Not exposed because we do not want people to construct `MintedValue`. Only

/// get them from the script context.

fn from\_internal\_list(xs: List<(PolicyId, AssetName, Int)>) -> MintedValue {

list.foldr(

xs,

MintedValue(dict.new()),

fn(elem, st) {

let (policy\_id, asset\_name, quantity) = elem

unchecked\_add(st, policy\_id, asset\_name, quantity)

},

)

}

fn unchecked\_add(

self: MintedValue,

policy\_id: PolicyId,

asset\_name: AssetName,

quantity: Int,

) -> MintedValue {

MintedValue(

dict.insert\_with(

self.inner,

policy\_id,

dict.from\_ascending\_pairs([Pair(asset\_name, quantity)]),

fn(\_, left, \_right) {

Some(

dict.insert\_with(

left,

asset\_name,

quantity,

fn(\_k, ql, qr) { Some(ql + qr) },

),

)

},

),

)

}