2.15. transaction.ak

use aiken/builtin

use aiken/dict.{Dict}

use aiken/hash.{Blake2b\_224, Blake2b\_256, Hash, blake2b\_256}

use aiken/interval.{Interval}

use aiken/list

use aiken/option

use aiken/time.{PosixTime}

use aiken/transaction/certificate.{Certificate}

use aiken/transaction/credential.{

Address, Script, ScriptCredential, StakeCredential, VerificationKey,

VerificationKeyCredential,

}

use aiken/transaction/value.{MintedValue, PolicyId, Value}

/// A context given to a script by the Cardano ledger when being executed.

///

/// The context contains information about the entire transaction that contains

/// the script. The transaction may also contain other scripts; to distinguish

/// between multiple scripts, the `ScriptContext` also contains a `purpose`

/// which indicates which script (or, for what purpose) of the transaction is

/// being executed.

pub type ScriptContext {

transaction: Transaction,

purpose: ScriptPurpose,

}

/// Characterizes the kind of script being executed.

pub type ScriptPurpose {

/// For scripts executed as minting/burning policies, to insert

/// or remove assets from circulation. It's parameterized by the identifier

/// of the associated policy.

Mint(PolicyId)

/// For scripts that are used as payment credentials for addresses in

/// transaction outputs. They govern the rule by which the output they

/// reference can be spent.

Spend(OutputReference)

/// For scripts that validate reward withdrawals from a reward account.

///

/// The argument identifies the target reward account.

WithdrawFrom(StakeCredential)

/// Needed when delegating to a pool using stake credentials defined as a

/// Plutus script. This purpose is also triggered when de-registering such

/// stake credentials.

///

/// It embeds the certificate that's being validated.

Publish(Certificate)

}

/// A Cardano `Transaction`, as seen by Plutus scripts.

///

/// Note that this is a representation of a transaction, and not the 1:1

/// translation of the transaction as seen by the ledger. In particular,

/// Plutus scripts can't see inputs locked by bootstrap addresses, outputs

/// to bootstrap addresses or just transaction metadata.

pub type Transaction {

inputs: List<Input>,

reference\_inputs: List<Input>,

outputs: List<Output>,

fee: Value,

mint: MintedValue,

certificates: List<Certificate>,

withdrawals: Pairs<StakeCredential, Int>,

validity\_range: ValidityRange,

extra\_signatories: List<Hash<Blake2b\_224, VerificationKey>>,

redeemers: Pairs<ScriptPurpose, Redeemer>,

datums: Dict<Hash<Blake2b\_256, Data>, Data>,

id: TransactionId,

}

/// A placeholder / empty `Transaction` to serve as a base in a transaction

/// builder. This is particularly useful for constructing test transactions.

///

/// Every field is empty or null, and we have in particular:

///

/// ```aiken

/// use aiken/transaction

///

/// transaction.placeholder().id == TransactionId {

/// hash: #"0000000000000000000000000000000000000000000000000000000000000000",

/// }

///

/// transaction.placeholder().validity\_range == interval.everything()

/// ```

pub fn placeholder() -> Transaction {

Transaction {

inputs: [],

reference\_inputs: [],

outputs: [],

fee: value.zero(),

mint: value.zero() |> value.to\_minted\_value(),

certificates: [],

withdrawals: [],

validity\_range: interval.everything(),

extra\_signatories: [],

redeemers: [],

datums: dict.new(),

id: TransactionId {

hash: #"0000000000000000000000000000000000000000000000000000000000000000",

},

}

}

/// An interval of POSIX time, measured in number milliseconds since 1970-01-01T00:00:00Z.

pub type ValidityRange =

Interval<PosixTime>

/// A unique transaction identifier, as the hash of a transaction body. Note that the transaction id

/// isn't a direct hash of the `Transaction` as visible on-chain. Rather, they correspond to hash

/// digests of transaction body as they are serialized on the network.

pub type TransactionId {

hash: Hash<Blake2b\_256, Transaction>,

}

/// An `Input` made of an output reference and, the resolved value associated with that output.

pub type Input {

output\_reference: OutputReference,

output: Output,

}

/// An `OutputReference` is a unique reference to an output on-chain. The `output\_index`

/// corresponds to the position in the output list of the transaction (identified by its id)

/// that produced that output

pub type OutputReference {

transaction\_id: TransactionId,

output\_index: Int,

}

/// A transaction `Output`, with an address, a value and optional datums and script references.

pub type Output {

address: Address,

value: Value,

datum: Datum,

reference\_script: Option<Hash<Blake2b\_224, Script>>,

}

/// An output `Datum`.

pub type Datum {

NoDatum

/// A datum referenced by its hash digest.

DatumHash(Hash<Blake2b\_256, Data>)

/// A datum completely inlined in the output.

InlineDatum(Data)

}

/// A type-alias for Redeemers, passed to scripts for validation. The `Data` is

/// opaque because it is user-defined and it is the script's responsibility to

/// parse it into its expected form.

pub type Redeemer =

Data

/// Find an input by its [`OutputReference`](#OutputReference). This is typically used in

/// combination with the `Spend` [`ScriptPurpose`](#ScriptPurpose) to find a script's own

/// input.

///

/// ```aiken

/// validator {

/// fn(datum, redeemer, ctx: ScriptContext) {

/// expect Spend(my\_output\_reference) =

/// ctx.purpose

///

/// expect Some(input) =

/// ctx.transaction.inputs

/// |> transaction.find\_input(my\_output\_reference)

/// }

/// }

/// ```

pub fn find\_input(

inputs: List<Input>,

output\_reference: OutputReference,

) -> Option<Input> {

inputs

|> list.find(fn(input) { input.output\_reference == output\_reference })

}

/// Find a [`Datum`](#Datum) by its hash, if present. The function looks first for

/// datums in the witness set, and then for inline datums if it doesn't find any in

/// witnesses.

pub fn find\_datum(

outputs: List<Output>,

datums: Dict<Hash<Blake2b\_256, Data>, Data>,

datum\_hash: Hash<Blake2b\_256, Data>,

) -> Option<Data> {

datums

|> dict.get(datum\_hash)

|> option.or\_try(

fn() {

outputs

|> list.filter\_map(

fn(output) {

when output.datum is {

InlineDatum(data) ->

if

blake2b\_256(builtin.serialise\_data(data)) == datum\_hash{

Some(data)

} else {

None

}

\_ -> None

}

},

)

|> list.head

},

)

}

/// Find all outputs that are paying into the given script hash, if any. This is useful for

/// contracts running over multiple transactions.

pub fn find\_script\_outputs(

outputs: List<Output>,

script\_hash: Hash<Blake2b\_224, Script>,

) -> List<Output> {

outputs

|> list.filter(

fn(output) {

when output.address.payment\_credential is {

ScriptCredential(addr\_script\_hash) ->

script\_hash == addr\_script\_hash

VerificationKeyCredential(\_) -> False

}

},

)

}