Plutus Tools SDK User Guide

Release 1.0.0

IOHK

Mar 06, 2023

EXPLORE PLUTUS

CHAPTER

ONE

PLUTUS TOOLS SDK

The Plutus Tools SDK is a collection of off-chain infrastructure resources built for external developers. The Plutus Tools SDK repository supports the underlying resources that developers need who are writing full applications using Plutus in Haskell, including off-chain code. The term "off-chain code" refers to the part of a contract application's code which runs outside of the blockchain. Off-chain code responds to events happening on or off the blockchain, usually by producing transactions.

This user guide is intended for developers who are authoring distributed applications ("DApps") by using smart contracts on the Cardano blockchain.

Note: If you are a developer who wants to contribute to the Plutus Tools SDK project, please refer to documentation residing in the Plutus Tools SDK repository.

CHAPTER

TWO

PLUTUS TOOLS SDK REPOSITORY

The Plutus Tools SDK repository contains packages such as:

- the Plutus Application Backend (PAB), an off-chain application for managing the state of Plutus contract instances;
- the chain-index, a lightweight, customizable chain follower application and library for DApp developers who need to index and query the Cardano blockchain;
- the Plutus Contract Package, a library for writing Plutus contracts and transforming them into executables that run on the application platform;
- the Plutus Ledger Constraints Package, containing an API to build transactions by providing a list of constraints and for constructing and validating Plutus transactions;
- the Trace Emulator, used for testing Plutus contracts on an emulated blockchain;
- a variety of other Plutus packages.

CHAPTER

THREE

USE CASES

Please refer to these use cases to see examples of Plutus applications.

CHAPTER

FOUR

PLUTUS STARTER TEMPLATE REPOSITORY (DEPRECATED)

See the Plutus starter template repository for a simple starter project using the Plutus Tools SDK.

CHAPTER

FIVE

PUBLIC PLUTUS LIBRARIES DOCUMENTATION

See also the public Plutus libraries documentation to access Haddock-generated documentation of all the code, including Plutus Core.

5.1 Explanations

5.1.1 Plutus tools in development

The Plutus tools are currently in development. Early iterations of the tools have undergone testing as part of our research into their performance and features. Based on test results and feedback, our team has entered a new cycle of development to address certain design aspects and to place a greater focus on selected features and capabilities.

Logical components

Plutus tools consists of libraries, executables and logical components within Haskell packages for external developers to use. A single Haskell package may contain multiple logical components.

The tools are located in the plutus-apps repository.

For each tool or logical component shown below, we have indicated its specific location within plutus-apps and provided a brief description.

1. Marconi

Marconi	
Location	`plutus-apps/marconi`

Marconi is a library for indexing data from the Cardano blockchain. *Marconi* is faster and more flexible than *chain-index* and will eventually replace it. *Marconi* is currently in an early alpha version. It has several indexers. While much of the architecture is settled, some aspects are in the design stage.

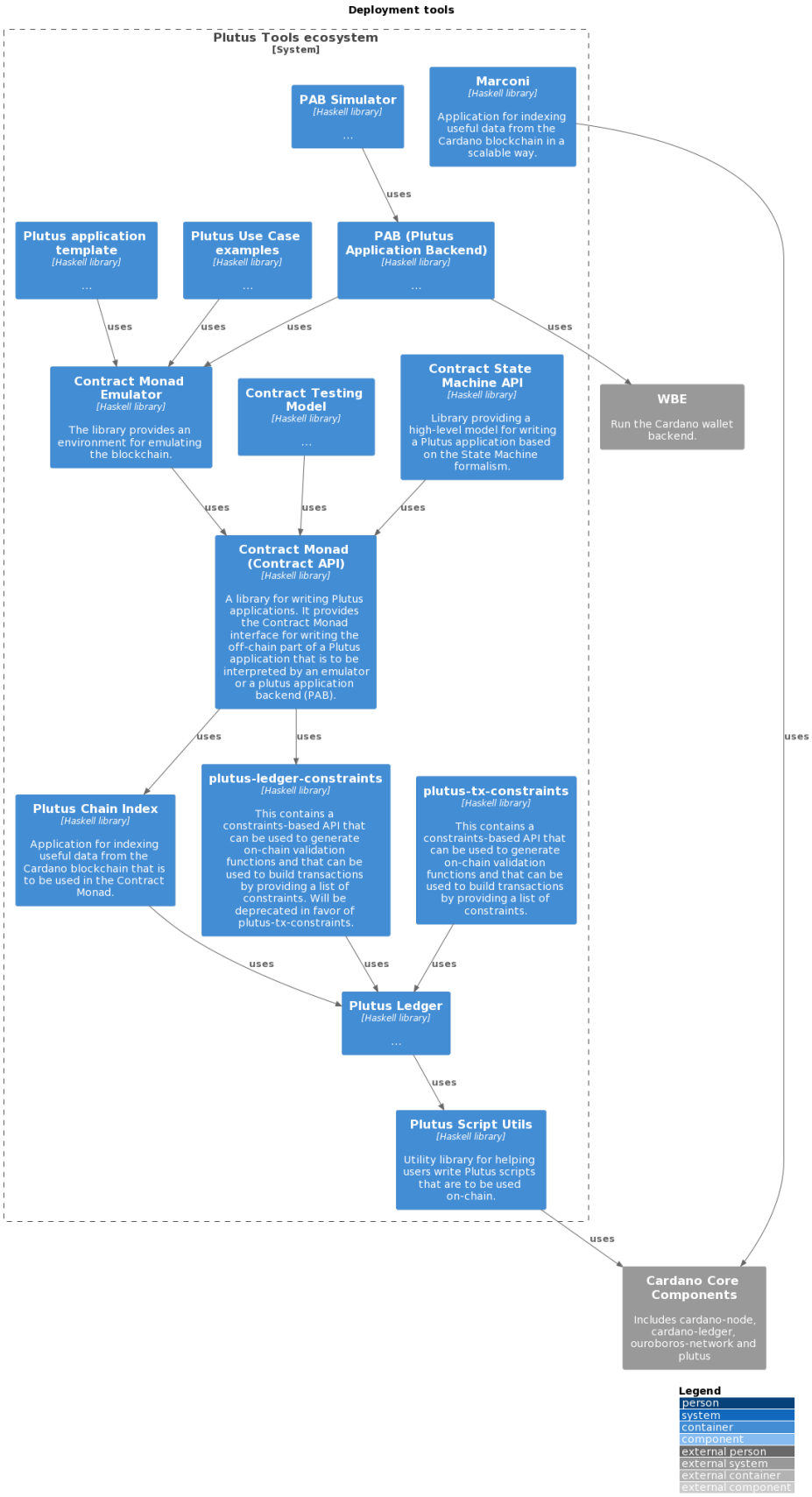


Fig. 1: Illustration of the Plutus Tools Ecosystem showing logical components and some indication of their dependencies and relationships. (image source)

2. Plutus use case examples

Plutus use case examples	
Location	`plutus-apps/plutus-use-cases`

Plutus use case examples contains hand-written examples for the use cases we currently have. The primary examples are:

- Auction,
- Crowdfunding,
- Game,
- GameStateMachine, and
- Vesting.

For each Plutus application use case, we provide test scenarios (or test cases) with and without the `Plutus.Contract.Test.ContractModel`.

The examples are for testing and educational purposes. While they work in the *plutus-contract emulator*, they are not guaranteed to work on the actual Cardano network, primarily because the size of the produced Plutus scripts are too big to fit in a transaction given current protocol parameters.

3. PAB (Plutus application backend)

PAB	
Location	`plutus-apps/plutus-pab`

PAB is a web server library for managing the state of Plutus contract instances. The PAB executes the off-chain component of Plutus applications. It manages application requests to the wallet backend, the Cardano node and the chain-index. PAB stores the application state and offers an HTTP REST API for managing application instances.

PAB wraps the contracts built with plutus-contract. It is the central point of contact, integrating many Cardano components.

4. Contract monad emulator

Contract monad emulator	
Location	`plutus-apps/plutus-contract`

Contract monad emulator is a library that provides an environment for emulating the blockchain. The environment provides a way for writing traces for the contract which are sequences of actions by simulated wallets that use the contract. The component is highly dependent on the Contract API (Contract monad).

5. Plutus contract model testing

Plutus contract model testing	
Location	`plutus-apps/plutus-contract`

Plutus contract model testing is used for testing prototype Plutus contracts with *contract models*, using the framework provided by `Plutus.Contract.Test.ContractModel`. This framework generates and runs tests on the Plutus emulator, where each test may involve a number of emulated wallets, each running a collection of Plutus contracts, all submitting transactions to an emulated blockchain. Once you have defined a suitable model, then QuickCheck can generate and run many thousands of scenarios, taking the application through a wide variety of states, and checking that it behaves correctly in each one.

See the following tutorials:

- *Property-based testing of Plutus contracts*
- *Testing Plutus Contracts with Contract Models*

6. Plutus contract state machine

Plutus contract state machine	
Location	`plutus-apps/plutus-contract`

Plutus contract state machine is a library that is a useful high-level tool for defining and modeling a Plutus application (smart contract) based on the State Machine formalism. It is helpful for writing a reference implementation for testing before creating the production version. However, we do not recommend using it in production as the scripts are too big to run on-chain.

7. Contract API (also known as Contract monad)

Contract API	
Location	`plutus-apps/plutus-contract`

Contract API is a logical component within the Plutus Contract package, providing an effect system for describing smart contracts that interact with wallets, DApps, a chain indexer and the blockchain. It provides the Contract API interface for writing the off-chain part of a Plutus application that is to be interpreted by an emulator or by Plutus application backend (PAB).

8. Plutus chain index

Plutus chain index	
Location	`plutus-apps/plutus-chain-index-core`
	`plutus-apps/plutus-chain-index`

Plutus chain index is an application for indexing data from the Cardano blockchain that is used in the Contract Monad. The main design goal is to keep the size of the indexed information proportional to the UTXO set.

9. Plutus Tx constraints

Plutus Tx constraints	
Location	`plutus-apps/plutus-tx-constraints`

Plutus-tx-constraints contains a constraints-based API that can be used to generate on-chain validation functions and to build transactions by providing a list of constraints. The main design goal is to be able to use the same constraints on-chain and off-chain in a Plutus application. The off-chain part generates transactions based on types in cardano-api.

For example:

- `checkScriptContext (MustSpendAtLeast 10Ada, MustProduceOutput myOutput, ...)`
- `mkTx (MustSpendAtLeast 10Ada, MustProduceOutput myOutput, ...)`

10. Plutus ledger

Plutus ledger	
Location	`plutus-apps/plutus-ledger`

Plutus ledger is a set of transitional types that simplify the cardano-api types. It is intended to be a comprehensive, easy-to-use set of types that replicate the current era of cardano-api. It currently considers only the last era. Plutus ledger contains data types and functions that complement cardano-ledger related to Plutus.

11. Plutus script utils

Plutus script utils	
Location	`plutus-apps/plutus-script-utils`

Plutus script utils is a utility library for helping users write Plutus scripts that are to be used on-chain. *Plutus script utils* includes a variety of useful functions for on-chain operations in Plutus scripts.

It provides a number of utilities including:

- hashing functions for Datums, Redeemers and Plutus scripts for any Plutus language version.
- functionality for wrapping the untyped Plutus script with a typed interface.
- utility functions for working with the ScriptContext of a Plutus Script.

5.1.2 What is a rollback?

The Cardano network is a distributed system with many nodes operating at the same time. Each node keeps its own local copy of the blockchain, extending it regularly with new blocks. At the same time, the node is talking to some of the other nodes in the network in order to establish a consensus about what *the* canonical blockchain should be. Sometimes the node discovers that its local version of the blockchain is different from the canonical one that the other nodes agree on. When that happens, the node has to switch to the correct blockchain. In order to perform this switch, the node first *rolls back* its own blockchain, by discarding the last couple of blocks that are different from the target blockchain. After the rollback, the node's local blockchain is a prefix of the target chain, so the node can safely *roll*

forward, to catch up with any missing blocks that exist on the target chain but haven't yet been added to the local chain.

Note: Rollbacks are part of the normal operation of the consensus algorithm and happen frequently.

Rollbacks cannot extend indefinitely into the past: The maximum number of blocks that may be discarded due to a rollback is defined as a chain constant k, and any blocks that are deeper than k blocks in are guaranteed to never be rolled back.

What does this mean for dapps?

To a Plutus app whose on-chain state is a set of *unspent transaction outputs*, the effect of a rollback is to undo changes to the state of transaction outputs that were performed by transactions which are now being dropped as part of the rollback. Let's look at a concrete example to illustrate the effect of rollbacks.

In the example above, we have three different UTXO sets, and two blocks that transition between them:

UTXO set #	Unspent outputs	Changes to previous
1	A, B, C, D	N/A
2	A, C, E, F, G	B, D removed. E,F,G added.
3	A, C, E, F, H	G removed. H added.

Now assume that the second block is rolled back, and our new ledger state is UTXO set No. 2 in the table. Rolling back the block undoes the changes of Tx3: The state of output G changes back to *unspent* and H disappears altogether, so its new state is *unknown*. If we roll back both blocks then the state of outputs B, D, E, F and G is also affected. B and D are now *unspent* and E, F, and G are *unknown*.

Note that transactions that were rolled back may be reapplied if they are still valid, that is if their inputs haven't been spent by other transactions. For example, after rolling back block 2, Tx3 could be added to new chain at a later stage (perhaps as part of a different block). Then the state of G would change again, from *unspent* to *spent*.

5.1.3 What is the PAB?

PAB is short for *Plutus Application Backend*. The Plutus Application Backend is the client-side runtime for *Plutus apps* that are built with the Plutus Platform. It is the PAB's task to deal with requests from running `Contract` instances, to forward user input to them, and to notify them of ledger state change events.

The `plutus-pab` cabal package in the Plutus repository defines a `plutus-pab` Haskell library. Application developers use this library to build the actual PAB executable, specialised to one or more of their `Contract` s.

Note: In an older version of the PAB, each `Contract` was compiled to a separate executable, and there was a single PAB that knew about all the locally available excutable contracts. This approach is not supported anymore.

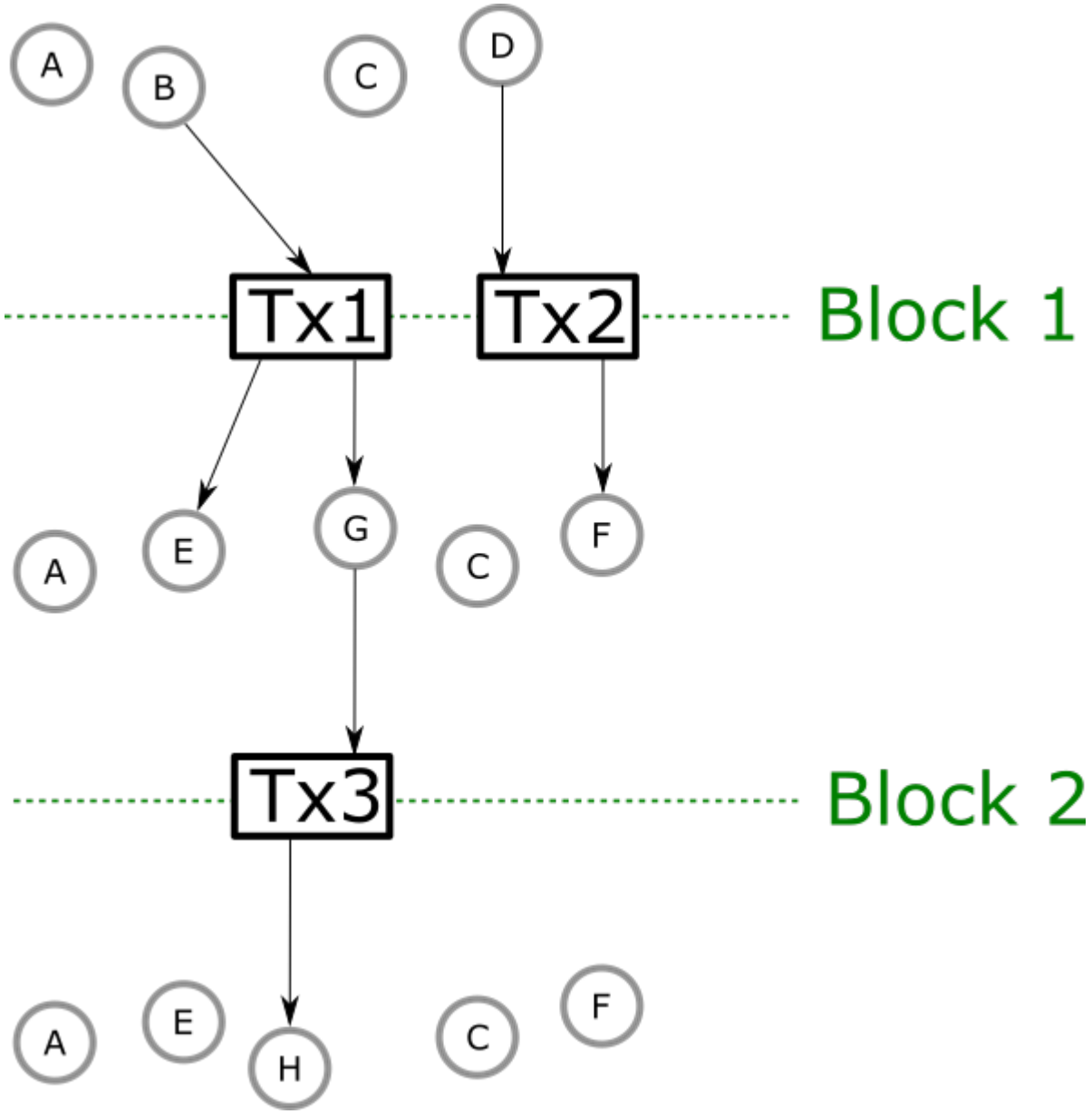


Fig. 2: Illustration of a blockchain with two blocks. Block 1 contains two transactions and block 2 contains one transaction.

Client interface

The PAB provides an HTTP and websocket interface for interacting with `Contract` instances. All PAB operations, including starting new instances, calling endpoints on instances, and querying instance state, are performed using this API. Application developers can build their own frontends and server processes that make HTTP calls to the PAB.

Other components

In addition to the PAB itself, the following components are required.

Chain index

The chain index is a database of data gathered from Cardano transactions. It uses the Cardano node's chain sync protocol. Therefore it needs to be co-located with a Cardano node. The chain index is a read-only component for the PAB. Multiple instances of the PAB can therefore share a single instance of the chain index.

The expressiveness of queries supported by the chain index lies somewhere between that of the node, which answers queries related to the ledger state, and that of `db-sync`, which has a full history of all transactions and an expressive database schema for staking and other information.

All chain index queries are served over an HTTP API.

Alonzo node

The PAB subscribes to ledger state updates from the node, using a socket protocol.

Wallet

A Cardano wallet is required for balancing and signing transactions (and optionally submitting transactions). Balancing means taking a partial transaction and adding inputs and outputs to make the transaction valid.

Take Marlowe as an example. When the user first starts a Marlowe contract, funds need to be transferred from one of the user's addresses to the contract address. This is achieved by sending a partial transaction that has zero inputs and a script output for the Marlowe contract instance to the wallet for balancing. The wallet adds some of its own inputs to cover the amount that is to be paid into the contract, plus a change output for any excess funds. When the Marlowe contract has finished, funds are transferred back to the user's wallet using the same mechanism: The PAB sends another partial transaction, this time with a single script input and no outputs. The wallet then adds an output at one of its own addresses to receive the funds.

There are multiple ways to setup a wallet:

1. Host a cardano wallet backend instance (WBE) using cardano-wallet
2. Setup a desktop wallet application (ex. Daedalus)
3. Setup a browser wallet application (ex. Nami, Yoroi, etc.)

These different wallet setups each imply a different use-case of the PAB.

Deployment Scenarios

There are two deployment models envisaged for the PAB: Hosted and in-browser. The hosted variant will be supported at the initial release of the PAB. The in-browser variant will be available after the initial release.

Hosted

In the "Hosted PAB" scenario, the dApp provider / developer hosts an instance of the PAB alongside the *chain index* and an Alonzo node. The off-chain code of the Plutus app is run on the dApp provider's infrastructure.

In the following sections, we illustrate the ways a hosted PAB can be used with the different type of wallets.

WBE (Supported)

In this wallet scenario, the dApp provider /developer also hosts an instance of the WBE, which handles the wallets for each user. The WBE handles balancing, signing and submitting transaction requests from the PAB.

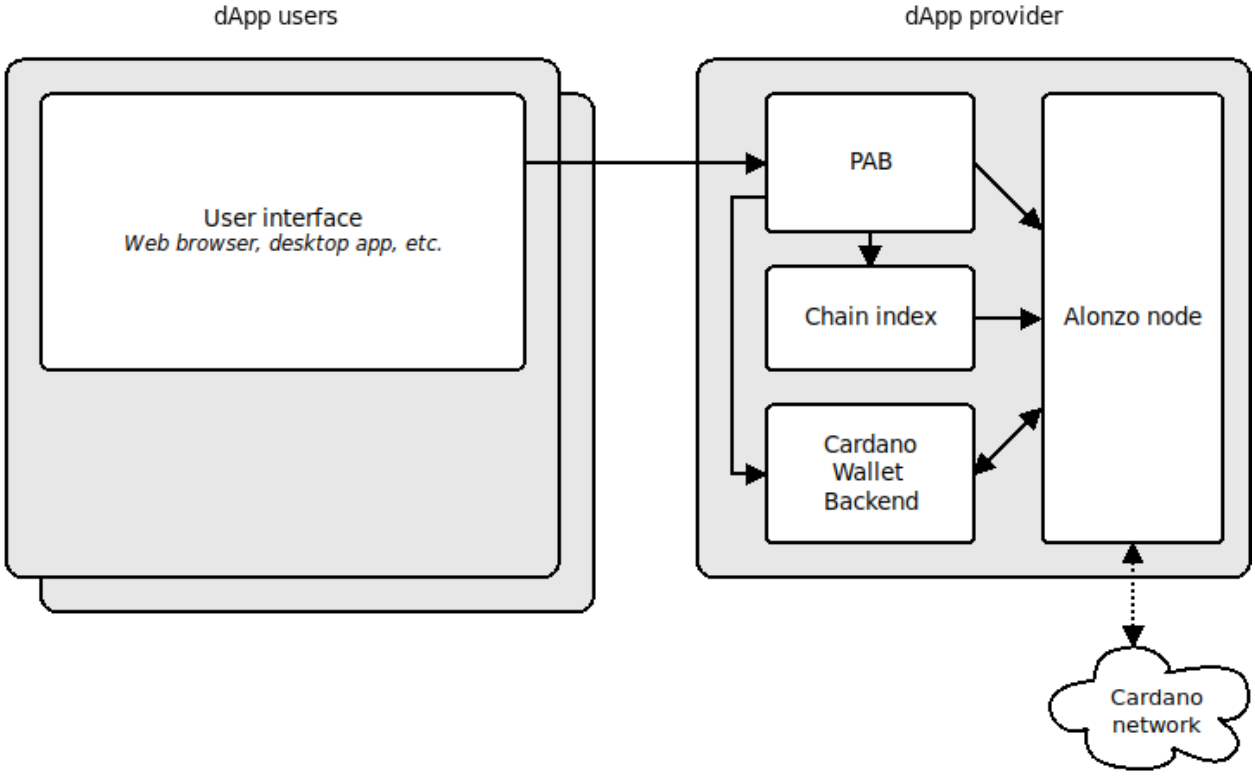


Fig. 3: The hosted deployment scenario for the PAB with the WBE

This is currently used for testing purposes and shouldn't be used in a production setting, because we wallets are normally controlled by the users themselves.

A simple demo of this scenario is available here: https://github.com/input-output-hk/plutus-apps/tree/main/plutus-pab/test-node.

Desktop wallet (Not yet supported)

In this wallet scenario, the user has setup a desktop wallet (light or full node) such as Daedalus. Transaction balancing (coin selection) and transaction signing (in short: anything that deals with the user's money) happens on the user's machine. The PAB produces a link (URI) for each partial transaction that needs to be balanced, signed and submitted. When the user clicks the link, the user's operating system opens the wallet that is registered to handle the link schema. This scheme is not restricted to Daedalus, or even to full node wallets. Any wallet that implements a handler for the link schema can be used to balance, sign and submit Plutus transactions.

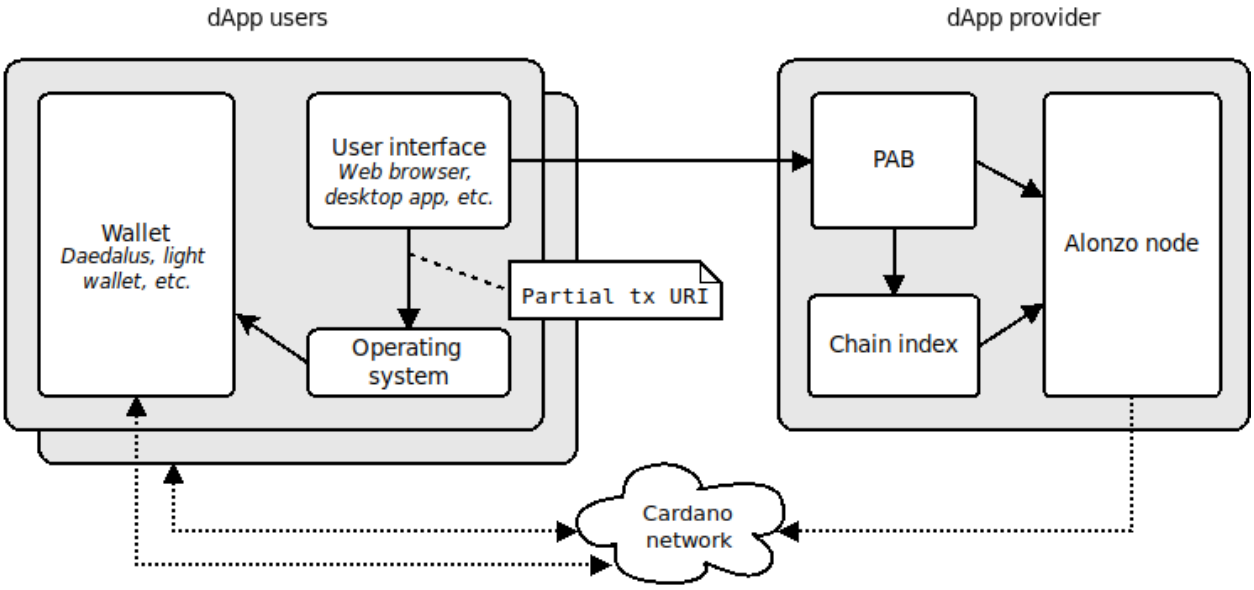


Fig. 4: The hosted deployment scenario for the PAB communicating with a desktop wallet.

Browser wallet (In progress)

In this wallet scenario, the user has setup a browser wallet such as Nami or Yoroi. The PAB updates it's contract instance status endpoint for each partial transaction that needs to be balanced, signed and submitted. Transaction signing happens on the user's machine. However, transaction balancing (coin selection) is handled by the PAB as it is not currently possible to balance transaction that contain script inputs in the browser (i.e. browser wallets can't balance transactions until it is possible to execute Plutus script in the browser). Therefore, browser wallets will need to call a PAB helper endpoint which can balance the transaction using funds from the user's browser wallet.

In-browser

In the "In-browser PAB" scenario, the dApp provider / developer hosts an instance of the *chain index* and an Alonzo node. The dApp users work with a browser interface which uses a light version of the PAB.

Similary to the hosted PAB scenario, we illustrate the ways it can be used the different type of wallets.

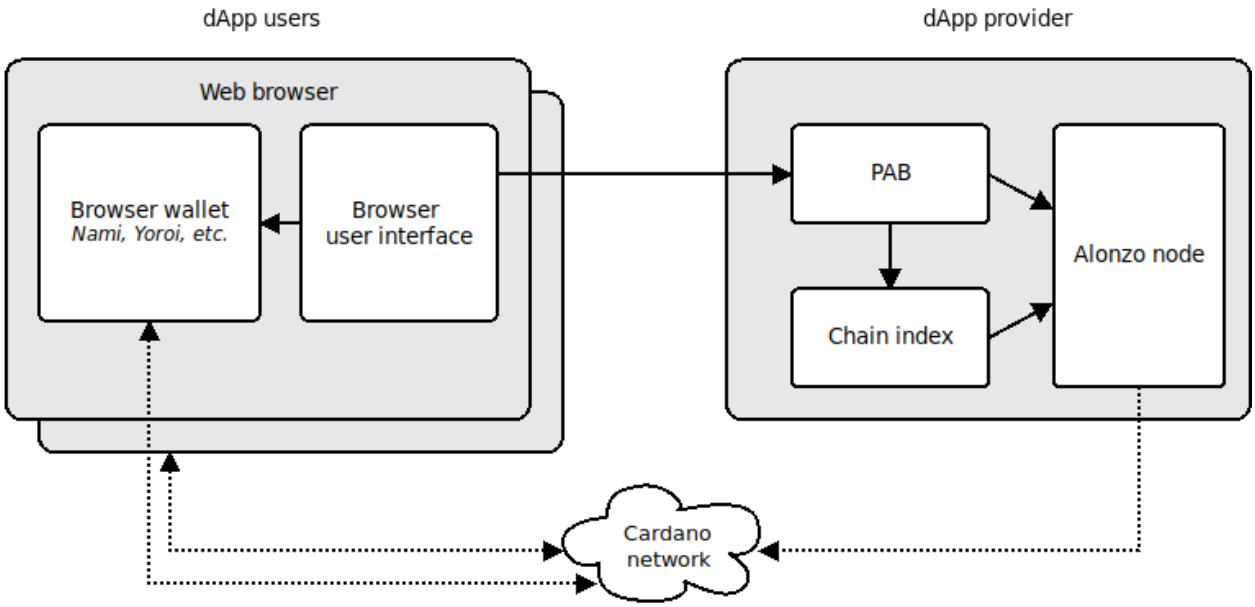


Fig. 5: The hosted deployment scenario for the PAB communicating with a browser wallet.

Desktop wallet (Not yet supported)

Browser wallet (Not yet supported)

5.1.4 What is the order book pattern?

The order book pattern is a way of organising distributed applications on *Cardano*. The key idea of the order book pattern is to materialise *actions* that act on some state as *UTXOs* on the *ledger*, separating them from the state they act on. The spending of those UTXOs (applying the action to the state) can be performed by an untrusted third party. The pattern is helpful for designing applications that follow the *Scalability guidelines*.

Example: Distributed exchange

This is the example that gives the order book its name. A distributed exchange is an app that lets users trade *currency* values in a decentralised fashion, without a central broker. At the heart of this app is the order book, a list of open buy and sell offers for specific amounts of currency. Every buy order needs to be matched with a sell order.

Example 1: "Buy 2000 PEAR at 15.00 ADA", "Sell 2000 PEAR at 14.95 ADA" are buy and sell orders for PEAR *tokens*. In this example, the quantities (2000 units) are identical, and the sell price is lower then the buy price, so we could match the two orders directly and keep the difference between buy and sell price as a fee.

Example 2: "Buy 2000 PEAR at 15.00 ADA", "Sell 1500 PEAR at 14.95 ADA". In this case we need to find at least one other sell order for 500 PEAR to make the quantities match up, for example "Sell 500 PEAR at 14.99". But this isn't the only solution to the problem: Maybe we can find another buy order and a bigger sell order, so that we can resolve four orders simultaneously.

The example illustrates where the complexity lies in the order book system: In finding and matching orders in a way that is profitable for the match maker (broker).

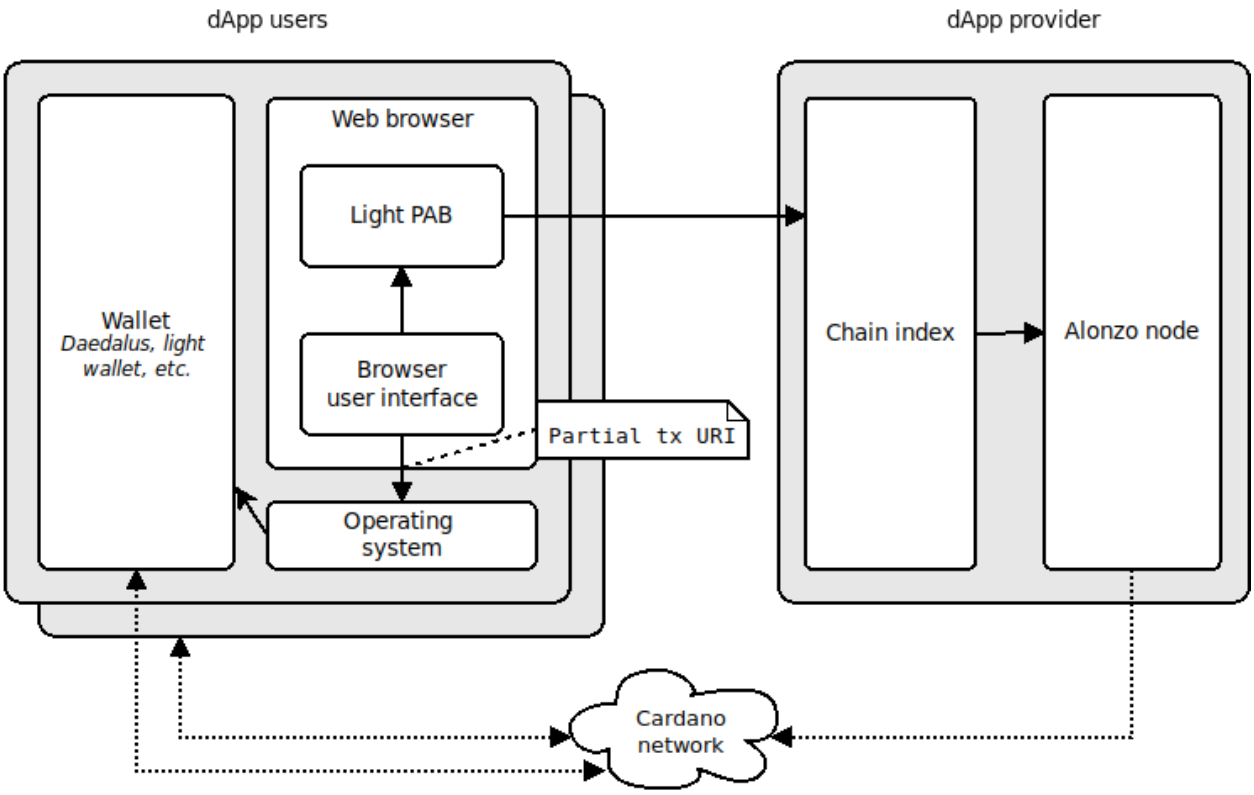


Fig. 6: The in-browser PAB communicating with a desktop wallet.

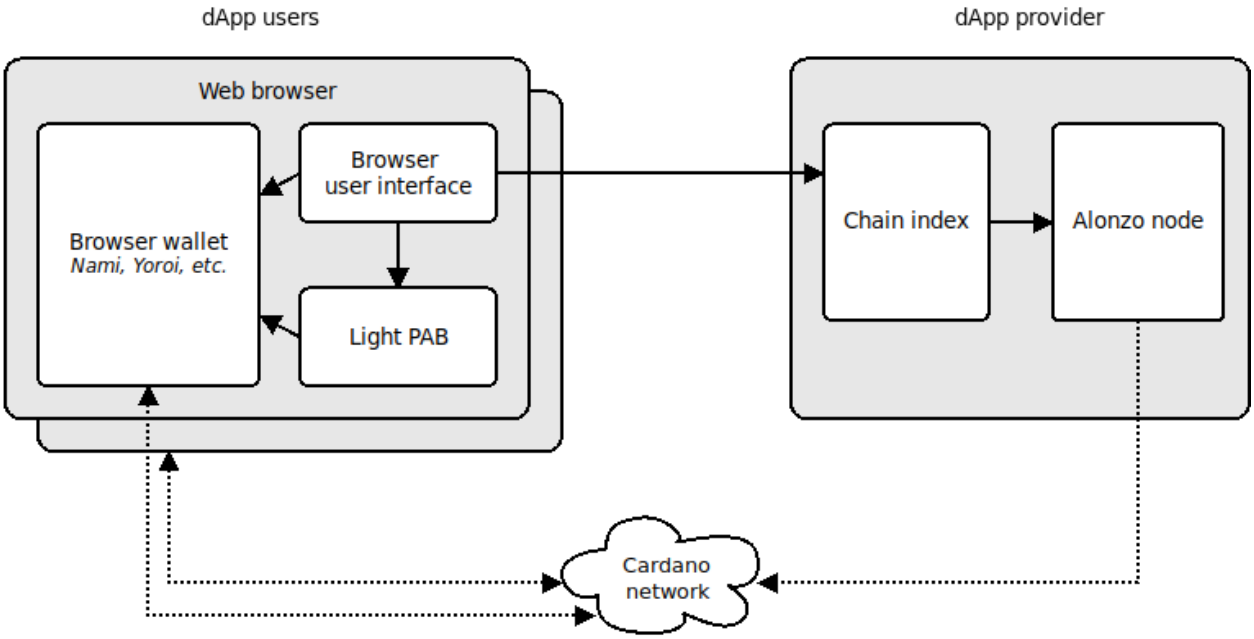


Fig. 7: The in-browser PAB communicating with a browser wallet.

Orders and Plutus scripts

The order book lends itself to a nice representation in the EUTXO model. Every order is a single UTXO, and matching a set of orders means building a transaction that spends the relevant UTXOs. The UTXOs are script UTXOs with a known address and a datum value that holds the quoted price and some bookkeeping information (for example, an address to pay the money to, and an expiration date). The currency value locked in the UTXO is the "inverse" of the order.

Example 3: "Buy 2000 PEAR at 15.00 ADA" results in a script UTXO with 2000 * 15 = 30,000 ADA. "Sell 2000 PEAR at 14.95 ADA" gives a UTXO with 2000 PEAR tokens. The match maker can build a transaction that spends the two UTXOs, pays 2000 PEAR tokens to the buyer, 29,000 ADA to the seller and the difference (100 ADA) to the match maker without ever owning any PEAR tokens.

Decentralisation

It is clear that the match maker is a crucial component of the system. Without someone monitoring the script address and building transactions that match buy orders with sell orders, the orders will never be fulfilled. How can we make sure that the match maker does not become too powerful or too centralised?

We can achieve decentralisation by open-sourcing the scripts that lock the order outputs. With this code available to the public, anyone can build and run a match-making service and earn fees from matching buy and sell orders. There could even be a specialised *PAB* available for download somewhere that only runs the match making service, allowing non-programmer users to run nodes for the DEX.

There is no risk of tokens being stolen because the Plutus script ensures that the outputs can only be spent if the order is met exactly as specified. And while it is possible that the spending transactions suffer from *UTXO congestion* (if multiple transactions that match a particular order are submitted), this does not have a negative impact on the user experience, because the buyer or seller does not care which particular match maker ends up fulfilling their order. In fact, match makers are incentivised to compete by providing faster fulfilment of orders, which actually results in a better outcome for end users.

Generalising the pattern

The original application matches buyers and sellers of currency values, but there are other areas where the pattern is useful.

Off-chain oracles

Imagine a Plutus script that says "If you give me the current USD/EUR exchange rate signed by a specific private key, then I will pay you 5 ADA (and use the exchange rate to run the rest of my *$BUSINESS_LOGIC*)". The match maker then builds a transaction that combines the oracle value with the Plutus script. Of course this example requires the match maker to be able to obtain the signed value, but it does succeed in decoupling consumers of the information from producers.

On-chain oracles

The pattern could also be a building block for on-chain oracles.

Let's say we have a crypto-backed stablecoin, not too dissimilar to Djed, that relies on recent quotes of the exchange rate between Ada and PEAR tokens. And we have a DEX like the one described above where Ada and PEAR are actively traded. Every order fulfilled on the DEX gives us a snapshot of the exchange rate at that time. Example 1 from above would result in "15.00 ADA / PEAR" (using the buy price here but that's just a technicality).

This is exactly the information that we need for our dealings with the stablecoin, but how do we get it from the DEX to the stablecoin? There are two options. To choose the right one we need to consider the requirement and usage patterns of our application.

1. Oracle UTXOs

We could change the DEX contract to produce a new script UTXO for each fulfilled order that records the time and exchange rate of the order. The stablecoin user creates a UTXO with a script that requires an oracle UTXO to be present in the spending transaction, and the match maker would put the oracle UTXO and the stablecoin-action UTXO into one transaction and submit it.

2. Oracle tokens

In a variation of the first idea, the DEX could produce *tokens* that encode the script-certified information we are interested in. We could set the asset name of the token to be the hash of the exchange rate data, and allow the transaction to produce any quantitiy of these tokens when the order is fulfilled.

The *minting policy* of the oracle token should allow any number of tokens of the same asset name to be created as long as at least one token with that asset name exists already, reflecting the idea that information is hard to obtain but easy to replicate.

The consumer of the oracle token needs to check that a token with the expected minting policy hash is present in the transaction, and that the datum value of the token's asset name is available. Then it can use the information from the datum. Maybe it could even destroy the token when it has been used.

This approach has the advantage of not clogging up the UTXO set too much, but the big question here is: How do we make the oracle token available to the match maker? It has to be stored in an output that the match maker can spend. The solution depends on the project. There is no general solution (yet) and some experimentation and research is needed. Perhaps the tokenomics of the exchange could have incentives to make this information flow to where it is needed.

State machines

State machines are a way of modeling smart contracts that is easy to understand and reason about. However, in their basic formulation they keep the entire state of each individual execution in a single UTXO, which puts them at risk of UTXO congestion caused by multiple users trying to transition the instance to a new state at the same time.

If we can batch multiple transitions into one (for example, by finding a suitable *Semigroup* instance for the state machine's input type) then we could use the order book pattern to allow a number of users to submit transitions *without spending the UTXO* with the state machine instance's state. The match maker would construct a transaction that applies the sum of all proposed transitions in a single step. IOG is actively pursuing research in this area.

Conclusion

In the order book pattern we materialise *actions* as transaction outputs on the ledger, separating them from the state that they act on. The pattern is attractive because it decouples submission of orders (actions, requests for oracle values, etc) from fulfilling them, and because it enables order fulfilment to be run in a fully decentralised, trustless fashion. At the same time it fits the UTXO model very well, because it reduces the number of data dependencies on a single unspent output.

5.2 Tutorials

5.2.1 Writing a basic Plutus app in an emulated environment

Plutus apps are programs that run off-chain and manage active contract instances. They monitor the blockchain, ask for user input, and submit transactions to the blockchain. If you are a contract author, building a Plutus app is the easiest way to create and spend Plutus script outputs. In this tutorial you are going to write a Plutus app that locks some ada in a script output and splits them evenly between two recipients.

```
import Cardano.Node.Emulator.Params (pNetworkId)
import Control.Monad (forever, void)
import Control.Monad.Freer.Extras.Log (LogLevel (Debug, Info))
import Data.Aeson (FromJSON, ToJSON)
import Data.Default (def)
import Data.Text qualified as T
import Data.Text qualified as Text
import GHC.Generics (Generic)
import Ledger (CardanoAddress, PaymentPubKeyHash (unPaymentPubKeyHash),␣
↪toPlutusAddress)
import Ledger.Tx.Constraints qualified as Constraints
import Ledger.Typed.Scripts qualified as Scripts
import Plutus.Contract (Contract, Endpoint, Promise, endpoint, getParams, logInfo,␣
↪selectList, submitTxConstraints,
                        submitTxConstraintsSpending, type (.\/), utxosAt)
import Plutus.Contract.Test (w1, w2)
import Plutus.Script.Utils.Ada qualified as Ada
import Plutus.Trace.Emulator qualified as Trace
import Plutus.V1.Ledger.Api (Address, ScriptContext (ScriptContext,␣
↪scriptContextTxInfo), TxInfo (txInfoOutputs),
                             TxOut (TxOut, txOutAddress, txOutValue), Value)
import PlutusTx qualified
import PlutusTx.Prelude (Bool, Maybe (Just, Nothing), Semigroup ((<>)), mapMaybe,␣
↪mconcat, ($), (&&), (-), (.), (==),
                         (>=))
import Prelude (IO, (<$>), (>>))
import Prelude qualified as Haskell
import Wallet.Emulator.Stream (filterLogLevel)
import Wallet.Emulator.Wallet (Wallet, mockWalletAddress)
```

Defining the types

You start by defining some data types that you're going to need for the *Split* app.

```
data SplitData =
    SplitData
        { recipient1 :: Address -- ^ First recipient of the funds
        , recipient2 :: Address -- ^ Second recipient of the funds
        , amount     :: Ada.Ada -- ^ How much Ada we want to lock
        }
    deriving stock (Haskell.Show, Generic)

-- For a 'real' application use 'makeIsDataIndexed' to ensure the output is stable␣
↪over time
PlutusTx.unstableMakeIsData ''SplitData
PlutusTx.makeLift ''SplitData
```

`SplitData` describes the two recipients of the funds, and the total amount of the funds denoted in ada.

You are using the `Plutus.V1.Ledger.Api.Address` type to identify the recipients. When making the payment you can use the hashes to create two public key outputs.

Instances for data types

The `SplitData` type has instances for a number of typeclasses. These instances enable the serialisation of `SplitData` to different formats. `ToJSON` and `FromJSON` are needed for JSON serialization. JSON objects are passed between the frontend (for example, the Plutus apps emulator) and the app instance. `PlutusTx.FromData` and `PlutusTx.ToData` are used for values that are attached to transactions, for example as the <redeemer> of a script output. This class is used by the Plutus app at runtime to construct `Data` values. Finally, `PlutusTx.makeLift` is a Template Haskell statement that generates an instance of the `PlutusTx.Lift.Class.Lift` class for `SplitData`. This class is used by the Plutus compiler at compile-time to construct Plutus core programs.

Defining the validator script

The validator script is the on-chain part of our Plutus app. The job of the validator is to look at individual transactions in isolation and decide whether they are valid. Plutus validators have the following type signature:

```
d -> r -> ScriptContext -> Bool
```

where `d` is the type of the <datum> and `r` is the type of the redeemer.

You are going to use the validator script to lock a script output that contains the `amount` specified in the `SplitData`.

Note: There is an n-to-n relationship between Plutus apps and validator scripts. Apps can deal with multiple validators, and validators can be used by different apps.

In this tutorial you only need a single validator. Its datum type is `SplitData` and its redeemer type is `()` (the unit type). The validator looks at the `Plutus.V1.Ledger.Api.ScriptContext` value to see if the conditions for making the payment are met:

```
validateSplit :: SplitData -> () -> ScriptContext -> Bool
validateSplit SplitData{recipient1, recipient2, amount} _ ScriptContext
↪{scriptContextTxInfo} =
```

(continues on next page)

(continued from previous page)

```
    let half = Ada.divide amount 2
        outputs = txInfoOutputs scriptContextTxInfo
    in
    Ada.fromValue (valuePaidToAddr outputs recipient1) >= half &&
    Ada.fromValue (valuePaidToAddr outputs recipient2) >= (amount - half)
where
    valuePaidToAddr :: [TxOut] -> Address -> Value
    valuePaidToAddr outs addr =
        let flt TxOut{txOutAddress, txOutValue} | txOutAddress == addr = Just␣
↪txOutValue
            flt _ = Nothing
        in mconcat $ mapMaybe flt outs
```

The validator checks that the transaction, represented by `Plutus.V1.Ledger.Api.scriptContextTxInfo`, pays half the specified amount to each recipient.

You then need some boilerplate to compile the validator to a Plutus script (see Writing basic validator scripts in the Plutus Core and Plutus Tx User Guide).

```
data Split
instance Scripts.ValidatorTypes Split where
    type instance RedeemerType Split = ()
    type instance DatumType Split = SplitData

splitValidator :: Scripts.TypedValidator Split
splitValidator = Scripts.mkTypedValidator @Split
    $$(PlutusTx.compile [|| validateSplit ||])
    $$(PlutusTx.compile [|| wrap ||]) where
        wrap = Scripts.mkUntypedValidator @ScriptContext @SplitData @()
```

The `Plutus.Script.Utils.V1.Typed.Scripts.Validators.ValidatorTypes` class defines the types of the validator, and `splitValidator` contains the compiled Plutus core code of `validateSplit`.

Asking for input

When you start the app, you want to ask the sender for a `SplitData` object. In Plutus apps, the mechanism for requesting inputs is called *endpoints*.

All endpoints that an app wants to use must be declared as part of the type of the app. The set of all endpoints of an app is called the *schema* of the app. The schema is defined as a Haskell type. You can build a schema using the `Plutus.Contract.Endpoint` type family to construct individual endpoint types, and the `.\/` operator to combine them.

```
data LockArgs =
        LockArgs
            { recipient1Address :: CardanoAddress
            , recipient2Address :: CardanoAddress
            , totalAda          :: Ada.Ada
            }
    deriving stock (Haskell.Show, Generic)
    deriving anyclass (ToJSON, FromJSON)

type SplitSchema =
        Endpoint "lock" LockArgs
        .\/ Endpoint "unlock" LockArgs
```

The `SplitSchema` defines two endpoints, `lock` and `unlock`. Each endpoint declaration contains the endpoint's name and its type.

To use the `lock` endpoint in our app, you call the `Plutus.Contract.Request.endpoint` function:

```
lock :: Promise () SplitSchema T.Text ()
lock = endpoint @"lock" (lockFunds . mkSplitData)

unlock :: Promise () SplitSchema T.Text ()
unlock = endpoint @"unlock" (unlockFunds . mkSplitData)
```

`endpoint` has a single argument, the name of the endpoint. The name of the endpoint is a Haskell type, not a value, and you have to supply this argument using the type application operator `@`. This operator is provided by the TypeApplications GHC extension.

Next you need to turn the endpoint parameter datatype `LockArgs` into the `SplitData` datatype used by the Plutus script.

```
mkSplitData :: LockArgs -> SplitData
mkSplitData LockArgs{recipient1Address, recipient2Address, totalAda} =
    SplitData
        { recipient1 = toPlutusAddress recipient1Address
        , recipient2 = toPlutusAddress recipient2Address
        , amount = totalAda
        }
```

Locking the funds

With the `SplitData` that you got from the user you can now write a transaction that locks the requested amount of ada in a script output.

```
lockFunds :: SplitData -> Contract () SplitSchema T.Text ()
lockFunds s@SplitData{amount} = do
    logInfo $ "Locking " <> Haskell.show amount
    let tx = Constraints.mustPayToTheScriptWithDatumInTx s (Ada.toValue amount)
    void $ submitTxConstraints splitValidator tx
```

Using the constraints library that comes with the Plutus SDK you specify a transaction `tx` in a single line.

After calling `Plutus.Contract.submitTxConstraints` in the next line, the Plutus app runtime examines the transaction constraints `tx` and builds a transaction that fulfills them. The runtime then sends the transaction to the wallet, which adds enough to cover the required funds (in this case, the ada amount specified in `amount`).

Unlocking the funds

All that's missing now is the code for retrieving the funds, and some glue to put it all together.

```
unlockFunds :: SplitData -> Contract () SplitSchema T.Text ()
unlockFunds SplitData{recipient1, recipient2, amount} = do
    networkId <- pNetworkId <$> getParams
    let contractAddress = Scripts.validatorCardanoAddress networkId splitValidator
    utxos <- utxosAt contractAddress
    let half = Ada.divide amount 2
        tx =
            Constraints.collectFromTheScript utxos ()
            <> Constraints.mustPayToAddress recipient1 (Ada.toValue half)
```

(continues on next page)

(continued from previous page)

```
            <> Constraints.mustPayToAddress recipient2 (Ada.toValue $ amount - half)
    void $ submitTxConstraintsSpending splitValidator utxos tx
```

In `unlockFunds` you use the constraints library to build the spending transaction. Here, `tx` combines three different constraints. `Ledger.Tx.Constraints.collectFromTheScript` takes the script outputs in `unspentOutputs` and adds them as input to the transaction, using the unit `()` as the redeemer. The other two constraints use `Ledger.Tx.Constraints.mustPayToAddress` to add payments for the recipients.

Running the app on the Plutus apps emulator

You have all the functions you need for the on-chain and off-chain parts of the app. Every contract in the Plutus apps emulator must define its public interface like this:

```
splitPlutusApp :: Contract () SplitSchema T.Text ()
```

`splitPlutusApp` is the high-level definition of our app:

```
splitPlutusApp = forever $ selectList [lock, unlock]
```

The `Plutus.Contract.selectList` function acts like a choice between two branches. The left branch starts with `lock` and the right branch starts with `unlock`. The app exposes both endpoints and proceeds with the branch that receives an answer first. So, if you call the `lock` endpoint in one of the simulated wallets, it will call `lockFunds` and ignore the `unlock` side of the contract. The `forever` call, which runs the application in an infinite loop, is necessary for the `Plutus.Trace.Emulator.EmulatorTrace`. If you omit it, you will only be able to trigger a single endpoint after activating the contract, from which the contract instance will close.

You can now compile the contract and create a simulation. The following action sequence results in two transactions that lock the funds and then distribute them to the two recipients.

```
runSplitDataEmulatorTrace :: IO ()
runSplitDataEmulatorTrace = do
    Trace.runEmulatorTraceIO mkSplitDataEmulatorTrace

mkSplitDataEmulatorTrace :: Trace.EmulatorTrace ()
mkSplitDataEmulatorTrace = do
    -- w1, w2, w3, ... are predefined mock wallets used for testing
    let w1Addr = mockWalletAddress w1
    let w2Addr = mockWalletAddress w2

    h <- Trace.activateContractWallet w1 splitPlutusApp
    Trace.callEndpoint @"lock" h $ LockArgs w1Addr w2Addr 10_000_000
    void Trace.nextSlot
    Trace.callEndpoint @"unlock" h $ LockArgs w1Addr w2Addr 10_000_000
```

You should see an output similar to what follows:

```
[INFO] Slot 0: TxnValidate␣
↪d0f5b08cc20688becb8eceba9770a18ea49a49d5df159715b899736bd1d1121d [  ]
[INFO] Slot 1: 00000000-0000-4000-8000-000000000000 {Wallet W[1]}:
                 Contract instance started
[INFO] Slot 1: 00000000-0000-4000-8000-000000000000 {Wallet W[1]}:
                 Receive endpoint call on 'lock' for Object (fromList [("contents",
↪Array [Object (fromList [("getEndpointDescription",String "lock")]),Object␣
↪(fromList [("unEndpointValue",Object (fromList [("recipient1Address",String "addr_
↪test1vz3vyrrh3pavu8xescvnunn4h27cny70645etn2ulnnqnssrz8utc"),("recipient2Address",
↪String "addr_test1vzq2fazm26ug6yfemg3mcnpuwhkx6v558sy87fgtscvnefckqs3wk"),("totalAda
↪",Object (fromList [("getLovelace",Number 1.0e7)]))]))])]),("tag",String
↪"ExposeEndpointResp")])
```

(continues on next page)

(continued from previous page)

```
[INFO] Slot 1: 00000000-0000-4000-8000-000000000000 {Wallet W[1]}:
                 Contract log: String "Locking Lovelace {getLovelace = 10000000}"
[INFO] Slot 1: W[1]: Balancing an unbalanced transaction:
                       Tx:
                         Tx␣
↪7733b05c8a3d6eb7ade1182beea8b1c1ad7440e7400ef238f7ffeab21e94cd9c:
                           {inputs:
                           reference inputs:
                           collateral inputs:
                           outputs:
                             - 10000000 lovelace addressed to
                               ScriptCredential:␣
↪3e4f54085c2eb253b81fb958f3c3369ab6139c12964ee894ae57a908 (no staking credential)
                               with datum hash␣
↪43492163ee71f886ebc65c85f3dfa8db313f00d701b433b539811464d4355873
                           mint:
                           fee: 0 lovelace
                           validity range: Interval {ivFrom = LowerBound NegInf True,␣
↪ivTo = UpperBound PosInf True}
                           data:
                             (␣
↪43492163ee71f886ebc65c85f3dfa8db313f00d701b433b539811464d4355873
                             , <<<"\162\194\fw\136z\206\FS\217\134\EM>Nu\186\189\137\
↪147\207\213i\149\205\\\252\230\t\194">,
                             <>>,
                             <<"\128\164\244[V\184\141\DC19\218#\188L<u\236m2\148<\b\
↪DEL%\v\134\EM<\167">,
                             <>>,
                             10000000> )
                           redeemers:}
                       Requires signatures:
                       Utxo index:
[INFO] Slot 1: W[1]: Finished balancing:
                       Tx␣
↪3aed0c9c37edee742d00559de3471f4ad6b791522ba224c17fe188a0efcdcda5:
                         {inputs:
                           -␣
↪d0f5b08cc20688becb8eceba9770a18ea49a49d5df159715b899736bd1d1121d!50

                           -␣
↪d0f5b08cc20688becb8eceba9770a18ea49a49d5df159715b899736bd1d1121d!51

                         reference inputs:
                         collateral inputs:
                         outputs:
                           - 10000000 lovelace addressed to
                             ScriptCredential:␣
↪3e4f54085c2eb253b81fb958f3c3369ab6139c12964ee894ae57a908 (no staking credential)
                             with datum hash␣
↪43492163ee71f886ebc65c85f3dfa8db313f00d701b433b539811464d4355873
                           - 9821079 lovelace addressed to
                             PubKeyCredential:␣
↪a2c20c77887ace1cd986193e4e75babd8993cfd56995cd5cfce609c2 (no staking credential)
                         mint:
                         fee: 178921 lovelace
                         validity range: Interval {ivFrom = LowerBound NegInf True,␣
↪ivTo = UpperBound PosInf True}
```

(continues on next page)

(continued from previous page)

```
                        data:
                          (␣
↪43492163ee71f886ebc65c85f3dfa8db313f00d701b433b539811464d4355873
                          , <<<"\162\194\fw\136z\206\FS\217\134\EM>Nu\186\189\137\
↪147\207\213i\149\205\\\252\230\t\194">,
                          <>>,
                          <<"\128\164\244[V\184\141\DC19\218#\188L<u\236m2\148<\b\DEL
↪%\v\134\EM<\167">,
                          <>>,
                          10000000> )
                        redeemers:}
[INFO] Slot 1: W[1]: Signing tx:␣
↪3aed0c9c37edee742d00559de3471f4ad6b791522ba224c17fe188a0efcdcda5
[INFO] Slot 1: W[1]: Submitting tx:␣
↪3aed0c9c37edee742d00559de3471f4ad6b791522ba224c17fe188a0efcdcda5
[INFO] Slot 1: W[1]: TxSubmit:␣
↪3aed0c9c37edee742d00559de3471f4ad6b791522ba224c17fe188a0efcdcda5
[INFO] Slot 1: TxnValidate␣
↪3aed0c9c37edee742d00559de3471f4ad6b791522ba224c17fe188a0efcdcda5 [  ]
[INFO] Slot 2: 00000000-0000-4000-8000-000000000000 {Wallet W[1]}:
               Receive endpoint call on 'unlock' for Object (fromList [("contents",
↪Array [Object (fromList [("getEndpointDescription",String "unlock")]),Object␣
↪(fromList [("unEndpointValue",Object (fromList [("recipient1Address",String "addr_
↪test1vz3vyrrh3pavu8xescvnunn4h27cny70645etn2ulnnqnssrz8utc"),("recipient2Address",
↪String "addr_test1vzq2fazm26ug6yfemg3mcnpuwhkx6v558sy87fgtscvnefckqs3wk"),("totalAda
↪",Object (fromList [("getLovelace",Number 1.0e7)]))])])]),("tag",String
↪"ExposeEndpointResp")])
[INFO] Slot 2: W[1]: Balancing an unbalanced transaction:
                      Tx:
                        Tx␣
↪91ed39867cbcc307d0beb619215e1c138e726105024dbb6668e5ffbfdd2fd754:
                          {inputs:
                            -␣
↪3aed0c9c37edee742d00559de3471f4ad6b791522ba224c17fe188a0efcdcda5!0

                          reference inputs:
                          collateral inputs:
                          outputs:
                            - 5000000 lovelace addressed to
                              PubKeyCredential:␣
↪a2c20c77887ace1cd986193e4e75babd8993cfd56995cd5cfce609c2 (no staking credential)
                            - 5000000 lovelace addressed to
                              PubKeyCredential:␣
↪80a4f45b56b88d1139da23bc4c3c75ec6d32943c087f250b86193ca7 (no staking credential)
                          mint:
                          fee: 0 lovelace
                          validity range: Interval {ivFrom = LowerBound NegInf True,␣
↪ivTo = UpperBound PosInf True}
                          data:
                            (␣
↪43492163ee71f886ebc65c85f3dfa8db313f00d701b433b539811464d4355873
                            , <<<"\162\194\fw\136z\206\FS\217\134\EM>Nu\186\189\137\
↪147\207\213i\149\205\\\252\230\t\194">,
                            <>>,
                            <<"\128\164\244[V\184\141\DC19\218#\188L<u\236m2\148<\b\
↪DEL%\v\134\EM<\167">,
                            <>>,
```

(continues on next page)

(continued from previous page)

```
                              10000000> )
                            redeemers:
                              RedeemerPtr Spend 0 : Constr 0 []
                            attached scripts:
                              PlutusScript PlutusV1 ScriptHash
↪"3e4f54085c2eb253b81fb958f3c3369ab6139c12964ee894ae57a908"}
                        Requires signatures:
                        Utxo index:
                          (␣
↪3aed0c9c37edee742d00559de3471f4ad6b791522ba224c17fe188a0efcdcda5!0
                          , - 10000000 lovelace addressed to
                              ScriptCredential:␣
↪3e4f54085c2eb253b81fb958f3c3369ab6139c12964ee894ae57a908 (no staking credential)
                              with datum hash␣
↪43492163ee71f886ebc65c85f3dfa8db313f00d701b433b539811464d4355873 )
[INFO] Slot 2: W[1]: Finished balancing:
                        Tx␣
↪6156586126d719203a5e22e67360550c8dd3d1565c2afeee576349b7ea84bc09:
                        {inputs:
                          -␣
↪3aed0c9c37edee742d00559de3471f4ad6b791522ba224c17fe188a0efcdcda5!0

                          -␣
↪d0f5b08cc20688becb8eceba9770a18ea49a49d5df159715b899736bd1d1121d!52

                        reference inputs:
                        collateral inputs:
                          -␣
↪d0f5b08cc20688becb8eceba9770a18ea49a49d5df159715b899736bd1d1121d!52

                        outputs:
                          - 5000000 lovelace addressed to
                            PubKeyCredential:␣
↪a2c20c77887ace1cd986193e4e75babd8993cfd56995cd5cfce609c2 (no staking credential)
                          - 5000000 lovelace addressed to
                            PubKeyCredential:␣
↪80a4f45b56b88d1139da23bc4c3c75ec6d32943c087f250b86193ca7 (no staking credential)
                          - 9595609 lovelace addressed to
                            PubKeyCredential:␣
↪a2c20c77887ace1cd986193e4e75babd8993cfd56995cd5cfce609c2 (no staking credential)
                        return collateral:
                          - 9393413 lovelace addressed to
                            PubKeyCredential:␣
↪a2c20c77887ace1cd986193e4e75babd8993cfd56995cd5cfce609c2 (no staking credential)
                        total collateral: 606587 lovelace
                        mint:
                        fee: 404391 lovelace
                        validity range: Interval {ivFrom = LowerBound NegInf True,␣
↪ivTo = UpperBound PosInf True}
                        data:
                          (␣
↪43492163ee71f886ebc65c85f3dfa8db313f00d701b433b539811464d4355873
                          , <<<"\162\194\fw\136z\206\FS\217\134\EM>Nu\186\189\137\
↪147\207\213i\149\205\\\252\230\t\194">,
                          <>>,
                          <<"\128\164\244[V\184\141\DC19\218#\188L<u\236m2\148<\b\DEL
↪%\v\134\EM<\167">,
```

(continues on next page)

(continued from previous page)

```
                          <>>,
                          10000000> )
                        redeemers:
                          RedeemerPtr Spend 0 : Constr 0 []
                        attached scripts:
                          PlutusScript PlutusV1 ScriptHash
↪"3e4f54085c2eb253b81fb958f3c3369ab6139c12964ee894ae57a908"}
[INFO] Slot 2: W[1]: Signing tx:␣
↪6156586126d719203a5e22e67360550c8dd3d1565c2afeee576349b7ea84bc09
[INFO] Slot 2: W[1]: Submitting tx:␣
↪6156586126d719203a5e22e67360550c8dd3d1565c2afeee576349b7ea84bc09
[INFO] Slot 2: W[1]: TxSubmit:␣
↪6156586126d719203a5e22e67360550c8dd3d1565c2afeee576349b7ea84bc09
[INFO] Slot 2: TxnValidate␣
↪6156586126d719203a5e22e67360550c8dd3d1565c2afeee576349b7ea84bc09 [ Data decoded␣
↪successfully
                                                                                ␣
↪      , Redeemer decoded successfully
                                                                                ␣
↪      , Script context decoded successfully ]
Final balances
Wallet 7: 100000000 lovelace
Wallet 8: 100000000 lovelace
Wallet 6: 100000000 lovelace
Wallet 4: 100000000 lovelace
Wallet 2: 105000000 lovelace
Wallet 1: 94416688 lovelace
Wallet 10: 100000000 lovelace
Wallet 9: 100000000 lovelace
Wallet 3: 100000000 lovelace
Wallet 5: 100000000 lovelace
```

Exercise

1. Extract the function that assigns funds to each recipient from `unlockFunds` and `validateSplit` to reduce redundancy in the code
2. Extend the contract to deal with a list of recipients instead of a fixed number of 2.

5.2.2 Extending the basic Plutus app with the constraints API

The previous tutorial (see *Writing a basic Plutus app in an emulated environment*) showed you how to write a Plutus app that locks some Ada in a script output and splits them evenly between two recipients. In this tutorial, we will reuse the same example, but we will use instead the constraints API which will be used to generate the on-chain and off-chain part of the Plutus app. This will allow your application to create a transaction which is *mostly* consistent with the validator function.

Given a *SplitData*, let's start by defining a function which generates the constraints to unlock funds locked by the split validator.

```
-- | Create constraints that will be used to spend a locked transaction output
-- from the script address.
--
-- These constraints will be used in the validation script as well as in the
```

(continues on next page)

(continued from previous page)

```
-- transaction creation step.
{-# INLINABLE splitDataConstraints #-}
splitDataConstraints :: SplitData -> TxConstraints () SplitData
splitDataConstraints SplitData{recipient1, recipient2, amount} =
            Constraints.mustPayToAddress recipient1 (Ada.toValue half)
  `mappend` Constraints.mustPayToAddress recipient2 (Ada.toValue $ amount - half)
 where
     half = Ada.divide amount 2
```

With the constraints, let's start by defining the validator function.

```
-- | The validation logic is generated with `checkScriptContext` based on the set
-- of constraints.
{-# INLINABLE validateSplit #-}
validateSplit :: SplitData -> () -> ScriptContext -> Bool
validateSplit splitData _ =
    Constraints.checkScriptContext (splitDataConstraints splitData)
```

As you can see, it's much simpler than the original version.

Now to the off-chain part. The *lock* endpoint doesn't change. However, we can change the *unlock* endpoint to use the constraints we defined above.

```
unlock :: Promise () SplitSchema T.Text ()
unlock = endpoint @"unlock" (unlockFunds . mkSplitData)

-- | Creates a transaction which spends all script outputs from a script address,
-- sums the value of the scripts outputs and splits it between two payment keys.
unlockFunds :: SplitData -> Contract () SplitSchema T.Text ()
unlockFunds splitData = do
    networkId <- pNetworkId <$> getParams
    -- Get the address of the Split validator
    let contractAddress = Scripts.validatorCardanoAddress networkId splitValidator
    -- Get all utxos that are locked by the Split validator
    utxos <- utxosAt contractAddress
    -- Generate constraints which will spend all utxos locked by the Split
    -- validator and split the value evenly between the two payment keys.
    let constraints = Constraints.collectFromTheScript utxos ()
                      <> splitDataConstraints splitData
    -- Create, Balance and submit the transaction
    void $ submitTxConstraintsSpending splitValidator utxos constraints
```

That's it! The rest of the contract is the same as the previous tutorial.

5.2.3 Property-based testing of Plutus contracts

Plutus comes with a library for testing contracts using QuickCheck. Tests generated by this library perform a sequence of calls to *contract endpoints*, checking that *tokens* end up in the correct wallets at the end of each test. These sequences can be generated at random, or in a more directed way to check that desirable states always remain reachable. This tutorial introduces the testing library by walking through a simple example: a contract that implements a guessing game.

An overview of the guessing game

The source code of the guessing game contract is provided as an example here, and the final test code is here.

The game is played as follows:

- The first player locks a sum of Ada in the contract, which is donated as a prize. The prize is protected by a secret password.
- Any player can now try to guess the password, by submitting a 'guess' transaction that attempts to withdraw some or all of the prize, along with a guess at the password, and a new password to replace the old one. If the guess is correct, and the contract contains enough Ada, then the guesser receives the withdrawal and the remainder of the prize is now protected by the new password. If the guess is wrong, the transaction is not accepted, and nothing changes.

As an extra wrinkle, when the first player locks the prize, a new *token* is also minted. Only the player currently holding the *token* is allowed to make a guess–which gives us an opportunity to illustrate minting and passing around *tokens*.

The generated tests will exercise the contract by locking the prize, then moving the game *token* and making guesses at random, checking that the game *token* and Ada move as they should.

Emulated wallets

To test contracts, we need emulated wallets. These and many other useful definitions for testing can be imported via

```
import Plutus.Contract.Test (Wallet, minLogLevel, mockWalletAddress, w1, w2, w3)
```

Now we can create a number of wallets: in this tutorial, we'll settle for three:

```
wallets :: [Wallet]
wallets = [w1, w2, w3]
```

Values and tokens

Wallets contain 'values', which are mixtures of different quantities of one or more types of *token*. The most common *token* is, of course, the Ada; we can import functions manipulating Ada, and the `Value` type itself, as follows:

```
import Ledger.Address qualified as Address
import Plutus.Script.Utils.Ada qualified as Ada
import Plutus.Script.Utils.Value qualified as Value
```

With these imports, we can construct values in the Ada *currency*:

```
> Ada.lovelaceValueOf 1
Value (Map [(,Map [(,1)])])
```

We will also need a game *token*. After importing the `Scripts` module

```
import Ledger.Typed.Scripts qualified as Scripts
```

we can define it as follows, applying a minting policy defined in the code under test (imported as module `G`):

```
import Plutus.Contracts.GameStateMachine qualified as G
```

```
import Cardano.Node.Emulator.TimeSlot qualified as TimeSlot
```

```
import Data.Default (Default (def))
```

```
gameParam :: G.GameParam
gameParam = G.GameParam (Address.toPlutusAddress $ mockWalletAddress w1) (TimeSlot.
↪scSlotZeroTime def)
```

```
guessTokenVal :: Value.Value
guessTokenVal =
    let sym = Scripts.forwardingMintingPolicyHash $ G.typedValidator gameParam
    in G.token sym "guess"
```

The value of the *token* is (with long hash values abbreviated):

```
> guessTokenVal
Value (Map [(f687...,Map [(guess,1)])])
```

We can even construct a `Value` containing an Ada and a game *token*:

```
> Ada.lovelaceValueOf 1 <> guessTokenVal
Value (Map [(,Map [(,1)]),(f687...,Map [(guess,1)])])
```

If you inspect the output closely, you will see that a `Value` contains maps nested within another `Map`. The outer `Map` is indexed by hashes of minting policy *scripts*, so each inner `Map` contains a bag of *tokens* managed by the same policy. *Token* names can be chosen freely, and each policy can manage any number of its own *token* types. In this case the game *token* is called a "guess", and the *script* managing game *tokens* has the hash f687... A little confusingly, the Ada *token* name is displayed as an empty string, as is the hash of the corresponding minting policy.

Introducing contract models

We test contracts using a *model* of the system, which includes the state(s) of all the agents involved–the on-chain state, the off-chain state on your computer, the off-chain state on anyone else's computer–everything relevant to the contract(s) under test. The first job to be done is thus defining that model. To do so, we import the contract modelling library

```
import Plutus.Contract.Test.ContractModel qualified as CM
```

and define the model type:

```
data GameModel = GameModel
```

This definition is incomplete: we shall fill in further details as we proceed.

The `GameModel` type must be an instance of the `Plutus.Contract.Test.ContractModel.Interface.ContractModel` class, which has an associated datatype defining the kinds of *actions* that will be performed in generated tests.

```
instance CM.ContractModel GameModel where

    data Action GameModel = Lock      Wallet String Integer
                          | Guess     Wallet String String Integer
                          | GiveToken Wallet
        deriving (Eq, Show, Generic)
```

In this case we define three actions:

- a `Lock` action to be performed by the first player when starting the game, containing the player's wallet (from which the Ada will be taken), the secret password, and the prize amount.
- a `Guess` action to be performed by the other players, containing the player's wallet (to receive the prize), the player's guess, a new password, and the amount to be claimed if the guess is right.
- a `GiveToken` action, to give the game *token* to a player so they can make a guess.

A generated test is called `Plutus.Contract.Test.ContractModel.Interface.Actions`, and is, as the name suggests, essentially a sequence of `Plutus.Contract.Test.ContractModel.Interface.Action` values. We can run tests by using `Plutus.Contract.Test.ContractModel.Interface.propRunActions_`:

```
prop_Game :: CM.Actions GameModel -> Property
prop_Game actions = CM.propRunActions_ actions
```

When we test this property, `quickCheck` will generate random action sequences to be tested, checking at the end of each test that *tokens* are transferred correctly, and contracts didn't crash.

> **Note:** There is also a more general function `Plutus.Contract.Test.ContractModel.Interface.propRunActions` that allows the check at the end of each test to be customized.

But how does `quickCheck` know what code to run when you check `prop_Game`? `Plutus.Contract.Test.ContractModel.Interface.propRunActions_` needs to create a handle for each contract instance, which is used to invoke their *endpoints* from the test. Different contracts have different *endpoints*, of different types–and thus different *schemas*. When we invoke an *endpoint*, we need to know the *schema* of the contract we are invoking, and the type of errors it can return, so that the type-checker can ensure that the call is valid. We thus need to know the *type* of contract that each handle refers to.

To achieve this, every contract instance in a test is *named* by a `Plutus.Contract.Test.ContractModel.Interface.ContractInstanceKey`, another associated datatype of the `Plutus.Contract.Test.ContractModel.Interface.ContractModel` class; we talk to a contract instance by referring to its `Plutus.Contract.Test.ContractModel.Interface.ContractInstanceKey`. The `Plutus.Contract.Test.ContractModel.Interface.ContractInstanceKey` type is parameterised *both* on the type of the contract model, and on the observable state, *schema*, and error type of the contract it refers to. Since the same test may refer to contracts of several different types, `Plutus.Contract.Test.ContractModel.Interface.ContractInstanceKey` is defined as a GADT.

In this particular case, there is only one type of contract under test, and so it suffices to define a `Plutus.Contract.Test.ContractModel.Interface.ContractInstanceKey` type with a single constructor. There is one contract instance running in each emulated wallet, so we simply distinguish contract instance keys by the wallet they are running in:

```
    data ContractInstanceKey GameModel w schema err param where
        WalletKey :: Wallet -> CM.ContractInstanceKey GameModel () G.
↪GameStateMachineSchema G.GameError ()
```

Once this type is defined, we can tell QuickCheck what code to run for a given contract by filling in the `Plutus.Contract.Test.ContractModel.Interface.initialInstances`, `Plutus.Contract.Test.ContractModel.Interface.instanceWallet`, and `Plutus.Contract.Test.ContractModel.Interface.instanceContract` fields of the `ContractModel` class:

```
    initialInstances = (`CM.StartContract` ()) . WalletKey <$> wallets

    instanceContract _ WalletKey{} _ = G.contract
```

(continues on next page)

(continued from previous page)

```
    instanceWallet (WalletKey w) = w
```

This specifies (reading top to bottom) that we should create one contract instance per wallet `w`, that will run `G.contract`, in wallet `w`.

Now we can run tests, although of course they will not yet succeed:

```
> quickCheck prop_Game
*** Failed! (after 1 test and 1 shrink):
Exception:
  GSM.hs:65:10-32: No instance nor default method for class operation arbitraryAction
```

The contract modelling library cannot generate test cases, unless *we* specify how to generate an `Plutus.Contract.Test.ContractModel.Interface.Action`, which we will do next.

Generating actions

To generate actions, we need to be able to generate wallets, guesses, and suitable values of Ada, since these appear as action parameters.

```
genWallet :: Gen Wallet
genWallet = elements wallets

genGuess :: Gen String
genGuess = elements ["hello", "secret", "hunter2", "*******"]

genValue :: Gen Integer
genValue = getNonNegative <$> arbitrary
```

We choose wallets from the three available, and we choose passwords from a small set, so that random guesses will often be correct. We choose Ada amounts to be non-negative integers, because negative amounts would be error cases that we choose not to test.

Now we can define a generator for `Plutus.Contract.Test.ContractModel.Interface.Action`, as a method of the `Plutus.Contract.Test.ContractModel.Interface.ContractModel` class:

```
    arbitraryAction s = oneof $
        [ Lock      <$> genWallet <*> genGuess <*> genValue              ] ++
        [ Guess     <$> genWallet <*> genGuess <*> genGuess <*> genValue ] ++
        [ GiveToken <$> genWallet                                        ]
```

With this method defined, we can start to generate test cases. Using `sample` we can see what action sequences look like:

```
> sample (arbitrary :: Gen (Actions GameModel))
Actions
  [Lock (Wallet 2) "hunter2" 5,
   Guess (Wallet 3) "*******" "hello" 6,
   Guess (Wallet 1) "secret" "*******" 10,
   Guess (Wallet 3) "*******" "*******" 6,
   GiveToken (Wallet 3),
   Guess (Wallet 2) "hunter2" "hunter2" 15]
.
.
```

We can even run 'tests' now, although they don't do much yet:

```
> quickCheck prop_Game
+++ OK, passed 100 tests:

Actions (2263 in total):
33.94% Lock
33.89% Guess
32.17% GiveToken
```

The output tells us the distribution of generated actions, aggregated across all the tests. We can see that each action was generated around one third of the time, which is to be expected since our generator does not weight them at all. Keep an eye on this table as we extend our generation; if any `Plutus.Contract.Test.ContractModel.Interface.Action` disappears altogether, or is generated very rarely, then this indicates a problem in our tests.

Modelling expectations

The ultimate purpose of our tests is to check that funds are transferred correctly by each operation–for example, that after a guess, the guesser receives the requested Ada only if the guess was correct. An important part of a `Plutus.Contract.Test.ContractModel.Interface.ContractModel` defines how funds are expected to move. However, it's clear that in order to define how we expect funds to move after a `Guess`, we need to know more than just where all the Ada are. We need to know:

- what the current secret password is, so we can decide whether or not the guess is correct.
- whether or not the guesser currently holds the game *token*, and so is entitled to make a guess.
- how much Ada is currently locked in the contract, so we can determine whether the guesser is requesting funds that actually exist.

These all depend on the previous steps in the test case. To keep track of such information, we store it in a *contract state*, which is the type parameter of the `Plutus.Contract.Test.ContractModel.Interface.ContractModel` class. (Note that this contract state is a part of the *model*, it may be quite different from the contract state in the implementation). In this case the contract state is the `GameModel` type, so let's complete its definition:

```
data GameModel = GameModel
    { _gameValue     :: Integer
    , _hasToken      :: Maybe Wallet
    , _currentSecret :: String }
  deriving (Show, Generic)

makeLenses 'GameModel
```

Initially the game *token* does not exist, so we record its current owner as a `Maybe Wallet`, so that we can represent the initial situation before its creation. The locked funds are always in Ada, so in the model it suffices to store an integer.

Now we can define the initial state of the model at the start of each test case, `Plutus.Contract.Test.ContractModel.Interface.initialState`, and a `Plutus.Contract.Test.ContractModel.Interface.nextState` function to model the way we expect each operation to change the state. These are both methods in the `Plutus.Contract.Test.ContractModel.Interface.ContractModel` class.

The initial state just records that the game *token* does not exist yet, and assigns default values to the other fields.

```
    initialState = GameModel
        { _gameValue     = 0
```

(continues on next page)

(continued from previous page)

```
        , _hasToken      = Nothing
        , _currentSecret = ""
        }
```

The `Plutus.Contract.Test.ContractModel.Interface.nextState` function is defined in the `Plutus.Contract.Test.ContractModel.Interface.Spec` monad

```
nextState :: CM.Action state -> CM.Spec state ()
```

and defines the expected effect of each operation.

The `Lock` operation creates the contract, initializing the model contract state (using `%=` and generated `Lens` operations), mints the game *token* (using `Plutus.Contract.Test.ContractModel.Interface.mint`), deposits it in the creator's wallet, and withdraws the Ada locked in the contract (using `Plutus.Contract.Test.ContractModel.Interface.deposit` and `Plutus.Contract.Test.ContractModel.Interface.withdraw`):

```
    nextState (Lock w secret val) = do
        hasToken      .= Just w
        currentSecret .= secret
        gameValue     .= val
        CM.mint guessTokenVal
        CM.deposit  w guessTokenVal
        CM.withdraw w $ Ada.lovelaceValueOf val
```

A `Plutus.Contract.Test.ContractModel.Interface.ContractModel` actually tracks not only the contract model state (in our case the `GameModel` type), but also the quantities of *tokens* expected to be in each wallet, which are checked at the end of each test. It is these expectations that are manipulated by `Plutus.Contract.Test.ContractModel.Interface.mint`, `Plutus.Contract.Test.ContractModel.Interface.deposit`, etc... don't confuse them with operations that *actually* mint or move *tokens* in the implementation. The `Plutus.Contract.Test.ContractModel.Interface.ModelState` type contains all of this information.

When making a guess, we need to check parts of the contract state (which we read using `Plutus.Contract.Test.ContractModel.Interface.viewContractState`), and then we update the stored password, game value, and wallet contents appropriately.

```
    nextState (Guess w old new val) = do
        correctGuess <- (old ==)    <$> CM.viewContractState currentSecret
        holdsToken   <- (Just w ==) <$> CM.viewContractState hasToken
        enoughAda    <- (val <=)    <$> CM.viewContractState gameValue
        when (correctGuess && holdsToken && enoughAda) $ do
            currentSecret .= new
            gameValue     %= subtract val
            CM.deposit w $ Ada.lovelaceValueOf val
```

`GiveToken` just transfers the game *token* from one wallet to another using `Plutus.Contract.Test.ContractModel.Interface.transfer`.

```
    nextState (GiveToken w) = do
        w0 <- fromJust <$> CM.viewContractState hasToken
        CM.transfer w0 w guessTokenVal
        hasToken .= Just w
```

At the end of each test, the `Plutus.Contract.Test.ContractModel.Interface.ContractModel` framework checks that every wallet contains the *tokens* that the model says it should.

We can exercise the `Plutus.Contract.Test.ContractModel.Interface.nextState` function already by generating and 'running' tests, even though we have not yet connected these tests to the real contract. Doing so immediately reveals a problem:

```
> quickCheck prop_Game
*** Failed! (after 3 tests and 3 shrinks):
Exception:
  Maybe.fromJust: Nothing
  CallStack (from HasCallStack):
    error, called at libraries/base/Data/Maybe.hs:148:21 in base:Data.Maybe
    fromJust, called at GSM0.hs:122:15 in main:GSM0
Actions
 [GiveToken (Wallet 1)]
```

Looking at the last two lines, we see the generated test sequence, and the problem is evident: we generated a test *that only gives the game :term:`token`* to wallet 1, but this makes no sense because the game *token* has not yet been minted–so the `fromJust` in the `Plutus.Contract.Test.ContractModel.Interface.nextState` function fails. We will see how to prevent this in the next section.

Restricting test cases with preconditions

As we just saw, not every sequence of actions makes sense as a test case; we need a way to *restrict* test cases to be 'sensible'. Note this is *not* the same as restricting tests to 'the happy path': we *want* to test unexpected sequences of actions, and indeed, this is part of the strength of property-based testing. But there are some actions–like trying to give the game *token* to a wallet before it has been minted–that are not even interesting to test. These are the cases that we rule out by defining preconditions for actions; the effect is to prevent such test cases ever being generated.

To introduce preconditions, we add a definition of the `Plutus.Contract.Test.ContractModel.Interface.precondition` method to our `Plutus.Contract.Test.ContractModel.Interface.ContractModel` instance.

```
precondition :: CM.ModelState state -> CM.Action state -> Bool
```

The `Plutus.Contract.Test.ContractModel.Interface.precondition` is parameterised on the entire model state, which includes the contents of wallets as well as our contract state, so we will need to extract this state as well as the fields we need from it. For now, we just restrict `GiveToken` actions to states in which the token exists:

```
    precondition s (GiveToken _) = isJust tok
        where
            tok = s ^. CM.contractState . hasToken
    precondition s _             = True
```

Now if we try to run tests, something more interesting happens:

```
> quickCheck prop_Game
*** Failed! Assertion failed (after 2 tests):
Actions
 [Lock (Wallet 1) "hello" 0]
Expected funds of W1 to change by
  Value (Map [(f687...,Map [(guess,1)])])
but they did not change
Test failed.
Emulator log:
[INFO] Slot 1: TxnValidate 4feb...
[INFO] Slot 1: 00000000-0000-4000-8000-000000000000 {Contract instance for wallet 1}:
```

(continues on next page)

(continued from previous page)

```
                Contract instance started
[INFO] Slot 1: 00000000-0000-4000-8000-000000000001 {Contract instance for wallet 2}:
                Contract instance started
[INFO] Slot 1: 00000000-0000-4000-8000-000000000002 {Contract instance for wallet 3}:
                Contract instance started
```

The test has failed, of course. The generated (and simplified) test case only performs one action:

```
Actions
 [Lock (Wallet 1) "hello" 0]
```

Wallet 1 attempts to create a game contract guarding zero Ada. Inspecting the error message, we can see that wallet 1 ended up with the wrong contents:

```
Expected funds of W1 to change by
  Value (Map [(f687...,Map [(guess,1)])])
but they did not change
```

Our model predicted that wallet 1 would end up containing the game *token*, but in fact its contents were unchanged.

In this test, we have actually performed actions in the emulator, as the log shows us: one transaction has been validated, and we have started three contract instances (one for each wallet in the test). But we have *not* created a game *token* for wallet 1, because thus far we have not defined how actions in a test should be performed–so the `Lock` action in the test case behaves as a no-op, which of course does not deposit a game *token* in wallet 1. It is time to link actions in a test to the emulator.

Performing actions

So far we are generating actions, but we have not yet linked them to the contract they are supposed to test–so 'running' the tests, as we did above, did not invoke the contract at all. To do so, we must import the emulator

```
import Plutus.Trace.Emulator qualified as Trace
```

Then we define the `Plutus.Contract.Test.ContractModel.Interface.perform` method of the `Plutus.Contract.Test.ContractModel.Interface.ContractModel` class:

```
 perform
     :: CM.HandleFun state
     -> (CM.SymToken -> Value.AssetClass)
     -> CM.ModelState state
     -> CM.Action state
     -> CM.SpecificationEmulatorTrace ()
```

The job of the `Plutus.Contract.Test.ContractModel.Interface.perform` method in this case is just to invoke the contract end-points, using the API defined in the code under test, and transfer the game *token* from one wallet to another as specified by `GiveToken` actions.

```
gameParam :: G.GameParam
gameParam = G.GameParam (Address.toPlutusAddress $ mockWalletAddress w1) (TimeSlot.
↪scSlotZeroTime def)
```

```
    perform handle _ s cmd = case cmd of
        Lock w new val -> do
            Trace.callEndpoint @"lock" (handle $ WalletKey w)
```

(continues on next page)

(continued from previous page)

```
                G.LockArgs
                    { G.lockArgsGameParam = gameParam
                    , G.lockArgsSecret = secretArg new
                    , G.lockArgsValue = Ada.lovelaceValueOf val
                    }
        Guess w old new val -> do
            Trace.callEndpoint @"guess" (handle $ WalletKey w)
                G.GuessArgs
                    { G.guessArgsGameParam = gameParam
                    , G.guessTokenTarget   = Address.toPlutusAddress $␣
↪mockWalletAddress w
                    , G.guessArgsOldSecret = old
                    , G.guessArgsNewSecret = secretArg new
                    , G.guessArgsValueTakenOut = Ada.lovelaceValueOf val
                    }
        GiveToken w' -> do
            let w = fromJust (s ^. CM.contractState . hasToken)
            Trace.payToWallet w w' guessTokenVal
            return ()
```

Every call to an end-point must be associated with one of the contract instances defined in our `initialInstances`; the `handle` argument to `Plutus.Contract.Test.ContractModel.Interface.perform` lets us find the contract handle associated with each `Plutus.Contract.Test.ContractModel.Interface.ContractInstanceKey`.

For the most part, it is good practice to keep the `Plutus.Contract.Test.ContractModel.Interface.perform` function simple: a direct relationship between actions in a test case and calls to *contract endpoints* makes interpreting test failures much easier.

Note: Helping shrinking work better by choosing test case actions well

In the definition of `Plutus.Contract.Test.ContractModel.Interface.perform` above, the `GiveToken` action is a little surprising: when we call the emulator, we have to specify not only the wallet to give the *token to*, but also the wallet to take the *token from*. So why did we choose to define a `GiveToken w` action to include in test cases, rather than an action `PassToken w w'`, which would correspond more directly to the code in `Plutus.Contract.Test.ContractModel.Interface.perform`?

The answer is that using `GiveToken` actions instead helps QuickCheck to shrink failing tests more effectively. QuickCheck shrinks test cases by attempting to remove actions from them–essentially replacing an action by a no-op. But consider a sequence such as

```
PassToken w1 w2
PassToken w2 w3
```

which transfers the game *token* in two steps from wallet 1 to wallet 3. Deleting either one of these steps means the game *token* will end up in the wrong place, probably causing the next steps in the test to behave very differently (and thus, preventing this shrinking step). But given the sequence

```
GiveToken w2
GiveToken w3
```

the first `GiveToken` can be deleted without affecting the behaviour of the second at all. Thus, by making *token*-passing steps independent of each other, we make it easier for QuickCheck to shrink a failing test without drastic changes to its behaviour.

Shrinking Actions

Before starting to run tests seriously, it is useful to make sure that any failing tests will shrink well to small examples. By default, the contract modelling library tries to shrink tests by removing actions, but it cannot know how to shrink the actions themselves. We can specify this shrinking by defining the `Plutus.Contract.Test.ContractModel.Interface.shrinkAction` operation in the `Plutus.Contract.Test.ContractModel.Interface.ContractModel` class:

```
shrinkAction :: CM.ModelState state -> CM.Action state -> [CM.Action state]
```

This function returns a list of 'simpler' actions that should be tried as replacements for the given `Plutus.Contract.Test.ContractModel.Interface.Action`, when QuickCheck is simplifying a failed test. In this case we define a shrinking function for wallets:

```
shrinkWallet :: Wallet -> [Wallet]
shrinkWallet w = [w' | w' <- wallets, w' < w]
```

and shrink actions by shrinking the wallet and Ada parameters.

```
    shrinkAction _s (Lock w secret val) =
        [Lock w' secret val | w' <- shrinkWallet w] ++
        [Lock w secret val' | val' <- shrink val]
    shrinkAction _s (GiveToken w) =
        [GiveToken w' | w' <- shrinkWallet w]
    shrinkAction _s (Guess w old new val) =
        [Guess w' old new val | w' <- shrinkWallet w] ++
        [Guess w old new val' | val' <- shrink val]
```

We choose not to shrink password/guess parameters, because they are not really significant–one password is as good as another in a failed test.

Debugging the model

At this point, the contract model is complete, and tests are runnable. However, they do not pass, and so we need to adapt either the tests or the contract to resolve the inconsistencies revealed. Testing `prop_Game` now results in:

```
> quickCheck prop_Game
*** Failed! Falsified (after 6 tests and 3 shrinks):
Actions
 [Lock (Wallet 1) "hunter2" 0]
Expected funds of W1 to change by
  Value (Map [(f687...,Map [(guess,1)])])
but they did not change
Test failed.
Emulator log:
... 49 lines of emulator log messages ...
```

In this test, wallet 1 attempts to lock zero Ada, and our model predicts that wallet 1 should receive a game *token*–but this did not happen. To understand why, we need to study the emulator log. Here are the relevant parts:

```
...
[INFO] Slot 1: 00000000-0000-4000-8000-000000000000 {Contract instance for wallet 1}:
                 Receive endpoint call: Object (fromList [("tag",String "lock"),...
[INFO] Slot 1: W1: Balancing an unbalanced transaction:
                     Tx:
```

(continues on next page)

(continued from previous page)

```
                      Tx 2542...:
                        {inputs:
                        outputs:
                          - Value (Map []) addressed to
                            ScriptAddress: d1e1...
...
[INFO] Slot 1: W1: TxSubmit: 2542...
[INFO] Slot 2: TxnValidate 2542...
[INFO] Slot 2: W1: Balancing an unbalanced transaction:
                     Tx:
                       Tx 1eba...:
                         {inputs:
                            - 2542...!0
                              Redeemer: <>
                         outputs:
                           - Value (Map []) addressed to
                             ScriptAddress: d1e1...
                         mint: Value (Map [(f687...,Map [(guess,1)])])
...
[INFO] Slot 2: W1: TxSubmit: 2d66...
```

Here we see the *endpoint* call to `lock` being received during slot 1, resulting in a transaction with ID `2542...`, which pays zero Ada to the contract *script*. The transaction is balanced (which has no effect in this case), submitted, and validated by the emulator at slot 2. Then another transaction, `1eba...`, is created, which mints the game *token*. This transaction is in turn balanced (resulting in a new hash, `2d66...`), and submitted without error–but although no errors are reported, *this transaction is not validated*.

Since the transaction is submitted in slot 2, we would expect it to be validated in slot 3. In fact, the problem here is just that the test stopped too early, before the blockchain had validated this second transaction. The solution is just to delay long enough for the blockchain to validate all the transactions we have submitted.

Adding delays to test cases

To give the blockchain time to validate the transactions generated by a `Lock` call, we need to delay by two slots. Why two? Because the `Lock` *contract endpoint* submits two transactions to the blockchain. Likewise, we delay one slot after each of the other actions. (If the delays we insert are too short, we will discover this later via failed tests).

We add a call to `delay` in each branch of `Plutus.Contract.Test.ContractModel.Interface.perform`:

```
gameParam :: G.GameParam
gameParam = G.GameParam (Address.toPlutusAddress $ mockWalletAddress w1) (TimeSlot.
↪scSlotZeroTime def)
```

```
    perform handle _ s cmd = case cmd of
        Lock w new val -> do
            Trace.callEndpoint @"lock" (handle $ WalletKey w)
                G.LockArgs
                    { G.lockArgsGameParam = gameParam
                    , G.lockArgsSecret    = secretArg new
                    , G.lockArgsValue     = Ada.lovelaceValueOf val
                    }
            CM.delay 2
        Guess w old new val -> do
            Trace.callEndpoint @"guess" (handle $ WalletKey w)
```

(continues on next page)

(continued from previous page)

```
                G.GuessArgs
                    { G.guessArgsGameParam     = gameParam
                    , G.guessTokenTarget       = Address.toPlutusAddress $␣
↪mockWalletAddress w
                    , G.guessArgsOldSecret     = old
                    , G.guessArgsNewSecret     = secretArg new
                    , G.guessArgsValueTakenOut = Ada.lovelaceValueOf val
                    }
            CM.delay 1
        GiveToken w' -> do
            let w = fromJust (s ^. CM.contractState . hasToken)
            Trace.payToWallet w w' guessTokenVal
            CM.delay 1
```

This makes the *emulator* delay one or two slots, but we also need to delay in our *model*, to keep the model state in sync with the emulator. We do this using corresponding calls to `Plutus.Contract.Test.ContractModel.Interface.wait` in the definition of `Plutus.Contract.Test.ContractModel.Interface.nextState`:

```
    nextState (Lock w secret val) = do
        hasToken      .= Just w
        currentSecret .= secret
        gameValue     .= val
        CM.mint guessTokenVal
        CM.deposit  w guessTokenVal
        CM.withdraw w $ Ada.lovelaceValueOf val
        CM.wait 2
```

and similarly in the other cases.

Does this change fix the problem? To find out, we should *rerun* the same test case, after updating the code.

Rerunning a failed test

The best way to save and rerun a QuickCheck test case is to copy-and-paste it from the QuickCheck output into your code. Since `prop_Game` is just a function that takes the generated test as an argument, then we can rerun a test by passing it to the property. In this case let us define

```
testLock :: Property
testLock = flip CM.forAllDL prop_Game $ CM.action $ Lock w1 "hunter2" 0
```

`testLock` is itself a `Property`, so we can test it using `quickCheck`. Testing it *before* adding the delays in the last section generates the same output as before. Testing it *after* the delays are added results in

```
> quickCheck testLock
+++ OK, passed 100 tests.

Actions (100 in total):
100% Lock
```

The test passes, and the problem is fixed.

> **Note:** Since there is no random generation in this test, there is no real need to test it 100 times. This can be avoided by adding `withMaxSuccess` to the definition:

```
testLock :: Property
testLock = withMaxSuccess 1 . flip CM.forAllDL prop_Game $ CM.action $ Lock␣
↪w1 "hunter2" 0
```

Note: We save the **failing test case**, not the random seed used to generate it. This is the only way to be sure that we repeat the *same* test that just failed. Usually, a failed test that QuickCheck reports is the result of both random generation *and shrinking*, not random generation alone. Reusing the same random seed would usually regenerate a much larger test, which might well fail for a different reason, leading QuickCheck to report a different shrunk failing test. It is then impossible to know for sure whether or not the change just made to the code fixed the problem it was intended to fix–it might just have changed the way failed tests shrink. By rerunning exactly the same test case we can be sure that our change did fix that problem, at least.

Controlling the log-level

When we rerun random tests, they fail for a different reason:

```
> quickCheck prop_Game
*** Failed! Assertion failed (after 5 tests and 7 shrinks):
Actions
 [Lock (Wallet 1) "hunter2" 0,
  Lock (Wallet 1) "hello" 0]
Outcome of Contract instance for wallet 1:
  False
Failed 'Contract instance stopped with error'
Test failed.
Emulator log:
... 73 lines of emulator log messages ...
```

Looking at the failing test case,

```
Actions
 [Lock (Wallet 1) "hunter2" 0,
  Lock (Wallet 1) "hello" 0]
```

we can see that it does something unexpected: wallet 1 tries to lock *twice*. Our model allows this, but the error message tells us that the contract instance crashed.

The emulator log output can be rather overwhelming, but we can eliminate the `INFO` messages by running the test sequence with appropriate options. If we define

```
import Control.Monad.Freer.Extras.Log (LogLevel)
```

```
propGame' :: LogLevel -> CM.Actions GameModel -> Property
propGame' l s = CM.propRunActionsWithOptions
                    (set minLogLevel l CM.defaultCheckOptionsContractModel)
                    CM.defaultCoverageOptions
                    (\ _ -> pure True)
                    s
```

then we can re-run the test and see more succinct output:

```
> quickCheck $ propGame' Warning
*** Failed! Assertion failed (after 7 tests and 4 shrinks):
Actions
 [Lock (Wallet 1) "hello" 0,
  Lock (Wallet 1) "*******" 0]
Outcome of Contract instance for wallet 1:
  False
Failed 'Contract instance stopped with error'
Test failed.
Emulator log:
[WARNING] Slot 4: 00000000-0000-4000-8000-000000000000 {Contract instance for wallet␣
↪1}:
                  Contract instance stopped with error: GameSMError (ChooserError
↪"Found 2 outputs, expected 1")
```

Now we see the problem: an error in the game implementation that stopped the second contract call, because two unspent transaction outputs had been created. These two outputs are the Ada amounts addressed to the contract *script* that are created by the first transaction of each call to the `Lock` *endpoint*. The off-chain contract is not designed to cope with more than one such *UTXO*; it is now in a broken state. In fact, the Ada now locked in these *UTXOs* cannot be recovered by the present *off-chain code*–the only way to recover the money is to revise the contract so that it can accept multiple *UTXOs*. Arguably, this is a bug in the contract: if any wallet tries to start the game for a second time, the Ada will be lost (until the bug is fixed).

Refining preconditions

We just learned that a second `Lock` call puts the contract into a broken state. But this is not how the game was intended to be used, so the developer might reasonably respond "you shouldn't do that". There could also be other problems in the code that we cannot presently find, because they are masked by the double-lock bug. Since a test case with two `Lock` calls is easy to generate, then QuickCheck is likely to report this particular problem in almost every subsequent run–unless we explicitly prevent it from doing so.

We can easily avoid this by *strengthening the precondition* of `Lock`, so that it can only be performed once per test case. We do so by checking whether any wallet holds the game *token*:

```
    precondition s cmd = case cmd of
            Lock {}     -> isNothing tok
            Guess {}    -> True
            GiveToken _ -> isJust tok
        where
            tok = s ^. CM.contractState . hasToken
```

Now the double-lock test case can no longer be generated. If we save the test case

```
testDoubleLock :: Property
testDoubleLock = flip CM.forAllDL prop_Game $ do
  CM.action $ Lock w1 "*******" 0
  CM.action $ Lock w1 "secret" 0
```

and try to rerun it, then QuickCheck will not do so:

```
> quickCheck testDoubleLock
*** Gave up! Passed only 0 tests; 1000 discarded tests.
```

When a precondition cannot be satisfied, then QuickCheck 'gives up' as we see here–the faulty test case was discarded (1000 times).

Rerunning random tests finds another 'bug':

```
> quickCheck $ propGame' Warning
*** Failed! Assertion failed (after 10 tests and 6 shrinks):
Actions
 [Lock (Wallet 2) "hello" 0,
  Guess (Wallet 1) "hello" "secret" 0]
Outcome of Contract instance for wallet 1:
  False
Failed 'Contract instance stopped with error'
Test failed.
Emulator log:
[WARNING] Slot 3: W1: handleTx failed: InsufficientFunds "Total: Value (Map [(,Map [(,
↪100000000)])]) expected: Value (Map [(f687...,Map [(guess,1)])])"
[WARNING] Slot 3: 00000000-0000-4000-8000-000000000000 {Contract instance for wallet␣
↪1}:
                  Contract instance stopped with error: GameSMError␣
↪(SMCContractError (WalletError (InsufficientFunds "Total: Value (Map [(,Map [(,
↪100000000)])]) expected: Value (Map [(f687...,Map [(guess,1)])])")))
```

In this case, the contract instance in wallet 1 crashes, because the wallet contains 'insufficient funds'. Reading the last line closely, we see that although the wallet contained 100 million Ada, it *lacked* the game *token*, and so making a guess was not allowed.

Arguably, the *off-chain code* should not have tried to submit the guess transaction without holding the game *token*, and the contract instance should not have crashed. Or we might take the view that no harm is done, since the transaction is rejected anyway. But the crashing contract does cause tests to fail, which–as before–is likely to prevent us discovering other problems.

We can strengthen the precondition of `Guess` to prevent this from happening.

```
    precondition s cmd = case cmd of
            Lock {}       -> isNothing tok
            Guess w _ _ _ -> tok == Just w
            GiveToken _   -> isJust tok
        where
            tok = s ^. CM.contractState . hasToken
```

With this change, the tests *still* fail, and we must study the entire log output to understand why:

```
> quickCheck $ prop_Game
*** Failed! Assertion failed (after 36 tests and 35 shrinks):
Actions
 [Lock (Wallet 1) "*******" 1,
  GiveToken (Wallet 2),
  Guess (Wallet 2) "*******" "hello" 2,
  Guess (Wallet 2) "*******" "hunter2" 1]
Expected funds of W2 to change by
  Value (Map [(,Map [(,1)]),(f687...,Map [(guess,1)])])
but they changed by
  Value (Map [(f687...,Map [(guess,1)])])
Test failed.
Emulator log:
... 52 lines of log output ...
[INFO] Slot 4: 00000000-0000-4000-8000-000000000001 {Contract instance for wallet 2}:
                 Receive endpoint call: Object (fromList [("tag",String "guess"),...
↪Number 2.0...
... 25 lines of log output ...
```

(continues on next page)

(continued from previous page)

```
[INFO] Slot 5: TxnValidationFail ab0d...: NegativeValue ...
[INFO] Slot 5: 00000000-0000-4000-8000-000000000001 {Contract instance for wallet 2}:
                 Receive endpoint call: Object (fromList [("tag",String "guess"),...
↪Number 1.0...
```

In this case, we lock one Ada, and then wallet 2 makes two guesses, both with the correct password. The first guess tries to withdraw more Ada than are available, which our model predicts should be a no-op. Recall we defined:

```
    nextState (Guess w old new val) = do
        correctGuess <- (old ==)    <$> CM.viewContractState currentSecret
        holdsToken   <- (Just w ==) <$> CM.viewContractState hasToken
        enoughAda    <- (val <=)    <$> CM.viewContractState gameValue
        when (correctGuess && holdsToken && enoughAda) $ do
            currentSecret .= new
            gameValue     %= subtract val
            CM.deposit w $ Ada.lovelaceValueOf val
        CM.wait 1
```

Our model predicts that the second guess, with the correct password and a withdrawal of only one Ada, ought to succeed. That is why we expected wallet 2 to end up with the game *token*, and one Ada. However, wallet 2 did not receive the Ada, only the game *token*. Reading the emulator log reveals why: in slot 4 we called the `guess` *endpoint* to withdraw two Ada, which would leave -1 Ada locked by the contract, but the transaction submitted to the blockchain was not validated, and we see the error message `NegativeValue`. We made the second *endpoint* call, for the second guess, but nothing more happened. This is because the validation failure *did not crash the off-chain contract instance* (which would have provoked a test failure after the first guess), it just left it waiting for a result from the blockchain. As a result, the contract instance is hanging, and ignores the second guess.

We can avoid this problem too, by strengthening the precondition further:

```
    precondition s cmd = case cmd of
            Lock _ _ v    -> isNothing tok
            Guess w _ _ v -> tok == Just w && v <= val
            GiveToken w   -> isJust tok
        where
            tok = s ^. CM.contractState . hasToken
            val = s ^. CM.contractState . gameValue
```

Now the tests pass:

```
> quickCheck . withMaxSuccess 10000 $ prop_Game
+++ OK, passed 10000 tests.

Actions (241234 in total):
87.1324% GiveToken
 9.0854% Guess
 3.7822% Lock
```

It is good practice to run *far more* than 100 tests, once tests are passing.

In this section we discovered ways to crash the off-line contract instances, or leave them hanging. We debugged the problems by strengthening preconditions–but of course, the problems are still there. We have just avoided provoking them with our tests, which enabled us to continue testing and find more problems. But unless these problems are corrected, enabling our preconditions to be weakened again, then all we know from our tests is that the contract behaves correctly *provided callers obey the preconditions*.

Measuring and tuning distributions

Running successful tests displays statistics over the test cases generated. By default, testing a `Plutus.Contract.Test.ContractModel.Interface.ContractModel` just displays the distribution of types of action. Looking at the output above, we can see that the vast majority of actions were `GiveToken` actions; only 9% were guesses, and fewer than 4% were `Lock` actions.

It is not a surprise that there were relatively few `Lock` actions: our precondition guarantees that there can be at most one `Lock` per test case, and this is intentional, so of course the other actions are much more common. However, we almost certainly *don't* want to test `GiveToken` almost ten times as often as `Guess`. What is going on?

The problem is this: after a `Lock` as the first action of a test case, *every attempt to generate a* `GiveToken` *action will succeed*; that is, the precondition of the generated action will be `True`. But for `Guess` actions, many randomly generated actions will not satisfy the precondition we ended up with, either because the wallet does not contain the game *token*, or because the amount to be withdrawn is greater than the amount available.

To achieve a better distribution of tests, we need to redefine the action generator so that `Guess` actions more often satisfy their precondition. The action generator is itself parameterized on the contract state, so we could *guarantee* that generated guesses satisfy their preconditions by redefining it as follows:

```
    arbitraryAction s = oneof $
        [ Lock      <$> genWallet <*> genGuess <*> genValue ] ++
        [ Guess w   <$> genGuess  <*> genGuess <*> choose (0, val)
        | Just w <- [tok] ] ++
        [ GiveToken <$> genWallet ]
        where
            tok = s ^. CM.contractState . hasToken
            val = s ^. CM.contractState . gameValue
```

With this change, `Guess` and `GiveToken` actions become equally frequent:

```
> quickCheck . withMaxSuccess 1000 $ prop_Game
+++ OK, passed 1000 tests.

Actions (23917 in total):
48.271% GiveToken
47.845% Guess
 3.884% Lock
```

Custom generators vs preconditions

It may seem like wasted effort to encode the form of valid `Guess` actions twice, once in the precondition, and then again in the generator. Would it not be sufficient to write the generator to target successful guesses in the first place, and omit the precondition?

The answer is **no**: it would not. By writing the generator carefully, we can ensure that the *generated* `Guess` actions are valid, but as soon as a test fails, and QuickCheck begins to shrink it, then the precondition becomes essential. Without it, QuickCheck might remove a `GiveToken` action that makes a subsequent `Guess` valid, and then report that the resulting test (not surprisingly) failed. It is only preconditions that ensure that *shrunk* test cases make sense.

Thus, the action generator cannot *ensure* that actions in test cases are valid; it can only skew the *distribution* of actions towards valid ones. This means there is no need for the action generator to guarantee that the actions it generates are valid; they will in any case have to pass the precondition before they are included in a test case. In fact, it is a little dangerous to define a generator so that *only* actions satisfying the precondition are generated, because we might later choose to weaken the precondition. If we do so, and forget to change the generator too, then we might end up with less thorough testing than we expect. So rather than generate guesses as we did above, it would be better to define

```
    arbitraryAction s = oneof $
        [ Lock      <$> genWallet <*> genGuess <*> genValue ] ++
        [ frequency $
          [ (10, Guess w   <$> genGuess  <*> genGuess <*> choose (0, val))
          | Just w <- [tok] ] ++
          [ (1, Guess <$> genWallet <*> genGuess <*> genGuess <*> genValue) ] ] ++
        [ GiveToken <$> genWallet ]
        where
            tok = s ^. CM.contractState . hasToken
            val = s ^. CM.contractState . gameValue
```

which generates valid guesses *most* of the time, with the occasional possibly-invalid one. This approach results in test cases with a reasonable balance between guessing and passing the game *token*, while ensuring that if the preconditions are later changed, then we can still generate every test case we could before.

Instrumenting contract models to gather statistics

It is possible to gather further statistics about the tests we are generating. For example, we might wonder what proportion of `Guess` actions are correct guesses. We can find out by defining the `Plutus.Contract.Test.ContractModel.Interface.monitoring` method in the `Plutus.Contract.Test.ContractModel.Interface.ContractModel` class:

```
 monitoring :: (CM.ModelState state, CM.ModelState state) -> CM.Action state ->␣
↪Property -> Property
```

This function is called for every `Plutus.Contract.Test.ContractModel.Interface.Action` in a test case, and given the `Plutus.Contract.Test.ContractModel.Interface.ModelState` before and after the `Plutus.Contract.Test.ContractModel.Interface.Action`. Its result is a function that is applied to the property being tested, so it can use any of the QuickCheck functions for analysing test case distribution or adding output to counterexamples.

To create a table showing the proportion of guesses which were right or wrong, we can define `Plutus.Contract.Test.ContractModel.Interface.monitoring` as

```
    monitoring (s, _) (Guess w guess new v) =
        tabulate "Guesses" [if guess == secret then "Right" else "Wrong"]
        where secret = s ^. CM.contractState . currentSecret
    monitoring _ _ = id
```

This generates output such as this:

```
> quickCheck . withMaxSuccess 1000 $ prop_Game
+++ OK, passed 1000 tests.

Actions (23917 in total):
48.271% GiveToken
47.845% Guess
 3.884% Lock

Guesses (11443 in total):
75.417% Wrong
24.583% Right
```

Around 25% of guesses were correct in this test run, which is not surprising since we chose guesses uniformly from a list of four possibilities (and the `Plutus.Contract.Test.ContractModel.Interface.precondition` for guesses does not depend on the choice). Since correct guesses are probably at least as interesting to test as incorrect

ones, a sensible next step would be to modify the guess generator to guess correctly more often–perhaps half the time. We leave this as an exercise for the reader.

It is always good practice to make measurements of the distribution of test cases like this, and then improve test case generation to that the distribution looks reasonable. Otherwise there is a risk of developing a false sense of security, engendered by running many thousands of trivial tests.

Goal-directed testing with dynamic logic

The tests we have developed so far test that *'nothing bad ever happens'*-the funds in a test always end up where the model says that they should. To put it another way, funds are never stolen. But this does not really cover everything we want to test: we also want to know that *'something good eventually happens'*, or at least, *'something good is always possible'*. Concretely, this will often mean testing that the funds in a contract can always be recovered–they cannot end up locked in a contract for ever. And indeed, in the case of the game contract, we would like to check that no matter what has happened previously, the Ada locked by the contract can always be recovered by a player who knows the password.

Here we are really identifying desirable 'goal states', namely those in which all the Ada have been recovered from the contract, and aiming to test that a goal state is always reachable. Obviously, random tests are quite unlikely to end in a goal state, so no particular conclusion can be drawn from one that does not. It is also hard to see how QuickCheck might determine automatically whether a goal state is reachable or not. So we test this kind of property by allowing the tester to *specify a strategy* for reaching a goal state; QuickCheck then tests that this strategy always works.

Introducing the dynamic logic monad

We write this kind of test using 'dynamic logic' wrapped in a monad, which just means that we write test case generators that can mix random actions, specified actions, and assertions. These generators are little programs in the `Plutus.Contract.Test.ContractModel.Interface.DL` monad, such as this one:

```
unitTest :: CM.DL GameModel ()
unitTest = do
    CM.action $ Lock w1 "hello" 10
    CM.action $ GiveToken w2
    CM.action $ Guess w2 "hello" "new secret" 3
```

This `Plutus.Contract.Test.ContractModel.Interface.DL` fragment simply specifies a unit test in terms of the underlying `Plutus.Contract.Test.ContractModel.Interface.ContractModel` we have already seen, using `Plutus.Contract.Test.ContractModel.Interface.action` to include a specific `Plutus.Contract.Test.ContractModel.Interface.Action` in the test. To run such a test, we must specify a QuickCheck property such as

```
propDL :: CM.DL GameModel () -> Property
propDL dl = CM.forAllDL dl prop_Game
```

which uses `Plutus.Contract.Test.ContractModel.Interface.forAllDL` to generate a test sequence from the `dl` provided, and runs it using the same underlying property as before. The execution is checked against the model, so *we do not need to add any further assertions* to this unit test. This gives us a very convenient way to define unit tests for a contract specified by a `Plutus.Contract.Test.ContractModel.Interface.ContractModel`.

We can run this test as follows:

```
> quickCheck . withMaxSuccess 1 $ propDL unitTest
+++ OK, passed 1 test.
```

(continues on next page)

(continued from previous page)

```
Actions (3 in total):
33% GiveToken
33% Guess
33% Lock
```

Quantifiers in dynamic logic

As well as writing unit tests in the `Plutus.Contract.Test.ContractModel.Interface.DL` monad, we can add random generation. For example, if we wanted to generalize the unit test above a little to lock a random amount of Ada in the contract, then we could instead write:

```
unitTest :: CM.DL GameModel ()
unitTest = do
    val <- CM.forAllQ $ CM.chooseQ (1, 20)
    CM.action $ Lock w1 "hello" val
    CM.action $ GiveToken w2
    CM.action $ Guess w2 "hello" "new secret" 3
```

Here `Plutus.Contract.Test.ContractModel.Interface.forAllQ` lets us generate a random value using *Test.QuickCheck.DynamicLogic.Quantify.chooseQ* from *quickcheck-dynamic*:

```
chooseQ :: (Arbitrary a, Random a, Ord a) => (a, a) -> Quantification a
```

`Plutus.Contract.Test.ContractModel.Interface.forAllQ` takes a *Test.QuickCheck.DynamicLogic.Quantify.Quantification*, which resembles a QuickCheck generator, but with a more limited API to support its use in dynamic logic.

When this is tested, random values in the range 1-20 are locked... and a test fails:

```
> quickCheck $ propDL unitTest
*** Failed! Falsified (after 3 tests):
BadPrecondition
  [Witness (1 :: Integer),
   Do $ Lock (Wallet 1) "hello" 1,
   Do $ GiveToken (Wallet 2)]
  [Action (Guess (Wallet 2) "hello" "new secret" 3)]
  (GameModel {_gameValue = 1, _hasToken = Just (Wallet 2), _currentSecret = "hello"})
```

Dynamic logic test cases are a little more complex than the simple action sequences we have seen so far, and they give us a little more information. Every such test contains a list of `Plutus.Contract.Test.ContractModel.Interface.Action`, tagged `Do`, and *witnesses*, tagged `Witness`. The witnesses record the results of random choices made by `Plutus.Contract.Test.ContractModel.Interface.forAllQ`: in this case, the Ada value to be locked was chosen to be 1. The test proceeds by locking the Ada and giving the game *token* to wallet 2, but the third action we specified–making the guess–cannot be run, because its precondition is `False`. This is what the `BadPrecondition` tells us, and the action that could not be performed appears as

```
[Action (Guess (Wallet 2) "hello" "new secret" 3)]
```

The last component is the model state at that point: we can see that the `gameValue` is only 1 Ada, so of course we cannot withdraw 3.

Note: We saw earlier that when tests are *generated* from a `Plutus.Contract.Test.ContractModel.Interface.ContractModel`, then QuickCheck only generates actions whose

`Plutus.Contract.Test.ContractModel.Interface.precondition` is satisfied. On the other hand, when we use dynamic logic to specify an action explicitly like this, then there is no guarantee that its precondition will hold, and so a 'bad precondition' error becomes a possibility. The problem here is really that this generalized unit test is inconsistent with our model.

Repeating a dynamic logic test

Once again, we can copy-and-paste the failed testcase into our source code:

```
badUnitTest :: CM.DL GameModel ()
badUnitTest = do
  CM.action $ Lock w1 "hello" 1
  CM.action $ GiveToken w2
  CM.action $ Guess w2 "hello" "new secret" 3
```

We can rerun the test by supplying the precise action sequence that was generated:

```
> quickCheck $ forAllDL badUnitTest prop_Game
*** Failed! Falsified (after 1 test):
```

If we now correct `unitTest` and do freshly generated random tests we see that the issue is resolved:

```
> quickCheck $ forAllDL unitTest prop_Game
+++ OK, passed 100 tests.

Actions (300 in total):
33.3% GiveToken
33.3% Guess
33.3% Lock
```

In this case the saved test 'passes' because it no longer matches the modified `Plutus.Contract.Test.ContractModel.Interface.DL` test, so it is not a counterexample to the property we are testing.

Something good is always possible

We saw above how to generate random *parameters* to actions in dynamic logic tests; what gives them their real power is that we can also include random *actions*.

Suppose we want to test that no Ada remain locked in the game contract for ever. We could try to specify this with a `Plutus.Contract.Test.ContractModel.Interface.DL` test that requires that *no Ada remain locked in the contract after any sequence of actions*. We can include a random sequence of actions in a `Plutus.Contract.Test.ContractModel.Interface.DL` test using `Plutus.Contract.Test.ContractModel.Interface.anyActions_`, and we can make assertions about the `Plutus.Contract.Test.ContractModel.Interface.ModelState` using `Plutus.Contract.Test.ContractModel.Interface.assertModel`. Thus we can define

```
noLockedFunds :: CM.DL GameModel ()
noLockedFunds = do
    CM.anyActions_
    CM.assertModel "Locked funds should be zero" $ CM.symIsZero . CM.lockedValue
```

to assert that, after any sequence of actions, no funds should remain locked (`Plutus.Contract.Test.ContractModel.Interface.lockedValue` extracts the total value locked in contracts from the `Plutus.Contract.Test.ContractModel.Interface.ModelState`).

Of course, this test fails:

```
> quickCheck $ forAllDL noLockedFunds prop_Game
*** Failed! Falsified (after 1 test and 2 shrinks):
BadPrecondition
  [Do $ Lock (Wallet 1) "*******" 1]
  [Assert "Locked funds should be zero"]
  (GameModel {_gameValue = 1, _hasToken = Just (Wallet 1), _currentSecret = "*******"}
↪)
```

If all we do is lock one Ada, then obviously the locked funds are not zero. The failed assertion is reported as a `BadPrecondition` (for the assertion).

The property we wrote above is wrong: what we really intended to say was that *after a correct guess that requests all the funds*, then no locked funds remain. Let us write a property that says that any wallet can recover the funds by making such a guess. To program our strategy, we will need to read the secret password, and the value remaining in the contract, from the contract model:

```
noLockedFunds :: CM.DL GameModel ()
noLockedFunds = do
    CM.anyActions_
    w      <- CM.forAllQ $ CM.elementsQ wallets
    secret <- CM.viewContractState currentSecret
    val    <- CM.viewContractState gameValue
    CM.action $ Guess w "" secret val
    CM.assertModel "Locked funds should be zero" $ CM.symIsZero . CM.lockedValue
```

After a random sequence of actions, we choose a random wallet and construct a correct guess that recovers all the locked Ada to this wallet. But this property also fails!

```
> quickCheck $ forAllDL noLockedFunds prop_Game
*** Failed! Falsified (after 1 test and 2 shrinks):
BadPrecondition
  [Witness (Wallet 1 :: Wallet)]
  [Action (Guess (Wallet 1) "" "" 0)]
  (GameModel {_gameValue = 0, _hasToken = Nothing, _currentSecret = ""})
```

Here QuickCheck has chosen the arbitrary sequence of actions to be *empty*, so the contract has not even been locked–and of course, in that case, a `Guess` is not possible. To pass the test, our strategy must work in *every* situation. However, if the contract has not been locked, then there are no locked funds, so the assertion in this property would pass without our doing anything at all. Perhaps we should only make a `Guess` if there are actually funds to be recovered:

```
noLockedFunds :: CM.DL GameModel ()
noLockedFunds = do
    CM.anyActions_
    w      <- CM.forAllQ $ CM.elementsQ wallets
    secret <- CM.viewContractState currentSecret
    val    <- CM.viewContractState gameValue
    when (val > 0) $ do
        CM.action $ Guess w "" secret val
    CM.assertModel "Locked funds should be zero" $ CM.symIsZero . CM.lockedValue
```

This is better, but testing the property still fails:

```
> quickCheck $ forAllDL noLockedFunds prop_Game
*** Failed! Falsified (after 1 test and 1 shrink):
```

(continues on next page)

(continued from previous page)

```
BadPrecondition
  [Do $ Lock (Wallet 1) "*******" 1,
   Witness (Wallet 2 :: Wallet)]
  [Action (Guess (Wallet 2) "" "*******" 1)]
  (GameModel {_gameValue = 1, _hasToken = Just (Wallet 1), _currentSecret = "*******"}
↪)
```

In this case we locked 1 Ada in the contract, chose wallet 2 to recover the funds, and then tried to make a correct guess–but the precondition for `Guess` still failed. And this is no surprise: the wallet does not hold the game *token*. This test case shows that, as part of our strategy for recovering the funds, we also need to give the game *token* to the wallet that will make the guess.

```
noLockedFunds :: CM.DL GameModel ()
noLockedFunds = do
    CM.anyActions_
    w      <- CM.forAllQ $ CM.elementsQ wallets
    secret <- CM.viewContractState currentSecret
    val    <- CM.viewContractState gameValue
    when (val > 0) $ do
        CM.action $ GiveToken w
        CM.action $ Guess w "" secret val
    CM.assertModel "Locked funds should be zero" $ CM.symIsZero . CM.lockedValue
```

Now we expect the tests to pass:

```
> quickCheck $ forAllDL noLockedFunds prop_Game
*** Failed! Falsified (after 1 test):
BadPrecondition
  [Do $ Lock (Wallet 1) "hello" 5,
   Witness (Wallet 3 :: Wallet),
   Do $ GiveToken (Wallet 3),
   Do $ Guess (Wallet 3) "" "hello" 5]
  [Assert "Locked funds should be zero"]
  (GameModel {_gameValue = 5, _hasToken = Just (Wallet 3), _currentSecret = "hello"})
```

They do not! We can see from the last line that, in the final state, our model indeed says that there are still 5 Ada locked in the contract. This is the effect of the `Plutus.Contract.Test.ContractModel.Interface.nextState` function in our model, so let us inspect the relevant part of its code:

```
    nextState (Guess w old new val) = do
        correctGuess <- (old ==)    <$> CM.viewContractState currentSecret
        -- ...
```

Comparing carefully with the failed test, we see that our strategy is supplying the empty string as the old password (the guess), and the correct password as the new one–so the guess is wrong, and the Ada was not recovered. Swapping the two password arguments to `Guess` does, at last, make the tests pass.

For this simple contract, recovering the locked funds is easy–but as we have seen, writing a property that says that it is always possible forces us to be precise about our strategy, and reveals anything we might have overlooked.

Monitoring and tuning dynamic logic tests

The dynamic logic test we have developed only uses our recovery strategy if there are locked funds remaining after a random sequence of actions. How often does that happen? Given that tests contain many more guesses than `Lock` actions, there is a risk that the contract is usually holding no funds before we even consider using our strategy. To find out, we can `Plutus.Contract.Test.ContractModel.Interface.monitor` the contract model during our tests. As in the `Plutus.Contract.Test.ContractModel.Interface.monitoring` method of the `Plutus.Contract.Test.ContractModel.Interface.ContractModel` class, we can use any of the QuickCheck operations for analyzing test cases, but instead of applying the `Plutus.Contract.Test.ContractModel.Interface.monitoring` at every action in a test case, we can `Plutus.Contract.Test.ContractModel.Interface.monitor` at selected points.

In this case, we choose to label test cases that actually invoke our fund recovery strategy:

```
noLockedFunds :: CM.DL GameModel ()
noLockedFunds = do
    CM.anyActions_
    w      <- CM.forAllQ $ CM.elementsQ wallets
    secret <- CM.viewContractState currentSecret
    val    <- CM.viewContractState gameValue
    when (val > 0) $ do
        CM.monitor $ label "Unlocking funds"
        CM.action $ GiveToken w
        CM.action $ Guess w secret "" val
    CM.assertModel "Locked funds should be zero" $ CM.symIsZero . CM.lockedValue
```

With the addition of the `Plutus.Contract.Test.ContractModel.Interface.monitor` line, QuickCheck tells us what proportion of our tests actually leave funds to recover:

```
> quickCheck $ forAllDL noLockedFunds prop_Game
+++ OK, passed 100 tests (31% Unlocking funds).

Actions (5112 in total):
49.24% GiveToken
48.81% Guess
 1.96% Lock
```

We can see that around 30% of generated tests leave some Ada in the contract for our strategy to recover. This is a bit low–it means that two thirds of our tests do not actually test the strategy. But it is easy to address: we can simply use the dynamic logic to specify the initial `Lock` action *explicitly*, and generate larger amounts for the initial funds locked in the game (lines 3-5 below):

```
noLockedFunds :: CM.DL GameModel ()
noLockedFunds = do
    (w0, funds, pass) <- CM.forAllQ (CM.elementsQ wallets, CM.chooseQ (1, 10000), CM.
↪elementsQ guesses)
    CM.action $ Lock w0 pass funds
    CM.anyActions_
    w      <- CM.forAllQ $ CM.elementsQ wallets
    secret <- CM.viewContractState currentSecret
    val    <- CM.viewContractState gameValue
    when (val > 0) $ do
        CM.monitor $ label "Unlocking funds"
        CM.action $ GiveToken w
        CM.action $ Guess w secret "" val
    CM.assertModel "Locked funds should be zero" $ CM.symIsZero . CM.lockedValue
```

With this addition, a much higher proportion of tests actually exercise our recovery strategy:

```
> quickCheck $ forAllDL noLockedFunds prop_Game
+++ OK, passed 100 tests (74% Unlocking funds).

Actions (5198 in total):
49.75% GiveToken
48.33% Guess
 1.92% Lock
```

More dynamic logic

Dynamic logic tests are much more expressive than we have seen hitherto. The `Plutus.Contract.Test.ContractModel.Interface.DL` monad is an instance of `Alternative`, so we can write tests with random control flow, weight choices suitably, and so on. For example, `Plutus.Contract.Test.ContractModel.Interface.anyActions`, which generates a random sequence of actions of expected length `n`, is defined by

```
anyActions :: Int -> CM.DL s ()
anyActions n = CM.stopping
           <|> CM.weight (1 / fromIntegral n)
           <|> (CM.anyAction >> anyActions n)
```

This code makes a random choice between three alternatives, expressed using `(<|>)`. The first two alternatives terminate (and return `()`), while the last alternative performs a random action followed by another random sequence of actions. The second alternative is weighted by `1/n`, so the third is chosen `n` times as often, resulting in an expected length of `n` actions. The first alternative is guarded by `Plutus.Contract.Test.ContractModel.Interface.stopping`, which means it will be chosen *only* if the test case is 'getting too long'; in this case `Plutus.Contract.Test.ContractModel.Interface.anyActions` will generate an empty sequence. We can exercise fine control over the way test cases are generated, including specifying strategies for bringing a long test case to a close. See the documentation for more details.

Limitations

The tests we have developed here suffer from three main limitations:

- We test only via the off-chain *contract endpoints*
- We test under favourable assumptions on timing
- We don't test for information leaks

What are the implications, and how can we address them?

Testing only via contract endpoints

We tested that the guessing game contract behaves as it should, *provided transactions are submitted using the off-chain contract code*. But what if a malicious actor writes their *own* off-chain code, submitting transactions that the contract's own *off-chain code* cannot generate? Is it possible, for example, to steal the Ada locked by the game, *without* submitting a guess? Our tests do not cover this.

One way to mitigate this problem is to add additional 'attack' *endpoints* to the contract under test, that carry out a variety of conceivable attacks. Our contract model would then model these attack actions as no-ops, to represent the fact that the attacks fail; our tests would then check that the attacks fail in all circumstances, and with all parameters.

If we think of the guessing game, not as a game, but instead as a contract that protects funds with a password, then we might consider 'guessing' with the correct password as a correct withdrawal, and guessing with an incorrect password

as an attack. Our model does test that these attacks always fail; this approach can be used in general to test that contracts are robust against *anticipated* attacks.

Test assumptions on timing

Our tests wait for the transactions generated by an `Plutus.Contract.Test.ContractModel.Interface.Action` to complete before performing the next `Plutus.Contract.Test.ContractModel.Interface.Action` (because we inserted calls to `delay` into `Plutus.Contract.Test.ContractModel.Interface.perform`, and `Plutus.Contract.Test.ContractModel.Interface.wait` into `Plutus.Contract.Test.ContractModel.Interface.nextState`). In reality, different wallets may perform actions simultaneously. This usually results in two or more transactions that try to spend the same *UTXO*, leading all but the first to fail.

Endpoint calls that submit several transactions are even more problematic, because such a call may fail part way through, leaving the blockchain in an intermediate, and possibly invalid state. Arguably, contracts should be written to undo the effects of earlier transactions if later ones fail–although, as we saw, the `Lock` *endpoint* of the game contract (which is implemented as two transactions) does not do this: a second call to `Lock` fails after the first transaction, and leaves the blockchain in a state that the contract cannot handle.

It is possible to test this kind of behaviour in our framework, by *not* delaying before the next action, so that several actions can be started in the same slot. Delays must then be included as an explicit `Plutus.Contract.Test.ContractModel.Interface.Action` in test cases instead. However, modelling becomes considerably harder, because the model must predict *which* transaction of several simultaneous ones succeeds, and must take into account how many transactions each end-point call results in, and which slot each is expected to be commited in. It isn't clear that the extra modelling effort is really worthwhile.

Moreover, in reality transactions might be delayed, which means that *endpoint* calls that generate several transactions might end up being interleaved in unexpected ways. The emulator doesn't currently simulate this, and so these kinds of tests cannot yet be run.

Testing for information leaks

We have tested that only a guess containing the correct secret can withdraw Ada from the game contract. So to steal the money, an adversary must discover the secret. But recall that an adversary can access everything on the blockchain, and also the contents of transactions.

There are least two serious bugs that the game contract could contain, that would permit an adversary to steal all the money:

- The secret could be stored in plain text in the contract state, instead of encrypted. The contract state is stored on the blockchain. The adversary could simply read the secret from the blockchain, and then make a correct guess to steal the money.
- The secret might be stored in encrypted form on the blockchain, but `Guess` transactions might contain an *encrypted* guess, rather than a plain text guess (as they do in this implementation). Then the adversary could simply read the encrypted secret from the blockchain, and submit it in a guess transaction to steal the money.

Neither of these bugs would be detected by our tests, nor is it clear how they could be.

This test framework is a powerful tool for testing that contracts behave correctly, when used as intended–but users should be aware of the limitations in this section, and be careful to avoid the pitfalls they expose.

Further Examples

In addition to the model for the Game contract presented in this tutorial, there are also models for the Prism and the Auction example contracts in the `plutus-use-cases` project.

5.2.4 Testing Plutus Contracts with Contract Models

Introduction

In this tutorial we will see how to test Plutus contracts with *contract models*, using the framework provided by `Plutus.Contract.Test.ContractModel` This framework generates and runs tests on the Plutus emulator, where each test may involve a number of emulated wallets, each running a collection of Plutus contracts, all submitting transactions to an emulated blockchain. Once the user has defined a suitable model, then QuickCheck can generate and run many thousands of scenarios, taking the application through a wide variety of states, and checking that it behaves correctly in each one. Once the underlying contract model is in place, then the framework can check user-defined properties specific to the application, generic properties such as that no funds remain locked in contracts for ever, and indeed both positive and negative tests—where positive tests check that the contracts allow the intended usages, and negative tests check that they do *not* allow the unintended ones.

The *ContractModel* framework is quite rich in features, but we will introduce them gradually and explain how they can best be used.

Basic Contract Models

Example: A Simple Escrow Contract

We begin by showing how to construct a model for a simplified escrow contract, which can be found in `Plutus.Contracts.Tutorial.Escrow`. This contract enables a group of wallets to make a predetermined exchange of tokens, for example selling an NFT for Ada. There are two endpoints, a `Plutus.Contracts.Tutorial.Escrow.pay` endpoint, and a `Plutus.Contracts.Tutorial.Escrow.redeem` endpoint. Each wallet pays in its contribution to the contract using the `Plutus.Contracts.Tutorial.Escrow.pay` endpoint, and once all the wallets have done so, then any wallet can trigger the predetermined payout using the `Plutus.Contracts.Tutorial.Escrow.redeem` endpoint.

For simplicity, we will begin by testing the contract for a fixed set of predetermined payouts. These are defined by the `Plutus.Contracts.Tutorial.Escrow.EscrowParams`, a type exported by the escrow contract, and which is actually passed to the on-chain validators. `Plutus.Contract.Test` provides ten emulated wallets for use in tests, `Plutus.Contract.Test.w1` to `Plutus.Contract.Test.w10`; in this case we will use five of them:

```
testWallets :: [Wallet]
testWallets = [w1, w2, w3, w4, w5]
```

Let us decide arbitrarily that `Plutus.Contract.Test.w1` will receive a payout of 10 Ada, and `Plutus.Contract.Test.w2` will receive a payout of 20, and define an `Plutus.Contracts.Tutorial.Escrow.EscrowParams` value to represent that:

```
escrowParams :: EscrowParams d
escrowParams =
  EscrowParams
    { escrowTargets  =
        [ payToPaymentPubKeyTarget (mockWalletPaymentPubKeyHash w1) (Ada.adaValueOf␣
↪10)
```

(continues on next page)

(continued from previous page)

```
        , payToPaymentPubKeyTarget (mockWalletPaymentPubKeyHash w2) (Ada.adaValueOf␣
↪20)
        ]
    }
```

The Contract Model Type

In order to generate sensible tests, and to decide how they should behave, we need to track the expected state of the system. The first step in defining a contract model is to define a type to represent this expected state. We usually need to refine it as the model evolves, but for now we keep things simple.

In this case, as wallets make payments into the escrow, we will need to keep track of how much each wallet has paid in. So let us define an `EscrowModel` type that records these contributions. Once the contributions reach the targets, then the escrow may be redeemed, so let us keep track of these targets in the model too. We define

```
data EscrowModel = EscrowModel { _contributions :: Map Wallet Value.Value
                               , _targets       :: Map Wallet Value.Value
                               } deriving (Eq, Show, CM.Generic)

makeLenses ''EscrowModel
```

Note that we use lenses to access the fields of the model. This is why the field names begin with an underscore; the `makeLenses` call creates lenses called just `contributions` and `targets` for these fields, which we will use to access and modify the fields below.

We turn this type into a contract model by making it an instance of the `Plutus.Contract.Test.ContractModel.Interface.ContractModel` class:

```
instance ContractModel EscrowModel where ...
```

The rest of the contract model is provided by defining the methods and associated data types of this class.

What contracts shall we test?

In general, a contract model can be used to test any number of contracts, of differing types, running in any of the emulated wallets. But we need to tell the framework *which* contracts we are going to test, and we need a way for the model to refer to each *contract instance*, so that we can invoke the right endpoints. We do so using a `Plutus.Contract.Test.ContractModel.Interface.ContractInstanceKey`, but since different models will be testing different collections of contracts, then this type is not *fixed*, it is defined as part of each model, as an associated type of the `Plutus.Contract.Test.ContractModel.Interface.ContractModel` class.

In this case only one kind of contract is involved in the tests, the escrow contract, but there will be many instances of it, one running in each wallet. To identify a contract instance, a `Plutus.Contract.Test.ContractModel.Interface.ContractInstanceKey` just has to record the wallet it is running in, we only need one constructor in the type. In general there will be one constructor for each type of contract instance in the test.

```
  data ContractInstanceKey EscrowModel w s e params where
    WalletKey :: Wallet -> CM.ContractInstanceKey EscrowModel () EscrowSchema␣
↪EscrowError ()
```

Note that the `Plutus.Contract.Test.ContractModel.Interface.ContractInstanceKey` type is a GADT, so it tracks not only the model type it belongs to, but also the type of the contract instance it refers to.

The framework also needs to be able to show and compare `Plutus.Contract.Test.ContractModel.Interface.ContractInstanceKey`, so you might expect that we would add a deriving clause to this type definition. But a deriving clause is actually not supported here, because the type is a GADT, so instead we have to give separate 'standalone deriving' declarations outside the `Plutus.Contract.Test.ContractModel.Interface.ContractModel` instance:

```
deriving instance Eq (CM.ContractInstanceKey EscrowModel w s e params)
deriving instance Show (CM.ContractInstanceKey EscrowModel w s e params)
```

Defining `Plutus.Contract.Test.ContractModel.Interface.ContractInstanceKey` is only part of the story: we also have to tell the framework how to *interpret* the contract instance keys, in particular

1. which contract instances to start
2. which emulated wallets to run them in
3. which actual contract each contract instance should run.

We do so by defining three methods in the `Plutus.Contract.Test.ContractModel.Interface.ContractModel` class:

```
  initialInstances = [StartContract (WalletKey w) () | w <- testWallets]

  instanceWallet (WalletKey w) = w

  instanceContract _ WalletKey{} _ = testContract
```

The first line above tells the test framework to start a contract instance in each of the test wallets (with contract parameter `()`), the second line tells the framework which wallet each contract key should run in, and the third line tells the framework which contract to run for each key–in this case, the same `testContract` in each wallet. `Spec.Tutorial.Escrow` does not actually export a complete concrete, only contract endpoints, so for the purposes of the test we just define a contract that allows us to invoke those endpoints repeatedly:

```
testContract = selectList [ void $ payEp escrowParams
                          , void $ redeemEp escrowParams
                          ] >> testContract
```

What actions should tests perform?

The type of Actions

The final *type* we need to define as part of a contract model tells the framework what *actions* to include in generated tests. This is defined as another associated datatype of the `Plutus.Contract.Test.ContractModel.Interface.ContractModel` class, and in this case, we will just need actions to invoke the two contract endpoints:

```
  data Action EscrowModel = Pay Wallet Integer
                          | Redeem Wallet -- ^ Refund Wallet
    deriving (Eq, Show, CM.Generic)
```

The framework needs to be able to show and compare `Plutus.Contract.Test.ContractModel.Interface.Action` too, but in this case we *can* just add a `deriving` clause to the definition.

Performing Actions

QuickCheck will generate sequences of `Plutus.Contract.Test.ContractModel.Interface.Action` as tests, but in order to *run* the tests, we need to specify how each action should be performed. This is done by defining the `Plutus.Contract.Test.ContractModel.Interface.perform` method of the `Plutus.Contract.Test.ContractModel.Interface.ContractModel` class, which maps `Plutus.Contract.Test.ContractModel.Interface.Action` to a computation in the emulator. `Plutus.Contract.Test.ContractModel.Interface.perform` takes several parameters besides the `Plutus.Contract.Test.ContractModel.Interface.Action` to perform, but for now we ignore all but the first, whose purpose is to translate a `Plutus.Contract.Test.ContractModel.Interface.ContractInstanceKey`, used in the model, into a `Plutus.Trace.Emulator.Types.ContractHandle`, used to refer to a contract instance in the emulator. The `Plutus.Contract.Test.ContractModel.Interface.perform` method is free to use any `Plutus.Trace.Emulator.EmulatorTrace` operations, but in practice we usually keep it simple, interpreting each `Plutus.Contract.Test.ContractModel.Interface.Action` as a single call to a contract endpoint. This gives QuickCheck maximal control over the interaction between the tests and the contracts. In this case, we just call either the `Plutus.Contracts.Tutorial.Escrow.pay` or the `Plutus.Contracts.Tutorial.Escrow.redeem` endpoint.

```
  perform h _ _ a = case a of
    Pay w v        -> do
      Trace.callEndpoint @"pay-escrow" (h $ WalletKey w) (Ada.adaValueOf $␣
↪fromInteger v)
      CM.delay 1
    Redeem w       -> do
      Trace.callEndpoint @"redeem-escrow" (h $ WalletKey w) ()
      CM.delay 1
```

Notice that we *do* need to allow each `Plutus.Contract.Test.ContractModel.Interface.Action` time to complete, so we include a `Plutus.Contract.Test.ContractModel.Interface.delay` to tell the emulator to move on to the next slot after each endpoint call. Of course we are free *not* to do this, but then tests will submit many endpoint calls per slot, and fail because the endpoints are not ready to perform them. This is not the most interesting kind of test failure, and so we avoid it by delaying an appropriate number of slots after each endpoint call. The number of slots we need to wait varies from contract to contract, so we usually determine these numbers experimentally. Exactly the same problem arises in writing unit tests, of course.

Modelling Actions

Remember that we need to track the real state of the system using the contract model state? We defined a type for this purpose:

```
data EscrowModel = EscrowModel { _contributions :: Map Wallet Value.Value
                               , _targets       :: Map Wallet Value.Value
                               } deriving (Eq, Show, CM.Generic)

makeLenses ''EscrowModel
```

We need to tell the framework what the *effect* of each `Plutus.Contract.Test.ContractModel.Interface.Action` is expected to be, both on wallet contents, and in terms of the model state. We do this by defining the `Plutus.Contract.Test.ContractModel.Interface.nextState` method of the `Plutus.Contract.Test.ContractModel.Interface.ContractModel` class, which just takes an `Plutus.Contract.Test.ContractModel.Interface.Action` as a parameter, and interprets it in the `Plutus.Contract.Test.ContractModel.Interface.Spec` monad, provided by the `Plutus.Contract.Test.ContractModel.Interface.ContractModel` framework.

```
nextState a = case a of
  Pay w v -> do
    withdraw w (Ada.adaValueOf $ fromInteger v)
    contributions %= Map.insertWith (<>) w (Ada.adaValueOf $ fromInteger v)
    wait 1
  Redeem w -> do
    targets <- viewContractState targets
    sequence_ [ deposit w v | (w, v) <- Map.toList targets ]
    contributions .= Map.empty
    wait 1
```

You can see that the `Plutus.Contract.Test.ContractModel.Interface.Spec` monad allows us to withdraw and deposit values in wallets, so that the framework can predict their expected contents, and also to read and update the model state using the lenses generated from its type definition. For a `Pay` action, we withdraw the payment from the wallet, and record the contribution in the model state, using `(%=)` to update the `contributions` field. For a `Redeem` action, we read the targets from the model state (using `Plutus.Contract.Test.ContractModel.Interface.viewContractState` and the lens generated from the type definition), and then make the corresponding payments to the wallets concerned. In both cases we tell the model to `Plutus.Contract.Test.ContractModel.Interface.wait` one slot, corresponding to the `Plutus.Contract.Test.ContractModel.Interface.delay` call in `Plutus.Contract.Test.ContractModel.Interface.perform`; this is necessary to avoid the model and the emulator getting out of sync.

We also have to specify the *initial* model state at the beginning of each test: we just record that no contributions have been made yet, along with the targets we chose for testing with.

```
initialState = EscrowModel { _contributions = Map.empty
                           , _targets       = Map.fromList [ (w1, Ada.adaValueOf 10)
                                                           , (w2, Ada.adaValueOf 20)
                                                           ]
                           }
```

Given these definitions, the framework can predict the expected model state after any sequence of `Plutus.Contract.Test.ContractModel.Interface.Action`.

Generating Actions

The last step, before we can actually run tests, is to tell the framework how to *generate* random actions. We do this by defining the `Plutus.Contract.Test.ContractModel.Interface.arbitraryAction` method, which is just a QuickCheck generator for the `Plutus.Contract.Test.ContractModel.Interface.Action` type. It gets the current model state as a parameter, so we can if need be adapt the generator depending on the state, but for now that is not important: we just choose between making a payment from a random wallet, and invoking `Plutus.Contracts.Tutorial.Escrow.redeem` from a random wallet. Since we expect to need several payments to fund a redemption, we generate `Pay` actions a bit more often than `Redeem` ones.

```
arbitraryAction _ = frequency $ [ (3, Pay <$> elements testWallets <*> choose (1,␣
↪30))
                                , (1, Redeem <$> elements testWallets) ]
```

Strictly speaking the framework now has enough information to generate and run tests, but it is good practice to define *shrinking* every time we define *generation*; we just defined a generator for actions, so we should define a shrinker too. We do so by defining the `Plutus.Contract.Test.ContractModel.Interface.shrinkAction` method, which, like the QuickCheck `shrink` function, just returns a list of smaller `Plutus.Contract.Test.ContractModel.Interface.Action` to try replacing an action by when a test fails. It is always worth defining

a shrinker: the small amount of effort required is repaid *very* quickly, since failed tests become much easier to understand.

In this case, as in most others, we can just reuse the existing shrinking for those parts of an `Plutus.Contract.Test.ContractModel.Interface.Action` that make sense to shrink. There is no sensible way to shrink a wallet, really, so we just shrink the amount in a payment.

```
  shrinkAction _ (Pay w n) = [Pay w n' | n' <- shrink n]
  shrinkAction _ _         = []
```

With this definition, failing test cases will be reported with the *minimum* payment value that causes a failure.

Running tests and debugging the model

We are finally ready to run some tests! We do still need to define a *property* that we can call QuickCheck with, but the `Plutus.Contract.Test.ContractModel.Interface.ContractModel` framework provides a standard one that we can just reuse. So we define

```
prop_Escrow :: CM.Actions EscrowModel -> Property
prop_Escrow = CM.propRunActions_
```

The important information here is in the type signature, which tells QuickCheck *which* contract model we want to generate and run tests for.

A failing test

Once the property is defined, we are ready to test–and a test fails immediately! This is not unexpected–it is quite rare that a model and implementation match on the first try, so we should expect a little debugging–*of the model*–before we start to find interesting bugs in contracts. When models are written *after* the implementation, as in this case, then the new code–the model code–is likely to be where bugs appear first.

Looking at the test output, the first thing QuickCheck reports is the failed test case:

```
Prelude Spec.Tutorial.Escrow Test.QuickCheck Main> quickCheck prop_Escrow
*** Failed! Assertion failed (after 7 tests and 2 shrinks):
Actions
 [Redeem (Wallet 5)]
```

Here we see what generated tests looks like: they are essentially lists of `Plutus.Contract.Test.ContractModel.Interface.Action`, performed in sequence. In this case there is only one `Plutus.Contract.Test.ContractModel.Interface.Action`: wallet 5 just attempted to redeem the funds in the contract.

The next lines of output tell us why the test failed:

```
Expected funds of W[2] to change by
  Value (Map [(,Map [("",20000000)])])
but they did not change
Expected funds of W[1] to change by
  Value (Map [(,Map [("",10000000)])])
but they did not change
```

Remember we defined the expected payout to be 10 Ada to `Plutus.Contract.Test.w1`, and 20 Ada to `Plutus.Contract.Test.w2`. Our model says (in `Plutus.Contract.Test.ContractModel.Interface.nextState`) that when we perform a `Redeem` then the payout should be made (in Lovelace, not

Ada, which is why the numbers are a million times larger than those in the model). But the wallets did not get the money–which is hardly surprising since no payments at all have been made *to* the contract, so there is no money to disburse.

The remaining output displays a log from the failing contract instance, and the emulator log, both containing the line

```
Contract instance stopped with error: RedeemFailed NotEnoughFundsAtAddress
```

This is an error thrown by the off-chain `Plutus.Contracts.Tutorial.Escrow.redeem` endpoint code, which (quite correctly) checks the funds available, and fails since there are not enough.

Positive testing with preconditions

We now have a failing test, that highlights a discrepancy between the model and the implementation–and it is the model that is wrong. The question is how to fix it, and there is a choice to be made. Either we could decide that the `Plutus.Contract.Test.ContractModel.Interface.nextState` function in the model should *check* whether sufficient funds are available, and if they are not, predict that no payments are made. Or perhaps, we should *restrict our tests so they do not attempt to use ``Redeem`` when it should not succeed*.

Both choices are reasonable. The first alternative is usually called *negative testing*–we deliberately test error situations, and make sure that the implementation correctly detects and handles those errors. The second alternative is *positive testing* (or "happy path" testing), when we make sure that the implementation provides the functionality that it should, when the user makes *correct* use of its API.

It is usually a good idea to focus on positive testing first–indeed, good positive testing is a prerequisite for good negative testing, because it enables us to get the system into a wide variety of interesting states (in which to perform negative tests). So we shall return to negative testing later, and focus–in this section–on making positive testing work well.

To do so, we have to *restrict* test cases, so that they do not include `Redeem` actions, when there are insufficient funds in the escrow. We restrict actions by defining the `Plutus.Contract.Test.ContractModel.Interface.precondition` method of the `Plutus.Contract.Test.ContractModel.Interface.ContractModel` class: any `Plutus.Contract.Test.ContractModel.Interface.Action` for which `Plutus.Contract.Test.ContractModel.Interface.precondition` returns `False` will *not* be included in any generated test. The `Plutus.Contract.Test.ContractModel.Interface.precondition` method is also given the current `Plutus.Contract.Test.ContractModel.Interface.ModelState` as a parameter, so that it can decide to accept or reject an `Plutus.Contract.Test.ContractModel.Interface.Action` based on where it appears in a test.

In this case, we want to *allow* a `Redeem` action only if there are sufficient funds in the escrow, so we just need to compare the contributions made so far to the targets:

```
precondition s a = case a of
  Redeem _ -> (s ^. contractState . contributions . to fold)
              `geq`
              (s ^. contractState . targets . to fold)
  _        -> True
```

In this code, `s` is the entire model state maintained by the framework (including wallet contents, slot number etc), but it contains the "contract state", which is the state we have defined ourselves, the `EscrowModel`. The *lens* `contractState . contributions . to fold` extracts the `EscrowModel`, extracts the `contributions` field from it, and then combines all the `Plutus.V1.Ledger.Value.Value` using `fold`. When we apply it to `s` using `(^.)`, we get the total value of all contributions. Likewise, the second lens application computes the combined value of all the targets. If the contributions exceed the targets, then the `Redeem` is allowed; otherwise, it will not be included in the test. Once we define `Plutus.Contract.`

`Test.ContractModel.Interface.precondition`, then it has to be defined for every form of `Plutus.Contract.Test.ContractModel.Interface.Action`, so we just add a default branch that returns `True`.

Note: We can't use `(>=)` to compare `Plutus.V1.Ledger.Value.Value`; there is no `Ord` instance. That is because some `Plutus.V1.Ledger.Value.Value` are incomparable, such as one Ada and one NFT, which would break our expectations about `Ord`. That is why we have to compare them using `Plutus.V1.Ledger.Value.geq` instead.

With this precondition, the failing test we have seen can no longer be generated, and will not appear again in our `quickCheck` runs.

A second infelicity in the model

Adding a precondition for `Redeem` prevents the previous failing test from being generated, but it does not make the tests pass: it just allows QuickCheck to reveal the next problem in the model. Running tests again, we see:

```
Prelude Spec.Tutorial.Escrow Test.QuickCheck Main> quickCheck prop_Escrow
*** Failed! Assertion failed (after 4 tests and 5 shrinks):
Actions
 [Pay (Wallet 2) 0]
```

This time the test just consists of a single `Pay` action, making a payment of zero (!) Ada to the the contract.

Note: It may seem surprising that the test tries to make a zero payment, given that the *generator* we saw above only generates payments in the range 1 to 30 Ada. But remember that the failing test cases we see are not necessarily freshly generated, they may also have been *shrunk*. In this case, the zero is a result of shrinking: the shrinker we saw can certainly shrink payments to zero, and the *precondition* for `Pay` allows that... it's always `True`. And so, a zero payment can appear in tests. If we wanted to prevent this, the correct way would be to tighten the precondition of `Pay`.

The next part of the output explains why the test failed:

```
Expected funds of W[2] to change by
  Value (Map [])
but they changed by
  Value (Map [(,Map [("",-2000000)])])
a discrepancy of
  Value (Map [(,Map [("",-2000000)])])
```

In other words, the model expected that a payment of zero would not affect the funds held by the calling wallet, but in fact, the wallet lost 2 Ada.

Why did this happen? In this case, the emulator log that follows provides an explanation:

```
.
.
.
[INFO] Slot 1: W[2]: Balancing an unbalanced transaction:
       Tx:
         Tx 2dc052b47a1faeacc0f50b99359990302885a34104df0109576597cc490b8a98:
           {inputs:
           collateral inputs:
           outputs:
             - Value (Map [(,Map [("",2000000)])]) addressed to
```

(continues on next page)

(continued from previous page)

```
                ScriptCredential:␣
↪bcf453ff769866e23d14d5104c36ce4da0ff5bcbed23c622f46b94f1 (no staking credential)
           mint: Value (Map [])
           fee: Value (Map [])
           mps:
           signatures:
           validity range: Interval {ivFrom = LowerBound NegInf True, ivTo =␣
↪UpperBound PosInf True}
           data:
             "\128\164\244[V\184\141\DC19\218#\188L<u\236m2\148<\b\DEL%\v\134\EM<\167
↪"}
       Requires signatures:
       Utxo index:
       Validity range:
         (-? , +?)
.
.
.
```

We see the transaction submitted by the contract, and we can see from its outputs that it *is* paying 2 Ada to the script, even though we specified a payment of zero. The reason for this is that the Cardano blockchain requires a *minimum Ada amount* in every transaction output, currently 2 Ada. It is therefore *impossible* to make a payment of zero Ada to the script–and the Plutus libraries avoid this by adding Ada to each output, if necessary, to meet the minimum requirement. It is these 2 Ada that the wallet has lost.

This is not really a bug in the escrow contract: it's a fundamental limitation enforced by the blockchain itself. Therefore we must adapt our model to allow for it. Once again we have a choice: we *could* specify that every `Pay` action costs at least the minimum Ada, even if the `Plutus.Contract.Test.ContractModel.Interface.Action` contains a lower payment, *or* we can restrict `Pay` actions to amounts greater than or equal to the minimum. We choose the latter, because it is simpler to express–we just tighten the precondition for `Pay`:

```
precondition s a = case a of
    Redeem _ -> (s ^. contractState . contributions . to fold)
                `geq`
                (s ^. contractState . targets . to fold)
    Pay _ v  -> Ada.adaValueOf (fromInteger v) `geq` Ada.toValue minAdaTxOutEstimated
```

Now this failure can no longer appear.

Note: It's debatable whether the contract's behaviour is actually buggy or not. We decided to accept it, and exclude payments smaller than 2 Ada from our tests. But of course, a *user* of the contract might attempt to make a payment of, say, 1 Ada–nothing prevents that. Such a user will see their wallet debited with 2 Ada, and may be surprised by that behaviour. Arguably the contract should explicitly *reject* payments below the minimum, rather than silently increase them. So this failing test does expose this issue.

A third infelicity in the model, and a design issue

Now that we have reasonable preconditions for each `Plutus.Contract.Test.ContractModel.Interface.Action` in a test, we can expect to see more interesting failures. And indeed, the tests still fail, but now with a test case that combines payment and redemption:

```
Prelude Spec.Tutorial.Escrow Test.QuickCheck Main> quickCheck prop_Escrow
*** Failed! Assertion failed (after 8 tests and 5 shrinks):
Actions
 [Pay (Wallet 4) 11,
  Pay (Wallet 5) 20,
  Redeem (Wallet 5)]
Expected funds of W[5] to change by
  Value (Map [(,Map [("",-20000000)])])
but they changed by
  Value (Map [(,Map [("",-19000000)])])
a discrepancy of
  Value (Map [(,Map [("",1000000)])])
```

Here we made two payments, totalling 31 Ada, *which is exactly one Ada more than the combined targets* (recall our targets are 10 Ada to `Plutus.Contract.Test.w1` and 20 Ada to `Plutus.Contract.Test.w2`). Then `Plutus.Contract.Test.w5` redeemed the escrow, *and ended up with 1 Ada too much* (last line). That extra Ada is, of course, the extra unnecessary Ada that was paid to the script in the previous action.

This raises the question: what *should* happen if an escrow holds more funds than are needed to make the target payments? The designers of this contract decided that any surplus should be paid to the wallet submitting the `Plutus.Contracts.Tutorial.Escrow.redeem` transaction. Since this is part of the intended behaviour of the contract, then our model has to reflect it. We can do so with a small extension to the `Plutus.Contract.Test.ContractModel.Interface.nextState` function in the model:

```
  nextState a = case a of
    Pay w v -> ...
    Redeem w -> do
      targets <- viewContractState targets
      sequence_ [ deposit w v | (w, v) <- Map.toList targets ]
      contribs <- viewContractState contributions                    -- NEW
      let leftoverValue = fold contribs <> inv (fold targets)        -- NEW
      deposit w leftoverValue                                        -- NEW
      contributions .= Map.empty
      wait 1
```

The extra code just computes the total contributions and the surplus, and deposits the surplus in the calling wallet.

Now, at last, the tests pass!

```
Prelude Spec.Tutorial.Escrow Test.QuickCheck Main> quickCheck prop_Escrow
+++ OK, passed 100 tests.
```

By default, quickCheck runs 100 tests, which is enough to reveal easily-caught bugs such as those we have seen, but far too few to catch really subtle issues. So at this point, it's wise to run many more tests–the number is limited only by your patience and the speed of the emulator:

```
Prelude Spec.Tutorial.Escrow Test.QuickCheck Main> quickCheck . withMaxSuccess 1000 $␣
↪prop_Escrow
+++ OK, passed 1000 tests.
```

Analysing the distribution of tests

Once tests are passing, then the framework displays statistics collected from the running tests. These statistics give us important information about the *effectiveness* of our tests; a risk with any automated test case generation is that, since we do not usually inspect the running tests, we may not notice if almost all of them are trivial.

The contract model framework gathers some basic statistics by default, and can be configured to gather more, but for now we just consider the built-in ones. After each successful test run, we see a number of tables, starting with a distribution of the actions performed by tests:

```
Prelude Spec.Tutorial.Escrow Test.QuickCheck Main> quickCheck . withMaxSuccess 1000 $␣
↪prop_Escrow
+++ OK, passed 1000 tests.

Actions (25363 in total):
75.894% Pay
12.771% Redeem
11.335% WaitUntil
```

Here we ran 1,000 tests, and as we see from the table, around 25,000 actions were generated. So, on average, each test case consisted of around 25 actions.

Of those actions, three quarters were `Pay` actions, and 10-15% were `Redeem`. This is not unreasonable–we decided when we wrote the `Plutus.Contract.Test.ContractModel.Interface.Action` generator to generate more payments than redemptions. The remaining actions are `WaitUntil` actions, inserted by the framework, which simply wait a number of slots to test for timing dependence; we shall return to them later, but can ignore them for now. Thus this distribution looks quite reasonable.

The second table that appears tells us how often a *generated* `Plutus.Contract.Test.ContractModel.Interface.Action` could not be included in a test, because its *precondition* failed.

```
Actions rejected by precondition (360 in total):
87.8% Redeem
12.2% Pay
```

We can see that 360 actions–in addition to the 25,000 that were included in tests–were generated, but *discarded* because their preconditions were not true. This does represent wasted generation effort, although rejecting 360 out of over 25,000 actions is not really a serious problem–especially given that test case generation is so very much faster than the emulator.

Nevertheless, we can see that the vast majority of rejected actions were `Redeem` actions, and this is because a `Redeem` is not allowed until sufficient payments have been made–but our generator produces them anyway.

We can, of course, change this, to generate `Redeem` actions only when redemption is actually possible:

```
  arbitraryAction s = frequency $ (3, Pay <$> elements testWallets <*> choose (1,␣
↪30)) :
                                  [ (1, Redeem <$> elements testWallets)
                                  | (s ^. CM.contractState . contributions . to fold)␣
↪   -- NEW
                                    `Value.geq`                                      -
↪- NEW
                                    (s ^. CM.contractState . targets . to fold)      ␣
↪   -- NEW
                                  ]
```

Measuring the distribution again after this change, we see that only valid `Redeem` actions are now generated; the only discarded actions are `Pay` actions.

```
Prelude Spec.Tutorial.Escrow Test.QuickCheck Main> quickCheck . withMaxSuccess 1000 $␣
↪prop_Escrow
+++ OK, passed 1000 tests.

Actions (25693 in total):
76.717% Pay
13.035% Redeem
10.248% WaitUntil

Actions rejected by precondition (650 in total):
100.0% Pay
```

The main *disadvantage* of making this change is that it limits the tests that *can* be generated, if the precondition of `Redeem` should be changed in the future. In particular, when we move on to negative testing, then we will want to test invalid attempts to redeem the escrow also. Once the generator is changed like this, then relaxing the precondition is no longer enough to introduce invalid calls. For this reason it could be preferable to *keep* the possibility of generation invalid calls alongside the generator for valid calls, but to assign the potentially-invalid generator a much lower weight.

We will discuss the remaining tables in a later section.

Exercises

You can find the final version of the contract model discussed in this section in `Spec.Tutorial.Escrow1`, in the `plutus-apps` repo.

1. Try running the code in `ghci`. You can do so by starting a `nix` shell, and starting `ghci` using

   ```
   cabal repl plutus-use-cases-test
   ```

   Then import QuickCheck and the contract model:

   ```
   import Test.QuickCheck
   import Spec.Tutorial.Escrow1
   ```

   and run tests using

   ```
   quickCheck prop_Escrow
   ```

   The tests should pass, and you should see tables showing the distribution of tested actions, and so on.

2. Try removing the preconditions for `Pay` and `Redeem`, and reinserting them one by one. Run `quickCheck` after each change, and inspect the failing tests that are generated.

3. Try removing the line

   ```
   deposit w leftoverValue
   ```

   from the `Plutus.Contract.Test.ContractModel.Interface.nextState` function, and verify that tests fail as expected.

4. Try removing one of the lines

   ```
   wait 1
   ```

   from the `Plutus.Contract.Test.ContractModel.Interface.nextState` function (so that the model and the implementation get out of sync). What happens when you run tests?

5. This model does generate `Pay` actions that are discarded by the precondition. Adjust the generator so that invalid `Pay` actions are no longer generated, and run `quickCheck` to verify that this is no longer the case.

Parameterising Models and Dynamic Contract Instances

One of the unsatisfactory aspects of the tests developed in the previous section is that they *always* pay 10 Ada to wallet 1, and 20 Ada to wallet 2. What if the contract only works for certain amounts, or what if it only works with exactly two beneficiary wallets? Of course, we would like to *generate* a random set of payment targets for each test. Such a generator is easy to write:

```
arbitraryTargets :: Gen [(Wallet,Integer)]
arbitraryTargets = do
  ws <- sublistOf testWallets
  vs <- infiniteListOf $ choose (1,30)
  return $ zip ws vs
```

but it is a little more intricate to make the model *use* these generated targets.

There are two problems to overcome:

1. The generated targets are an important part of a test case, *so they must be included in the test case* somehow. But a test case is just a list of actions. So where do we put the targets?
2. The running contracts need to know what the targets are–but our model just contains a static list of contract instances in the test (`Plutus.Contract.Test.ContractModel.Interface.initialInstances`). How can we pass the generated targets to each running contract instance?

Solve these two problems, and we can test escrows with arbitrary payout targets. The techniques we learn will be applicable in many other situations.

Adding an initial action, and test case phases

The first problem is quite easy to solve, in principle. The generated payment targets are an important part of a test case, and a test case is just a list of actions, therefore the generated payment targets must be included in one or more of the actions. Quite simply, *any generated data in a contract model test must be part of an action*. In this case, we just decide that every test should begin with an `Init` action, that specifies the targets to be used in this test case. So we must extend the `Plutus.Contract.Test.ContractModel.Interface.Action` type to include `Init`:

```
  data Action EscrowModel = Init [(Wallet, Integer)]     -- NEW!
                          | Redeem Wallet
                          | Pay Wallet Integer
    deriving (Eq, Show, CM.Generic)
```

We must also ensure that `Init` actions *only* appear as the first action of a test case, and that every test case starts with an `Init` action. We restrict the form of test cases using preconditions, so this means that we must refine the `Plutus.Contract.Test.ContractModel.Interface.precondition` function so that the `Init` precondition only holds at the beginning of a test case, and the other operations' preconditions only hold *after* an `Init` has taken place.

However, the `Plutus.Contract.Test.ContractModel.Interface.precondition` method is only given the action and the contract state as parameters, which means in turn that we must be able to tell whether or not we are at the beginning of the test case, just from the model state. So we have to add a field to the model, to keep track of where in a test case we are. In this simple case we could just add a boolean `initialised`, but we will be a little more general and say that a test case is made up of a number of *phases*, in this case just two, `Initial` and `Running`:

```
data EscrowModel = EscrowModel { _contributions :: Map Wallet Value.Value
                               , _targets       :: Map Wallet Value.Value
                               , _phase         :: Phase              -- NEW!
                               } deriving (Eq, Show, CM.Generic)
```

Now we can specify that at the beginning of a test case we are in the `Initial` phase, and there are no targets:

```
  initialState = EscrowModel { _contributions = Map.empty
                             , _targets       = Map.empty
                             , _phase         = Initial
                             }
```

and that when we model the `Init` action, we update both the phase and the targets accordingly:

```
  nextState a = case a of
    Init wns -> do
      phase   .= Running
      targets .= Map.fromList [(w, Ada.adaValueOf (fromInteger n)) | (w,n) <- wns]
    ...
```

We have to specify how to perform an `Init` action also, but in this case it exists only to initialise the model state with generated targets, so performing it need not do anything:

```
  perform h _ _ a = case a of
    Init _          -> do
      return ()
    ...
```

Now we can add a precondition for `Init`, and restrict the other actions to the `Running` phase only:

```
  precondition s a = case a of
    Init _    -> currentPhase == Initial
    Redeem _  -> currentPhase == Running && ...
    Pay _ v   -> currentPhase == Running && ...
    where currentPhase = s ^. contractState . phase
```

It only remains to *generate* `Init` actions, using the generator for targets that we saw above. We can take the phase into account, and generate an `Init` action only at the start of the test case, and other actions only in the `Running` phase.

```
  arbitraryAction s
    | s ^.contractState . phase == Initial
      = Init <$> arbitraryTargets
    | otherwise
      = ...as before...
```

Note: Here we ensure that we always *generate* test cases that begin with `Init`, but this is *not* enough to ensure that every test case we *run* begins with `Init`. Remember that failed tests are always shrunk, and the first thing the shrinker will try is to discard the leading `Init` action (if that still results in a failing test, which it probably will). The only way to prevent shrinking from discarding the leading `Init` is for the *preconditions* to require it to be there. This is why we focussed on writing the preconditions first: they are more important.

As a matter of principle, when we write a generator, we also write a shrinker, which just requires a one-line addition to the `Plutus.Contract.Test.ContractModel.Interface.shrinkAction` function:

```
  shrinkAction _ (Init tgts) = map Init (shrinkList (\(w,n)->(w,) <$> shrink n) tgts)
```

We cannot shrink wallets, which is why we can't simply apply `shrink` to the list of targets, but using the `shrinkList` function from `Test.QuickCheck` we can easily write a shrinker that will discard list elements and shrink the target values.

Dynamic contract instances

At this point we can generate tests that begin by initialising the escrow targets randomly, but we cannot yet run them successfully. If we try, we see failures like this:

```
*** Failed! Assertion failed (after 11 tests and 5 shrinks):
Actions
 [Init [],
  Redeem (Wallet 1)]
Contract instance log failed to validate:
...
Slot 1: 00000000-0000-4000-8000-000000000000 {Wallet W[1]}:
          Contract instance stopped with error: RedeemFailed NotEnoughFundsAtAddress
...
```

Here we started a test with an empty list of targets, and tried to redeem the escrow, but failed because there were 'not enough funds'. Why not? Because *the contracts we are running still expect the fixed targets* that we started with; we have not yet passed our generated targets to the contract instances under test.

Recall the contract we are testing:

```
testContract = selectList [ void $ payEp escrowParams
                          , void $ redeemEp escrowParams
                          ] >> testContract
```

It invokes the contract endpoints with the fixed set of `Plutus.Contracts.Tutorial.Escrow.EscrowParams` we defined earlier. Clearly we need to parameterise the contract on these `Plutus.Contracts.Tutorial.Escrow.EscrowParams` instead:

```
testContract :: EscrowParams Datum -> Contract () EscrowSchema EscrowError ()
testContract params = selectList [ void $ payEp params
                                 , void $ redeemEp params
                                 ] >> testContract params
```

Now the question is: how do we pass this parameter to each `testContract` as we start them?

Recall the way we started contracts in the previous section. We defined the contracts to start at the beginning of a test in the `Plutus.Contract.Test.ContractModel.Interface.initialInstances` method:

```
  initialInstances = [CM.StartContract (WalletKey w) () | w <- testWallets]
```

Each contract is specified by a `Plutus.Contract.Test.ContractModel.Interface.StartContract`, containing not only a contract instance key, but also a *parameter*–in this case `()`, since we did not need to pass any generated values to `testContract`. Now we do need to, so we must replace that `()` with escrow parameters generated from our payment targets. Moreover, we can no longer start the contracts at the beginning of the test–we must see the `Init` action first, so that we know what the generated targets are. To do so, we redefine

```
  initialInstances = []
```

and instead add a definition of the `Plutus.Contract.Test.ContractModel.Interface.startInstances` method:

```
  startInstances _ (Init wns) =
    [CM.StartContract (WalletKey w) (escrowParams wns) | w <- testWallets]
  startInstances _ _ = []
```

where the escrow parameters are now constructed from our generated targets:

```
escrowParams :: [(Wallet, Integer)] -> EscrowParams d
escrowParams tgts =
  EscrowParams
    { escrowTargets  =
        [ payToPaymentPubKeyTarget (mockWalletPaymentPubKeyHash w) (Ada.adaValueOf␣
↪(fromInteger n))
        | (w,n) <- tgts
        ]
    }
```

The effect of this is to start the contracts *just before* the `Init` action; in fact, using this mechanism, we can start contracts dynamically at any point in a test case.

Note: We should be careful to avoid reusing the same contract instance key more than once, though, since this may lead to confusing results.

You may wonder why we don't simply start new contract instances in the `Plutus.Contract.Test.ContractModel.Interface.perform` method instead. The answer is the framework needs to track the running contract instances, and using `Plutus.Contract.Test.ContractModel.Interface.startInstances` makes this explicit.

The `Plutus.Contract.Test.ContractModel.Interface.StartContract` just specifies the `Plutus.Contract.Test.ContractModel.Interface.ContractInstanceKey` to be started; we define the actual contract to start in the `Plutus.Contract.Test.ContractModel.Interface.instanceContract` method, *which receives the contract parameter* from `Plutus.Contract.Test.ContractModel.Interface.StartContract` as its last argument. So we can just define

```
  instanceContract _ WalletKey{} params = testContract params
```

and our work is (almost) done. The last step is just to update the *type* of `WalletKey`, since it includes the type of the parameter that `Plutus.Contract.Test.ContractModel.Interface.StartContract` accepts.

```
  data ContractInstanceKey EscrowModel w s e params where
    WalletKey :: Wallet -> CM.ContractInstanceKey EscrowModel () EscrowSchema␣
↪EscrowError (EscrowParams Datum)
```

Now, at last, our extended model is complete.

Running our extended tests; another design issue

We can now run our new tests, and, as so often happens when the scope of QuickCheck tests is extended, they do not pass. Here is an example of a failure:

```
Actions
 [Init [(Wallet 5,0)],
  Redeem (Wallet 4)]
Expected funds of W[4] to change by
  Value (Map [])
but they changed by
  Value (Map [(,Map [("",-2000000)])])
a discrepancy of
  Value (Map [(,Map [("",-2000000)])])
Expected funds of W[5] to change by
  Value (Map [])
but they changed by
  Value (Map [(,Map [("",2000000)])])
a discrepancy of
  Value (Map [(,Map [("",2000000)])])
Test failed.
```

In this case the generated target just specifies that wallet 5 should receive 0 Ada–a slightly odd target, perhaps, but not obviously invalid. Since the total of all targets is 0 Ada, then the target is already met, and wallet 4 attempts to redeem the escrow. We might expect the effect to be a no-op–and this is what our model predicts–but it is not what happens. Instead, *wallet 4 pays two Ada to wallet 5*!

The reason this happens is the blockchain rule that every transaction output must contain at least 2 Ada. When wallet 4 attempts to redeem the escrow, then the off-chain code attempts to create a transaction with an output paying 0 Ada to wallet 5, but that is increased to 2 Ada to make the transaction valid. Then when the transaction is balanced, the 2 Ada is taken from the submitting wallet.

Is this a bug in the contract? It is certainly an inconsistency with the `Plutus.Contract.Test.ContractModel.Interface.nextState` function in the model, and we could modify that function to reflect the *actual* transfers of Ada that the contract performs. But these transfers were surely not intentional: a more reasonable approach is to say that target payments that are too small to be accepted by the blockchain are invalid; such targets should not be chosen.

We can make our tests pass by tightening the precondition of `Init` so that targets below the minimum are not accepted:

```
  precondition s a = case a of
    Init tgts-> currentPhase == Initial
              && and [Ada.adaValueOf (fromInteger n) `geq` Ada.toValue␣
↪minAdaTxOutEstimated | (w,n) <- tgts]
    ...
```

This demonstrates that the contract works as expected, provided we *don't* specify targets less than the minimum, but nothing prevents a *user* of the contract from specifying such targets–and we know that the contract code will accept them, and deliver surprising results in those cases. Arguably *all* the contract endpoints should check that the specified targets are valid, and raise an error if they are not. This would prevent the *creation* of invalid escrows, rather than generating unexpected behaviour when they are redeemed.

Thus these failing tests *do* suggest a way in which the contract implementation can be improved, even if the failing cases are fairly unlikely in practice.

Note: QuickCheck was able to find this bug because our *generator* for target payments includes invalid values; we chose values in the range 1 to 31, where 1 is invalid (and shrinking reduced the 1 to a 0 in the failing case that was

reported). It is a good thing we did not ensure, from the start, that only valid target values could be generated–had we done so, we would not have discovered this anomalous behaviour.

In general, it is a good idea for generators to produce, at least occasionally, every kind of input that a user can actually supply, even if some of them are invalid (and may be filtered out by preconditions). Doing so enables this kind of strange behaviour to be discovered.

Exercises

1. You will find the code presented here in `Spec.Tutorial.Escrow2`, with the exception of the last precondition we discussed for `Init`. Run the tests using

   ```
   quickCheck prop_Escrow
   ```

   and make sure you understand how they fail.

2. Make your own copy of the code, and add the tighter precondition for `Init`. Verify that the tests then pass.

3. An alternative explanation for the problem might have been that a target of *zero* should not be allowed (and perhaps the contract implementation should interpret a target of zero by not creating a transaction output at all). *Change the precondition* of `Init` so that it only excludes targets of zero, rather than any target below the minimum. Verify that the tests still fail, and make sure you understand the (slightly more complex) failure.

4. There are quite a few steps involved in introducing these dynamically chosen targets. You can practice these steps by taking the code from `Spec.Tutorial.Escrow1`, which uses fixed targets, and following the steps outlined in this tutorial to turn it into a copy of `Spec.Tutorial.Escrow2`.

Testing "No Locked Funds" with Dynamic Logic

So far, we have tested that a contract's actual transfers of tokens are consistent with the model. That is, *nothing goes wrong*–or to put it bluntly, nobody steals your money. This is an example of a *safety property*. But when we use smart contracts, this is not the only kind of property we care about. Very often, we *also* want to be certain that we can eventually reach some kind of *goal* state–an example of a *liveness property*. In particular, it would be bad if tokens were to be trapped in a contract for ever, with no possibility of recovering them. The Cardano model certainly allows this... imagine a UTXO whose verifier always returns `False`... and so it is our responsibility to ensure that contracts do not fall into this trap. Not only does nothing go wrong, but *something good is always possible*. Not only does no-one steal your money, but you can always recover it yourself.

We call these properties "no locked funds" properties, because that is usually what we want to test: that we can always reach a state in which all tokens have been recovered from the contracts under test. Of course, there is no *general* way to recover tokens held by a contract, so we cannot expect QuickCheck to find a way to reach this goal automatically; instead, we *specify a strategy* for recovering funds, and what we test is that the given strategy always works.

Writing and testing properties using Dynamic Logic

We specify this kind of property using *dynamic logic*. This part of the contract testing framework is inspired by dynamic logic for reasoning about programs, but it can be thought of just as a way of writing *test scenarios*, in which we mix random generation, explicit actions, and assertions. We write such scenarios in the `Plutus.Contract.Test.ContractModel.Interface.DL` monad; for example, here is a scenario that first performs a random sequence of actions, then invokes a finishing strategy, and finally asserts that no tokens remain locked in contracts.

```
finishEscrow :: DL EscrowModel ()
finishEscrow = do
    anyActions_
    finishingStrategy w1
    assertModel "Locked funds are not zero" (symIsZero . lockedValue)
```

Here `Plutus.Contract.Test.ContractModel.Interface.assertModel` lets us include an assertion about the contract model state, `Plutus.Contract.Test.ContractModel.Interface.lockedValue` is a function provided by the framework that computes the total value held by contracts, and `Plutus.Contract.Test.ContractModel.Interface.symIsZero` checks that this is zero. The value is returned here as a `Plutus.Contract.Test.ContractModel.Symbolics.SymValue`, but for now it can be thought of just as a normal Plutus `Plutus.V1.Ledger.Value.Value` with an extra type wrapper.

This scenario just tests that the given finishing strategy always succeeds in recovering all tokens from contracts, no matter what actions have been performed beforehand. The finishing strategy itself is written in the same monad. For example, if we think we should use a `Redeem` action to recover the tokens, then we can define

```
finishingStrategy :: Wallet -> DL EscrowModel ()
finishingStrategy w = do
    currentPhase <- viewContractState phase
    when (currentPhase /= Initial) $ do
      action $ Redeem w
```

Of course, since the strategy must work in any state, including the initial one, then we do have to check that the escrow has been initialised before we attempt to `Redeem`.

Note: These test scenarios are very flexible, and can be used for other purposes too. For example, we could write a test scenario that fixes the escrow targets, thus undoing the generalization we made in the previous section:

```
fixedTargets :: DL EscrowModel ()
fixedTargets = do
  action $ Init [(w1,10),(w2,20)]
  anyActions_
```

Note that generated actions are always *appropriate for the current state*, so here `Plutus.Contract.Test.ContractModel.Interface.anyActions_` will pick up generating the test case from the point after the escrow targets are initialised.

We can use dynamic logic to express everything from unit tests to full random generation (by just specifying `Plutus.Contract.Test.ContractModel.Interface.anyActions_` as the scenario). But for now, we focus on testing "no locked funds" properties.

Now, dynamic logic just specifies a *generator* for tests to perform; we still need to specify *how* to perform those tests. Usually, we just reuse the existing property we have already written, which runs the test case on the emulator and performs the usual checks. In this case, we can define

```
prop_FinishEscrow :: Property
prop_FinishEscrow = CM.forAllDL finishEscrow prop_Escrow
```

Then we can run the tests by passing the property to `quickCheck`, as usual:

```
> quickCheck prop_FinishEscrow
*** Failed! Falsified (after 1 test and 3 shrinks):
BadPrecondition
  [Do $ Init [(Wallet 2,2)]]
```

(continues on next page)

(continued from previous page)

```
  [Action (Redeem (Wallet 1))]
  (EscrowModel {_contributions = fromList [],
                _targets = fromList [(Wallet 2,Value (Map [(,Map [("",
↪2000000)])]))],
                _phase = Running})

BadPrecondition
[Do $ Var 0 := Init [(Wallet 2,2)]]
Some (Redeem (Wallet 1))
```

The property fails, which is not surprising: our "finishing strategy" is quite simplistic, and not yet expected to work. But let us inspect the error message. The test failed because of a bad precondition, after running the sequence

```
Init [(Wallet 2,2)]
```

So we set up a target to pay wallet 2 a sum of 2 Ada. Then we tried to apply our finishing strategy, which is just for wallet 1 to issue a `Redeem` request:

```
Redeem (Wallet 1)
```

This wasn't possible, because the precondition of `Redeem` wasn't satisfied. The message also shows us the model state–we have set up the escrow targets successfully, but there are no contributions, and the `Redeem` precondition says that the contributions must cover the targets before `Redeem` is possible. So of course, it doesn't work.

But the counterexample does show us what we need to do to *make* `Redeem` possible: we need to pay in sufficient contributions to cover the targets. So that suggests a refined finishing strategy:

```
finishingStrategy :: Wallet -> DL EscrowModel ()
finishingStrategy w = do
    currentPhase <- viewContractState phase
    when (currentPhase /= Initial) $ do
      currentTargets  <- viewContractState targets
      currentContribs <- viewContractState contributions
      let deficit = fold currentTargets <> inv (fold currentContribs)
      when (deficit `gt` Ada.adaValueOf 0) $
        action $ Pay w $ round $ Ada.getAda $ max minAdaTxOutEstimated $ Ada.
↪fromValue deficit
      action $ Redeem w
```

We read the contributions and targets from the contract state, compute the remaining deficit, and if the deficit is positive, then we make a payment to cover it. After this, a `Redeem` should be successful. And indeed, testing the property passes: this finishing strategy works.

```
>  quickCheck . withMaxSuccess 1000 $ prop_FinishEscrow
+++ OK, passed 1000 tests.

Actions (51925 in total):
73.483% Pay
14.278% Redeem
10.315% WaitUntil
 1.924% Init
```

Digression: revisiting a design decision

In section *A third infelicity in the model, and a design issue* above, we discussed the situation in which contributors pay in *more* to the escrow than is needed to meet the targets. The actual contract allows that, and so do we in our model; as a consequence we had to *specify* where the surplus funds end up on redemption (in the wallet that invokes `Redeem`). But there is another approach we could have taken: we could simply have said that `Redeem` *requires* the contributions and targets to match exactly, by strengthening the precondition:

```
precondition s (Redeem _) =
     currentPhase == Running
  && fold (s ^. contractState . contributions) == fold (s ^. contractState . targets)
```

This does make `prop_Escrow` pass:

```
> quickCheck prop_Escrow
+++ OK, passed 100 tests.

Actions (2845 in total):
82.81% Pay
13.78% WaitUntil
 3.30% Init
 0.11% Redeem

Actions rejected by precondition (870 in total):
88.3% Redeem
10.8% Pay
 0.9% Init
```

But should we be satisfied with this? There are warning signs in the statistics that QuickCheck collects:

1. We have tested `Redeem` an extremely small number of times.
2. A high proportion of generated `Redeem` actions were *discarded* by the precondition.

The explanation for this is that we can now only include `Redeem` in a test case if the previous (random) payments have hit the target *exactly*, and this is very unlikely. Moreover, once we have overshot the target, then further random payments cannot help.

We could add a stronger *precondition* to `Pay`, that forbids payments taking us over the target, and that would result in a better distribution of actions. But it is not a *realistic* solution, because at the end of the day, there is no way to *prevent* someone making a payment to a script on the Cardano blockchain. *Making a payment to a contract does not require the contract's approval*.

So there is a problem here, but when we test `prop_Escrow`, then it is revealed only by careful inspection of the generated statistics–the property does not *fail*.

On the other hand, when we test `prop_FinishEscrow`, then it fails immediately:

```
> quickCheck prop_FinishEscrow
*** Failed! Falsified (after 5 tests and 6 shrinks):
BadPrecondition
  [Do $ Init [],
   Do $ Pay (Wallet 2) 2]
  [Action (Redeem (Wallet 1))]
  (EscrowModel {_contributions = fromList [(Wallet 2,Value (Map [(,Map [("",
↪2000000)])]))],
                _targets = fromList [],
                _phase = Running})
```

(continues on next page)

(continued from previous page)

```
BadPrecondition
[Do $ Var 0 := Init [],Do $ Var 3 := Pay (Wallet 2) 2]
Some (Redeem (Wallet 1))
```

The counterexample sets up an escrow with an empty list of targets (which may seem odd, but is allowed, and tells us that no particular targets are *needed* to make the property fail). Then it makes a payment to the escrow, thus overshooting the targets. Finally, we try to use the given finishing strategy–which just attempts to use `Redeem`, and fails because the strong precondition we wrote does not allow it.

In this case, not only does the given finishing strategy fail, but the bug cannot be fixed: *there is no possible finishing strategy that works*. Once we have overshot the targets, there is no way to return to a state in which `Redeem` is possible! And that is why the contract authors did *not* follow this path: had they done so, then an attacker would be able to 'brick' an escrow contract just by making an unexpected payment to it.

Fair's fair: Unilateral strategies

We saw above how to test that a finishing strategy succeeds in recovering all the tokens. But not all strategies are created equal. For example, suppose you use an escrow contract to buy an NFT. You place your funds in the escrow, but before the seller can place the NFT there, they get a better offer. Now the seller will never place the NFT in the escrow–and neither can the buyer–and so the buyer's funds *will* be locked for ever, even though there is a way (using the NFT) to recover them.

This little story shows that there is a need for each wallet to be able to recover their "fair share" of the funds in the contract, without any other wallet's cooperation. And the contract model framework provides a way of testing this too.

The idea is to provide *two* strategies, one that recovers all the tokens from contracts, and is also interpreted to define each wallet's "fair share", and a second strategy *that can be followed by any single wallet*, and recovers that wallet's tokens. We call this kind of strategy a *unilateral* strategy; it is defined in the `Plutus.Contract.Test.ContractModel.Interface.DL` monad in just the same way as the strategies we saw earlier, but only a single wallet is allowed to perform actions. Indeed, this is why we gave `finishingStrategy` a wallet as a parameter: it defines the unilateral strategy for that wallet. Since the strategy uses `Redeem`, which actually pays out *all* the targets, then we can reuse it as the general strategy too, just by choosing a wallet to perform it (and we chose wallet 1 above).

The framework lets us package the general and unilateral strategies together, into a "no locked funds proof":

```
noLockProof :: NoLockedFundsProof EscrowModel
noLockProof = defaultNLFP
  { nlfpMainStrategy   = finishingStrategy w1
  , nlfpWalletStrategy = finishingStrategy    }
```

Note: There are other components in a `Plutus.Contract.Test.ContractModel.Interface.NoLockedFundsProof`, which we will see later; we can ignore them for now, but we do need to take suitable default values from `Plutus.Contract.Test.ContractModel.Interface.defaultNLFP` in the definition above.

and we can test them together using `Plutus.Contract.Test.ContractModel.Interface.checkNoLockedFundsProof`

```
prop_NoLockedFunds :: Property
prop_NoLockedFunds = CM.checkNoLockedFundsProof noLockProof
```

```
> quickCheck prop_NoLockedFunds
*** Failed! Falsified (after 1 test and 5 shrinks):
DLScript
  [Do $ Init [(Wallet 4,2)]]

Unilateral strategy for Wallet 4 should have gotten it at least
  SymValue {symValMap = fromList [], actualValPart = Value (Map [(,Map [("",
↪2000000)])])}
but it got
  SymValue {symValMap = fromList [], actualValPart = Value (Map [])}
```

The property actually fails, because if all we do is create a target that wallet 4 should receive 2 Ada, then wallet 4's unilateral strategy is unable to recover that–even though, when wallet 1 follows the strategy, then wallet 4 does receive the money.

What happens here is that the *general* strategy, which is just the same strategy followed by wallet 1, *does* pay out to wallet 4, and so we *define* wallet 4's "fair share" to be 2 Ada. But this isn't really right, because since no Ada have been paid into the contract, then there are no tokens to disburse. Indeed, if anything, the "general" strategy is *unfair* to wallet 1, which has to stump up 2 Ada in this situation so that the escrow can be redeemed. So this test failure does reveal a fairness problem, even if the victim is really wallet 1 rather than wallet 4.

We will see how to fix this problem in the next section. In the meantime, to summarize, defining a `Plutus.Contract.Test.ContractModel.Interface.NoLockedFundsProof` requires us

1. to define a general strategy that can recover *all* the tokens from the contracts under test, and moreover implies a *fair share* of the tokens for each wallet *in any state* (for example, a fair share of the profits-so-far of any trading contract),
2. to define a *unilateral strategy* for each wallet, that can recover that wallet's fair share of the tokens from any state, without cooperation from any other wallet.

Fixing the contract: refunds

The fundamental problem with the finishing strategy we have developed so far, is that in order to recover tokens already held by the contract, we may need to pay in even more tokens! This seems a poor design. It would make far more sense, in the event that the contract is not followed to completion, to *refund* contributions to the wallets that made them. And indeed, the actual implementation of the contract supports a `Plutus.Contracts.Tutorial.Escrow.refund` endpoint as well.

To add refunds to our model, we need to add a new action

```
  data Action EscrowModel = Init [(Wallet, Integer)]
                          | Redeem Wallet
                          | Pay Wallet Integer
                          | Refund Wallet            -- NEW!
    deriving (Eq, Show, CM.Generic)
```

and add it to `Plutus.Contract.Test.ContractModel.Interface.nextState`, `Plutus.Contract.Test.ContractModel.Interface.precondition`, `Plutus.Contract.Test.ContractModel.Interface.perform`, and `Plutus.Contract.Test.ContractModel.Interface.arbitraryAction`:

```
  nextState (Refund w) = do
      v <- viewContractState $ contributions . at w . to fold
      contributions %= Map.delete w
      deposit w v
```

(continues on next page)

(continued from previous page)

```
      wait 1

  precondition s (Refund w) =
       currentPhase == Running
    && w `Map.member` (s ^. contractState . contributions)
    where currentPhase = s ^. contractState . phase

  perform h _ _ (Refund w) = do
      Trace.callEndpoint @"refund-escrow" (h $ WalletKey w) ()
      delay 1

  arbitraryAction s
    ...
      = frequency $ ... ++
                    [ (1, Refund <$> elements testWallets) ]
```

(In the `Plutus.Contract.Test.ContractModel.Interface.nextState` clause, the first line uses a more complex lens to extract the contributions, select the value for wallet `w`, if present, and then pass the resulting `Maybe Value` to `fold`, thus returning zero if there was no contribution, and the value itself if there was). We also have to extend the `testContract` to include the refund endpoint:

```
testContract :: EscrowParams Datum -> Contract () EscrowSchema EscrowError ()
testContract params = selectList [ void $ payEp params
                                 , void $ redeemEp params
                                 , void $ refundEp params      -- NEW!
                                 ] >> testContract params
```

With these additions, `prop_Escrow` still passes, but now tests refunds as well:

```
> quickCheck prop_Escrow
+++ OK, passed 100 tests.

Actions (2625 in total):
66.44% Pay
12.46% WaitUntil
 9.64% Redeem
 7.96% Refund
 3.50% Init

Actions rejected by precondition (478 in total):
85.8% Refund
12.6% Pay
 1.7% Init
```

We can see that `Refund` is tested almost as often as `Redeem`, although many refunds are rejected by the precondition (which requires that there actually *is* a contribution to refund). This isn't a big deal, though, because the overall proportion of rejected actions is low (15%), and sufficiently many `Refund` actions are being tested.

The payoff, though, is that we can now define a far better finishing strategy: the general strategy will just refund all the contributions, and the unilateral strategies will claim a refund for the wallet concerned.

```
finishingStrategy :: CM.DL EscrowModel ()
finishingStrategy = do
    contribs <- CM.viewContractState contributions
    CM.monitor (tabulate "Refunded wallets" [show . Map.size $ contribs])
    sequence_ [CM.action $ Refund w | w <- testWallets, w `Map.member` contribs]
```

(continues on next page)

(continued from previous page)

```
walletStrategy :: Wallet -> CM.DL EscrowModel ()
walletStrategy w = do
    contribs <- CM.viewContractState contributions
    when (w `Map.member` contribs) $ CM.action $ Refund w
```

Note: Here we use `Plutus.Contract.Test.ContractModel.Interface.monitor` to gather statistics on the number of wallets receiving refunds during the finishing strategy, just to make sure, for example, that it is not always zero. We place such monitoring in the *general* strategy, not the wallet-specific ones, because the general strategy is invoked exactly once per test, while the wallet-specific ones may be invoked a variable–and unpredictable–number of times. This makes statistics gathered in the wallet-specific strategies harder to interpret.

We put these strategies together into a `Plutus.Contract.Test.ContractModel.Interface.NoLockedFundsProof`:

```
noLockProof :: CM.NoLockedFundsProof EscrowModel
noLockProof = CM.defaultNLFP
  { CM.nlfpMainStrategy   = finishingStrategy
  , CM.nlfpWalletStrategy = walletStrategy    }
```

and run tests:

```
> quickCheck prop_NoLockedFunds
+++ OK, passed 100 tests.

Actions (31076 in total):
65.211% Pay
11.794% WaitUntil
10.117% Redeem
 9.506% Refund
 1.847% Init
 1.525% Unilateral

Refunded wallets (100 in total):
30% 2
23% 1
17% 4
16% 3
13% 0
 1% 5
```

Now the tests pass–each wallet can indeed recover its own fair share of tokens–and moreover we test each action fairly often, and the number of refunded wallets has a reasonable-looking distribution.

Exercises

You will find the code presented in this section in `Spec.Tutorial.Escrow3`.

1. Strengthen the precondition of `Redeem` to require the contributions and targets to match exactly, as discussed in *Digression: revisiting a design decision*. Verify that `prop_Escrow` passes and `prop_FinishEscrow` fails. Now, *add a precondition to* `Pay` to disallow payments that take the contributions over the target.
    1. Test `prop_Escrow`, and make sure it passes; have you achieved a better distribution of actions in tests?
    2. Test `prop_FinishEscrow`; does it pass now?
2. The code provided uses the poor finishing strategy based on `Redeem`. Verify that `prop_NoLockedFunds` fails, and replace the strategy with the better one described above (you will find the code in comments in the file). Verify that `prop_NoLockedFunds` passes now.

    Do not be surprised if testing `prop_NoLockedFunds` is considerably slower than testing `prop_FinishEscrow`. The latter runs the emulator only once per test, while the former must run it repeatedly to test each wallet's unilateral strategy.
3. Sometimes a wallet which is targetted to *receive* funds might do better to complete the contributions and redeem the escrow, rather than refund its own contribution. Implement this idea as a per-wallet strategy, and see whether `prop_NoLockedFunds` still passes. Add a call of `Plutus.Contract.Test.ContractModel.Interface.monitor` to your strategy to gather statistics on how often `Redeem` is used instead of `Refund`.

Taking Time into Account

In the last section we added refunds to our tests; now a client can pay into an escrow, and claim a refund of their contribution freely–but this doesn't really correspond to the intention of an escrow contract. In reality, an escrow contract should have a deadline for payments and redemption, with refunds permitted only after the deadline has passed. In fact, the *real* escrow contract, in `Plutus.Contracts.Escrow`, provides such a deadline: the main difference between this and the simplified contract we have tested so far, `Plutus.Contracts.Tutorial.Escrow`, is that the latter omits the deadline and associated checks.

In this section, we'll switch to testing the real contract, which we can achieve just by changing the import in our model to be the real contract. (As usual, you can find the code presented in this section in `Spec.Tutorial.Escrow4`).

Slots and POSIXTime

Just changing the import leads to a compiler warning: the `Plutus.Contracts.Escrow.EscrowParams` type, which is passed to the contract under test, has a new field `Plutus.Contracts.Escrow.escrowDeadline`, and so far, our code does not initialise it. We will generate the deadlines, so that they vary from test to test, but there is a slight mismatch to overcome first. In a contract model we measure time in *slots*, but the `Plutus.Contracts.Escrow.escrowDeadline` field is not a slot number, it is a `Plutus.V1.Ledger.Time.POSIXTime`. So while we shall generate the deadline as a slot number (for convenience in the model), we must convert it to a `Plutus.V1.Ledger.Time.POSIXTime` before we can pass it to the contract under test.

To do so, we need to know when slot 0 happens in POSIX time, and how long the duration of each slot is. These are defined in a `Cardano.Node.Emulator.TimeSlot.SlotConfig`, a type defined in `Cardano.Node.Emulator.TimeSlot`. In principle the slot configuration might vary, but we will use the default values for testing (by using `def` from `Data.Default` as our configuration. Putting all this together, we can add a deadline to our `Plutus.Contracts.Escrow.EscrowParams` as follows:

```
escrowParams :: Slot -> [(Wallet, Integer)] -> EscrowParams d
escrowParams s tgts =
```

(continues on next page)

(continued from previous page)

```
  EscrowParams
    { escrowTargets  =
        [ payToPaymentPubKeyTarget (mockWalletPaymentPubKeyHash w) (Ada.adaValueOf␣
↪(fromInteger n))
        | (w,n) <- tgts
        ]
    , escrowDeadline = scSlotZeroTime def + POSIXTime (getSlot s * scSlotLength def) ␣
↪   -- NEW!!!
    }
```

Note: If you are familiar with the `POSIXTime` type from `Data.Time.Clock.POSIX`, then beware that *this is not the same type*. That type has a resolution of picoseconds, while Plutus uses its own `Plutus.V1.Ledger.Time.POSIXTime` type with a resolution of milliseconds.

Initialising the deadline

The deadline, like the escrow targets, is fixed for each test, so it makes sense to add the deadline as a new field to the `Init` action–recall that it is the `Init` action that starts the contract instances under test, and so must supply the deadline as part of the `Plutus.Contracts.Escrow.EscrowParams`. So we add the deadline slot to this action

```
  data Action EscrowModel = Init Slot [(Wallet, Integer)]     -- NEW!!!
                          | Redeem Wallet
                          | Pay Wallet Integer
                          | Refund Wallet
    deriving (Eq, Show, CM.Generic)
```

and pass it to the contracts in the `Plutus.Contract.Test.ContractModel.Interface.startInstances` method:

```
  startInstances _ (Init s wns) =
    [CM.StartContract (WalletKey w) (escrowParams s wns) | w <- testWallets]
```

Just as we record the escrow targets in the model state, so we will need to include the deadline as part of the model, so we extend our model type

```
data EscrowModel =
    EscrowModel
        { _contributions :: Map Wallet Value.Value
        , _targets       :: Map Wallet Value.Value
        , _refundSlot    :: Slot              -- NEW!!!
        , _phase         :: Phase
        } deriving (Eq, Show, CM.Generic)
```

and record the deadline in our model state transition:

```
  nextState (Init s wns) = do
      phase    .= Running
      targets .= Map.fromList [(w, Ada.adaValueOf (fromInteger n)) | (w,n) <- wns]
      refundSlot .= s                  -- NEW!!!
```

It just remains to generate deadline slots (we choose positive integers), and shrink them (by reusing integer shrinking):

```
  arbitraryAction s
    | s ^. CM.contractState . phase == Initial
      = Init <$> (Slot . getPositive <$> arbitrary) <*> arbitraryTargets
```

```
  shrinkAction _ (Init s tgts) = map (Init s) (shrinkList (\(w,n)->(w,)<$>shrink n)␣
↪tgts)
                               ++ map (`Init` tgts) (map Slot . shrink . getSlot $ s) ␣
↪     -- NEW!!!
```

Now we are ready to run tests.

Modelling the passage of time

We can now run tests, but they do not pass:

```
> quickCheck prop_Escrow
*** Failed! Assertion failed (after 5 tests and 7 shrinks):
Actions
 [Init (Slot {getSlot = 0}) [],
  Pay (Wallet 1) 2]
Expected funds of W[1] to change by
  Value (Map [(,Map [("",-2000000)])])
but they did not change
Test failed.
Emulator log:
[INFO] Slot 0: TxnValidate␣
↪ee3a44b98e0325e19bc6be1e6f25cdb269301666a3473758296e96cd7ea9a851
[INFO] Slot 1: 00000000-0000-4000-8000-000000000000 {Wallet W[1]}:
                 Contract instance started
[INFO] Slot 1: 00000000-0000-4000-8000-000000000001 {Wallet W[2]}:
                 Contract instance started
...
```

We tried to pay 2 Ada from wallet 1, but the payment did not take effect. Notice that the generated deadline is slot zero, though; in other words, the deadline passed before we started the test. This might seem surprising, since we *generated* the deadline as a positive integer (and zero does not count as positive), but it is the result of shrinking. If we don't want to test a deadline of slot zero, then we must strengthen the precondition of `Init` to prevent it.

Noting that the contract instances do not start until slot one, let us require the deadline slot to be greater than that–at least slot two. When we add this to the precondition then tests still fail, but the shrunk counterexample is different:

```
> quickCheck prop_Escrow
*** Failed! Assertion failed (after 2 tests and 5 shrinks):
Actions
 [Init (Slot {getSlot = 2}) [],
  WaitUntil (Slot {getSlot = 2}),
  Pay (Wallet 3) 2]
Expected funds of W[3] to change by
  Value (Map [(,Map [("",-2000000)])])
but they did not change
Test failed.
```

This test case makes the problem easier to see: it

1. first, initializes the deadline to slot 2

2. then, *waits until* slot 2,
3. and finally, attempts a payment, which does not go through.

The second action, `WaitUntil`, is one we have not seen in counterexamples previously; it only appears when *timing is important* to provoke a failure. In this case it's now clear what the problem is: *the contract does not allow payments after the deadline*. So the next step is to encode this in our model.

Note: `WaitUntil` actions are inserted automatically into test cases by the framework, to explore timing dependence. It is *possible* to control the probability of a `WaitUntil` action, and the distribution of the slots that we wait for, but it is often not *necessary*–the default behaviour is often good enough.

The contract model framework automatically keeps track of the current slot number for us, so we *could* write a precondition for `Pay` that refers explicitly to the slot number. However, all that really matters is *whether or not the deadline has passed*–and probably other parts of the model will depend on this too. So it is simpler to check for this in one place, and then just refer to it elsewhere in the model.

Now we can benefit from our choice earlier to introduce a `phase` field in the model: hitherto it has only distinguished initialization from running the test, but now we can add a new phase: `Refunding`

```
data Phase = Initial | Running | Refunding deriving (Eq, Show, CM.Generic)
```

The idea is that when the deadline passes, we move into the `Refunding` phase, and we can refer to the current phase in preconditions. In fact, our preconditions *already* refer to the phase, so with this change then `Pay` and `Redeem` will be restricted to take place *before* the deadline. All we have to do is to adjust the precondition for `Refund`, which should of course be restricted to *after* the deadline:

```
  precondition s a = case a of
    Init s tgts -> currentPhase == Initial
                && s > 1
                && and [Ada.adaValueOf (fromInteger n) `Value.geq` Ada.toValue␣
↪minAdaTxOutEstimated | (_,n) <- tgts]
    Redeem _    -> currentPhase == Running
                && fold (s ^. CM.contractState . contributions) `Value.geq` fold (s ^.
↪ CM.contractState . targets)
    Pay _ v     -> currentPhase == Running
                && Ada.adaValueOf (fromInteger v) `Value.geq` Ada.toValue␣
↪minAdaTxOutEstimated
    Refund w    -> currentPhase == Refunding            -- NEW!!!
                && w `Map.member` (s ^. CM.contractState . contributions)
    where currentPhase = s ^. CM.contractState . phase
```

One question remains: *where do we change the phase?* Changing the phase changes the model state, but not in response to an `Plutus.Contract.Test.ContractModel.Interface.Action`: it doesn't matter whether or not an action is performed on the deadline, the phase must change anyway. This means that *we cannot change the phase in the* `Plutus.Contract.Test.ContractModel.Interface.nextState` *function*, because this is invoked only when actions are performed. We need to be able to *change the contract state in response to the passage of time*. We can do this by defining the `Plutus.Contract.Test.ContractModel.Interface.nextReactiveState` method of the `Plutus.Contract.Test.ContractModel.Interface.ContractModel` class.

This method is called every time the slot number advances in the model (although not necessarily every slot–slot numbers can jump during a test). In this case all we need to do is compare the new slot number with the deadline, and move to the `Refunding` phase if appropriate:

```
  nextReactiveState slot = do
    deadline <- CM.viewContractState refundSlot
    when (slot >= deadline) $ phase .= Refunding
```

Now `prop_Escrow` passes.

Monitoring and the distribution of tests

Testing `prop_Escrow` generates some interesting statistics:

```
> quickCheck prop_Escrow
+++ OK, passed 100 tests.

Actions (2291 in total):
62.03% WaitUntil
27.37% Pay
 3.71% Redeem
 3.62% Init
 3.27% Refund

Actions rejected by precondition (11626 in total):
70.437% Pay
23.757% Refund
 5.746% Redeem
 0.060% Init
```

In comparison with previous versions of this property, we can see from the first table that there are *many* more `WaitUntil` actions in these tests (previously they were around 10% of the tested actions). Moreover, we can see that many more generated actions were rejected by their precondition: we rejected over 11,000 actions, while generating 2291 that were included in tests. Rejecting so many actions is undesirable: not only does it waste testing time, but there is a risk that the *distribution* of accepted actions is quite different from that of generated actions, which can lead to ineffective testing.

But why do we see this behaviour? It is because *once the deadline has passed, then neither* `Pay` *nor* `Redeem` *is possible*; when we generate these actions, then they will *always* be rejected by their preconditions. Moreover, *after the deadline then we can* `Refund` *each wallet at most once*. Once the deadline has passed, and all the contributions have been refunded, then the preconditions allow no further actions–except `WaitUntil`. And so, test case generation will choose `WaitUntil`, over and over again, and this is why so many of them appear in our tests.

The following tables tell us more about the passage of time in our tests:

```
Wait interval (1421 in total):
28.85% <10
25.83% 10-19
23.15% 20-29
15.76% 30-39
 5.77% 40-49
 0.63% 50-59

Wait until (1421 in total):
14.07% 100-199
12.03% 1000-1999
 9.29% 200-299
 8.94% 300-399
 7.67% 400-499
 ...
```

(continues on next page)

(continued from previous page)

```
 2.32% 2000-2999
 ...
```

The first table shows us *how long* we waited at each individual occurrence of `WaitUntil`: mostly under 30 slots, but up to 59 slots at a maximum. The second table shows us which slot numbers we waited until: we can see that many tests ran for several hundred slots, and indeed, some ran for over 2000 slots.

Luckily, waiting is cheap, but since we are performing fewer useful actions in each test, then we should probably run more tests overall for the same level of confidence in our code.

No locked funds?

We still need to test that we can recover all tokens from the escrow, and do so fairly. Recall our previous finishing strategy:

```
finishingStrategy :: DL EscrowModel ()
finishingStrategy = do
    contribs <- viewContractState contributions
    monitor (tabulate "Refunded wallets" [show . Map.size $ contribs])
    sequence_ [action $ Refund w | w <- testWallets, w `Map.member` contribs]
```

If we just use this as it is, it will fail. As before, we begin by testing `prop_FinishEscrow`, before we worry about unilateral strategies for individual wallets:

```
> quickCheck prop_FinishEscrow
*** Failed! Falsified (after 5 tests and 5 shrinks):
BadPrecondition
  [Do $ Init (Slot {getSlot = 3}) [],
   Do $ Pay (Wallet 3) 2]
  [Action (Refund (Wallet 3))]
  (EscrowModel {_contributions = fromList [(Wallet 3,Value (Map [(,Map [("",
↪2000000)])]))],
                _targets = fromList [],
                _refundSlot = Slot {getSlot = 3},
                _phase = Running})
```

In this test we set the deadline to slot 3, make a payment, and then the finishing strategy attempts to refund the payment… in slot two. It doesn't work: the precondition forbids a refund in that slot. So we have to adapt our finishing strategy, which must simply wait until the deadline before refunding the contributions.

```
finishingStrategy :: CM.DL EscrowModel ()
finishingStrategy = do
    contribs <- CM.viewContractState contributions
    CM.monitor (tabulate "Refunded wallets" [show . Map.size $ contribs])
    waitUntilDeadline                                                -- NEW!!!
    sequence_ [CM.action $ Refund w | w <- testWallets, w `Map.member` contribs]
```

To wait until the deadline, we use `Plutus.Contract.Test.ContractModel.Interface.waitUntilDL`; since this fails if we try to wait until a slot in the past, then we have to check the `Plutus.Contract.Test.ContractModel.Interface.currentSlot` (maintained by the model) before we decide whether or not to wait.

```
waitUntilDeadline :: CM.DL EscrowModel ()
waitUntilDeadline = do
```

(continues on next page)

(continued from previous page)

```
    deadline <- CM.viewContractState refundSlot
    slot     <- CM.viewModelState CM.currentSlot
    when (slot < deadline) $ CM.waitUntilDL deadline
```

With this extended strategy, the property passes:

```
> quickCheck prop_FinishEscrow
+++ OK, passed 100 tests.

Actions (3588 in total):
68.87% WaitUntil
20.71% Pay
 4.77% Refund
 3.18% Redeem
 2.48% Init

Refunded wallets (100 in total):
67% 0
13% 2
 7% 1
 6% 3
 6% 4
 1% 5
```

The strategy works, but the statistics we gathered on the number of wallets to be refunded in each test are a little suspect. *In two thirds of the tests, there were no refunds to be made!* This is not ideal, given that we are testing whether or not our refund strategy works.

This leads us to wonder: *which phase of the test did we reach* before testing our finishing strategy? To find out, we can just add a couple of lines to the `finishingStrategy` code, to `Plutus.Contract.Test.ContractModel.Interface.monitor` the phase:

```
finishingStrategy :: DL EscrowModel ()
finishingStrategy = do
    contribs <- viewContractState contributions
    monitor (tabulate "Refunded wallets" [show . Map.size $ contribs])
    phase <- viewContractState phase           -- NEW!!!
    monitor $ tabulate "Phase" [show phase]    -- NEW!!!
    waitUntilDeadline
    sequence_ [action $ Refund w | w <- testWallets, w `Map.member` contribs]
```

Testing the property again, we see

```
Phase (100 in total):
68% Refunding
32% Running
```

So in two thirds of our tests, we had already reached the `Refunding` phase before the finishing strategy was invoked–which means, in many cases, that the addition we made to the strategy was not needed.

While we certainly want to run *some* tests of the finishing strategy starting in the `Refunding` phase, two thirds seems far too many. How can we ensure that more tests invoke the strategy in the `Running` phase? The simplest way is just to *choose longer deadlines*. There is no particular reason why QuickCheck's default positive integer distribution should be the right one for deadlines. The simplest way to increase the values chosen is just to apply QuickCheck's `scale` combinator to the generator concerned:

```
  arbitraryAction s
    | s ^.contractState . phase == Initial
      = Init <$> (Slot . getPositive <$> scale (*10) arbitrary) <*> arbitraryTargets
```

Here we scale the positive integer generator by multiplying the QuickCheck size parameter by ten before generating; the effect is to increase the range of values by a factor of ten.

Is ten the right number? The only way to tell is to run tests and measure how often we reach the refunding stage:

```
> quickCheck . withMaxSuccess 1000 $ prop_FinishFast
+++ OK, passed 1000 tests.

Phase (1000 in total):
81.5% Running
18.5% Refunding

Refunded wallets (1000 in total):
34.1% 0
18.5% 1
17.3% 2
13.5% 3
11.2% 4
 5.4% 5
```

It seems that we reach the refunding stage in around 20% of tests, which seems reasonable. Moreover the propertion of cases in which there are no refunds to be made is now lower–one third instead of two thirds. So this is a useful improvement.

Finally, we also need to update the unilateral strategy for each wallet in the same way. Once we have done so, then `prop_NoLockedFunds` passes again.

Digression: testing the model alone for speed

We ran a thousand tests to measure the proportion that reach the refunding stage, because one hundred tests is rather few to estimate this percentage from. In fact even a thousand tests is rather few to get accurate results; repeating that thousand-test run ten times yielded a refunding-percentage ranging from 17.4% to 21.6%. Ideally one might run millions of tests to measure the distribution, so we can tune the generation more accurately. Yet running a thousand tests is already quite slow, because of the speed of the emulator.

However, *it is not actually necessary to run the tests, to measure their distribution*! The measurements we are making *depend only on the model*, and so we can make them much more rapidly by taking the emulator out of the test. This is simple to do: recall we defined `prop_FinishEscrow` by

```
prop_FinishEscrow :: Property
prop_FinishEscrow = CM.forAllDL finishEscrow prop_Escrow
```

which *generates* a test case from the dynamic logic test scenario `finishEscrow`, and then *runs* it using `prop_Escrow`. All we have to do to take out the emulator is to replace `prop_Escrow` by the property that is always `True`:

```
prop_FinishFast :: Property
prop_FinishFast = forAllDL finishEscrow $ const True
```

This property generates tests in exactly the same way, and gathers statistics in the same way (and checks preconditions in the same way), but does not actually run the test on the emulator. In other words, it's an excellent test of the *model*, and can be used to tune it (or find bugs in it) without the cost of emulation.

With this version, we can at least run 100,000 tests in a short time, and obtain much more accurate statistics:

```
> quickCheck . withMaxSuccess 100000 $ prop_FinishFast
+++ OK, passed 100000 tests.

Phase (100000 in total):
80.514% Running
19.486% Refunding

Refunded wallets (100000 in total):
34.204% 0
18.387% 1
17.514% 2
14.877% 3
10.016% 4
 5.002% 5
```

The results confirm that the distribution of test cases is reasonably good.

Exercises

You will find the code discussed in this section in `Spec.Tutorial.Escrow4`.

1. Run `quickCheck prop_Escrow` and observe the distributions reported. You will see that, even though we have extended the escrow deadlines, many actions are still rejected by their preconditions. Adapt the *generator* for actions, so that it only generates each action during the correct phase. How does that affect the proportion of rejected actions?
2. The supplied code still has a buggy `walletStrategy`. Verify this by checking that `prop_NoLockedFunds` fails, and inspect the counterexample. Correct the `walletStrategy`, and veryify that `prop_NoLockedFunds` now passes.
3. The code also contains a fast version of `prop_NoLockedFunds` that does not run the emulator. Use *this* property to test your model, with and without the fix to the `walletStrategy`. You should find that the bug is found anyway (it is at the model level), and that verifying that it has been fixed runs satisfyingly faster.
4. Modify the provided code to *remove* the scaling we applied to the deadline generator, and test `prop_FinishFast` repeatedly to judge the effect on the test case distribution. Reinsert the bug in `walletStrategy`, and use

   ```
   quickCheck . withMaxSuccess 10000 $ prop_NoLockedFundsFast
   ```

   to find it. Run this repeatedly, and make an estimate of the number of tests needed to find the bug. Reinsert the scaling, and repeat your estimate. Hopefully this will help persuade you of the value of tuning your test case distributions!

Measuring coverage of on-chain code

It is always good practice to measure the source-code coverage of tests. Coverage information provides a sanity check that nothing has been missed altogether: while covering a line of code is no guarantee that a bug in that line will be revealed, *failing to cover* a line of code *does* guarantee that any bug there will *not* be found. For critical code, it is reasonable to aim for 100% coverage.

Coverage of Haskell code can be measured using the Haskell Program Coverage toolkit; we will not discuss this further here. But while this works well for measuring the coverage of *off-chain* code, it does not apply to *on-chain* code, because this is compiled using the Plutus compiler and executed on the blockchain, rather than by GHC. If we

want to measure the coverage of *on-chain* code–which is the most critical code in a Plutus contract–then we need to use a separate tool. This is what we cover in this section.

Adding a coverage index

In order to generate a coverage report, the framework needs to know

1. what was covered by tests,
2. what should have been covered.

Indeed, the most important part of a coverage report is often the parts that were *not* covered by tests. This latter information–what should be covered–is represented by a `PlutusTx.Coverage.CoverageIndex` that the Plutus compiler constructs. Since the Plutus compiler is invoked using Template Haskell in the code of the contract itself, then this is where we have to save, and export, the coverage index. That is, we must make additions to the code of a contract in order to enable coverage measurement.

To do so, we first inspect the code, and find all the occurrences of `PlutusTx.compile`. In the case of the escrow contract, they are in the definition of `Plutus.Contracts.Escrow.typedValidator`:

```
typedValidator :: EscrowParams Datum -> Scripts.TypedValidator Escrow
typedValidator escrow = go (Haskell.fmap Scripts.datumHash escrow) where
    go = Scripts.mkTypedValidatorParam @Escrow
        $$(PlutusTx.compile [|| validate ||])
        $$(PlutusTx.compile [|| wrap ||])
    wrap = Scripts.mkUntypedValidator
```

The on-chain code consists of `validate` and `wrap`. The latter is a library function, whose coverage we do not need to measure, so we just add (and export) a definition of a `PlutusTx.Coverage.CoverageIndex` that covers `validate`:

```
covIdx :: PlutusTx.CoverageIndex
covIdx = PlutusTx.getCovIdx $$(PlutusTx.compile [|| validate ||])
```

Note: It is important that the coverage index is computed in the same module as the calls to `PlutusTx.compile`, or else the Haskell compiler–and thus by extension, the Plutus compiler–may produce different code for the coverage index and for the code under test, resulting in misleading coverage reports.

It just remains to *import* the necessary types and functions

```
import PlutusTx.Code qualified as PlutusTx
import PlutusTx.Coverage qualified as PlutusTx
```

and to supply GHC options that cause the Plutus compiler to generate coverage information:

```
{-# OPTIONS_GHC -g -fplugin-opt PlutusTx.Plugin:coverage-all #-}
```

With these additions, the contract implementation is ready for coverage measurement.

Measuring coverage

Once we have created a suitable `PlutusTx.Coverage.CoverageIndex`, we must create a test that uses it. To do so, we need to

1. Run the test using `Plutus.Contract.Test.ContractModel.Interface.quickCheckWithCoverage`, and give it coverage options specifying the coverage index,
2. Pass the (modified) coverage options that `Plutus.Contract.Test.ContractModel.Interface.quickCheckWithCoverage` constructs in to `Plutus.Contract.Test.ContractModel.Interface.propRunActionsWithOptions` (instead of `Plutus.Contract.Test.ContractModel.Interface.propRunActions_`) when we run the action sequence, and
3. (Ideally) visualize the resulting `PlutusTx.Coverage.CoverageReport` as annotated source code.

Here is the code to do all this (we also need to import `Plutus.Contract.Test.Coverage`):

```
check_propEscrowWithCoverage :: IO ()
check_propEscrowWithCoverage = do
  cr <- CM.quickCheckWithCoverage stdArgs (set coverageIndex covIdx␣
↪defaultCoverageOptions) $ \covopts ->
    withMaxSuccess 1000 $
      CM.propRunActionsWithOptions @EscrowModel CM.
↪defaultCheckOptionsContractModel covopts
        (const (pure True))
  writeCoverageReport "Escrow" cr
```

First we call `Plutus.Contract.Test.ContractModel.Interface.quickCheckWithCoverage` with options containing `covIdx`; it passes modified options to the rest of the property. We test the property 1000 times, so that we are very likely to cover all the reachable code in the tests. We cannot just reuse `prop_Escrow`, because we must pass in the modified coverage options `covopts` when we run the actions, but otherwise this is just the same as `prop_Escrow`. The result returned by `Plutus.Contract.Test.ContractModel.Interface.quickCheckWithCoverage` is a `PlutusTx.Coverage.CoverageReport`, which is difficult to interpret by itself, so we bind it to `cr` and then generate an HTML file `Escrow.html` using `Plutus.Contract.Test.Coverage.writeCoverageReport`.

Running this does take a little while, because we run a large number of tests; on the other hand, diagnosing *why* a part of the code has not been covered can be very time-consuming, and is wasted effort if the reason is simply that we were unlucky when we ran the tests. It is worth waiting a few minutes for more accurate coverage data, before starting this kind of diagnosis.

Quite a lot of output is generated, including lists of coverage items that were covered respectively not covered. We shall ignore these for now; the same information is presented much more readably in the generated HTML file. But note that we do see statistics on endpoint invocations:

```
> check_propEscrowWithCoverage
+++ OK, passed 1000 tests:
63.1% Contract instance for W[4] at endpoint pay-escrow
62.5% Contract instance for W[1] at endpoint pay-escrow
62.5% Contract instance for W[2] at endpoint pay-escrow
61.2% Contract instance for W[3] at endpoint pay-escrow
60.8% Contract instance for W[5] at endpoint pay-escrow
29.1% Contract instance for W[5] at endpoint redeem-escrow
28.2% Contract instance for W[1] at endpoint redeem-escrow
27.4% Contract instance for W[3] at endpoint redeem-escrow
25.8% Contract instance for W[2] at endpoint redeem-escrow
25.6% Contract instance for W[4] at endpoint redeem-escrow
 4.5% Contract instance for W[1] at endpoint refund-escrow
```

(continues on next page)

(continued from previous page)

```
 4.1% Contract instance for W[2] at endpoint refund-escrow
 3.9% Contract instance for W[4] at endpoint refund-escrow
 3.5% Contract instance for W[3] at endpoint refund-escrow
 3.3% Contract instance for W[5] at endpoint refund-escrow

...
```

This table tells us what percentage of test cases made a call to each endpoint from the given wallet; for example, 63.1% of test cases made (somewhere) a call to the `pay-escrow` endpoint from wallet 4. As we can see, the `pay-escrow` endpoint is called in most tests from each wallet, `redeem-escrow` is a bit rarer, and `refund-escrow` is used quite rarely. Most serious, of course, would be if one of the endpoints doesn't appear in this table at all.

It is possible to supply coverage goals for each wallet/endpoint combination via an additional coverage option. We don't consider this further here, except to note that by default the framework expects each combination to appear in 20% of tests, and so we get warnings in this case:

```
Only 4.5% Contract instance for W[1] at endpoint refund-escrow, but expected␣
↪20.0%
Only 4.1% Contract instance for W[2] at endpoint refund-escrow, but expected␣
↪20.0%
Only 3.5% Contract instance for W[3] at endpoint refund-escrow, but expected␣
↪20.0%
Only 3.9% Contract instance for W[4] at endpoint refund-escrow, but expected␣
↪20.0%
Only 3.3% Contract instance for W[5] at endpoint refund-escrow, but expected␣
↪20.0%
```

These warnings can be eliminated by specifying more appropriate (lower) coverage goals for these endpoint calls.

Interpreting the coverage annotations

Running the test above writes annotated source code to `Escrow.html`. The entire contents of the file are reproduced *here*. The report contains all of the on-chain code provided in the `PlutusTx.Coverage.CoverageIndex`, together with a few lines of code around it for context. Off-chain code appears in grey, so it can be distinguished. On-chain code on a white background was covered by tests, and we know no more about it. Code on a red or green background was also covered, but it is a boolean expression, and only took one value (red for `False`, green for `True`). Orange code on a black background is on-chain code that was not covered at all–and thus may represent a gap in testing.

Looking at the last section of code in the report,

we see that it is the construction of the coverage index, and parts of this code are labelled on-chain and uncovered. We can ignore this, it's simply an artefact of the way the code labelling is done.

More interesting is the second section of the report:

This is the main validator, and while some of its code is coloured white, much of it is coloured green. This means the checks in this function always returned `True` in our tests; we have not tested the cases in which they return `False`.

This does indicate a weakness in our testing: since these checks always passed in our tests, then those tests would *also* have passed if the checks were removed completely (replaced by `True`)–but the contract would have been quite wrong. We will return to this point later, when we discuss *negative testing*. For now, though, we just note that *if the checks had returned* `False`, *then the transaction would have failed*–and the off-chain code is, of course, designed not

to submit failing transactions. So, in a sense, we should expect this code to be coloured green–at least, when we test through well-designed off-chain endpoints, as we have been doing.

This code fragment also contains some entirely uncovered code–the strings passed to `PlutusTx.Trace.traceIfFalse` to be used as error messages if a check fails. Since correct off-chain code never submits failing transactions, then these error messages are never used–and hence the code is labelled as 'uncovered'. Again, this is not really a problem.

The most interesting part of the report is the first section:

This is the function that is used to check that each target payment is made when the escrow is redeemed, and as we see from the coverage report, there are two cases, of which only one has been tested. In fact the two cases handle payments to a wallet, and payments to a script, and the second kind of payment is *not tested at all* by our tests–yet it is handled by entirely different code in the on-chain function.

This exposes a serious deficiency in the tests developed so far: they give us no evidence at all that target payments to a script work as intended. To test this code as well, we would need to add 'proxy' contracts to the tests, to act as recipients for such payments. We leave making this extension as an exercise for the reader.

The generated coverage report

This is the generated coverage report in its entirety:

Crashes, and how to tolerate them

One awkward possibility, that we cannot avoid, is that a contract instance might crash during execution–for example, because of a power failure to the machine it is running on. We don't want anything to be lost permanently as a result–it should be possible to recover by restarting the contract instance, perhaps in a different state, and continue. Yet how should we test this? We need to deliberately crash and restart contracts in tests, and check that they still behave as the model says they should.

The `Plutus.Contract.Test.ContractModel.Interface.ContractModel` framework provides a simple way to *extend* a contract model, so that it can test crash-tolerance too. If `m` is a `Plutus.Contract.Test.ContractModel.Interface.ContractModel` instance, then so is `WithCrashTolerance m`–and testing the latter model will insert actions to crash and restart contract instances at random. To define a property that runs these tests, all we have to do is import `Plutus.Contract.Test.ContractModel.CrashTolerance` and include `Plutus.Contract.Test.ContractModel.CrashTolerance.WithCrashTolerance` in the type signature:

```
prop_CrashTolerance :: CM.Actions (WithCrashTolerance EscrowModel) ->␣
↪Property
prop_CrashTolerance = CM.propRunActions_
```

(The actual code here is the same as `prop_Escrow`, only the type is different).

We do have to provide a little more information before we can run tests, though.

1. Firstly, we cannot expect to include an action in a test, when the contract(s) that should perform the action are not running. We thus need to tell the framework whether or not an action is *available*, given the contracts currently running.
2. Secondly, when we restart a contract it may need to take some recovery action, and so we must be able to give it the necessary information to recover. We achieve this by specifying possibly-different contract parameters to use, when a contract is restarted. These parameters may depend on the model state.

We provide this information by defining an instance of the `Plutus.Contract.Test.ContractModel.CrashTolerance.CrashTolerance` class:

```
instance CrashTolerance EscrowModel where
  available (Init _ _) _      = True
  available a          alive = (Key $ WalletKey w) `elem` alive
    where w = case a of
                Pay w _  -> w
                Redeem w -> w
                Refund w -> w
                Init _ _ -> undefined

  restartArguments s WalletKey{} = escrowParams' slot tgts
    where slot = s ^. CM.contractState . refundSlot
          tgts = Map.toList (s ^. CM.contractState . targets)
```

The `Plutus.Contract.Test.ContractModel.CrashTolerance.available` method returns `True` if an action is available, given a list of active contract keys `alive`; since contract instance keys have varying types, then the list actually contains keys wrapped in an existential type, which is why the `Key` constructor appears there.

The `Plutus.Contract.Test.ContractModel.CrashTolerance.restartArguments` method provides the parameter for restarting an escrow contract, which in this case can be just the same as when the contract was first started. We need to recover the targets from the model state, in which they are represented as a `Map Wallet Value`, so we convert them back to a list and refactor the `escrowParams` function so we can give `escrowParams'` a list of `(Wallet, Value)` pairs, rather than a list of `(Wallet, Int)`:

```
escrowParams :: Slot -> [(Wallet, Integer)] -> EscrowParams d
escrowParams s tgts = escrowParams' s [(w, Ada.adaValueOf (fromInteger n)) |␣
↪(w,n) <- tgts]

escrowParams' :: Slot -> [(Wallet, Value.Value)] -> EscrowParams d
escrowParams' s tgts' =
  EscrowParams
    { escrowTargets  = [ payToPaymentPubKeyTarget␣
↪(mockWalletPaymentPubKeyHash w) v | (w,v) <- tgts' ]
    , escrowDeadline = scSlotZeroTime def + POSIXTime (getSlot s *␣
↪scSlotLength def)
    }
```

It is possible to define the effect of crashing or restarting a contract instance on the *model* too, if need be, by defining additional methods in this class. In this case, though, crashing and restarting ought to be entirely transparent, so we can omit them.

Surprisingly, the tests do not pass!

```
> quickCheck prop_CrashTolerance
*** Failed! Assertion failed (after 24 tests and 26 shrinks):
Actions
 [Init (Slot {getSlot = 6}) [(Wallet 1,2),(Wallet 4,2)],
  Crash (WalletKey (Wallet 4)),
  Restart (WalletKey (Wallet 4)),
  Pay (Wallet 4) 4,
  Redeem (Wallet 1)]
Expected funds of W[4] to change by
  Value (Map [(,Map [("",-2000000)])])
but they changed by
  Value (Map [(,Map [("",-4000000)])])
a discrepancy of
```

(continues on next page)

(continued from previous page)

```
  Value (Map [(,Map [("",-2000000)])])
Expected funds of W[1] to change by
  Value (Map [(,Map [("",2000000)])])
but they did not change
Contract instance log failed to validate:
...
Slot 5: 00000000-0000-4000-8000-000000000000 {Wallet W[1]}:
          Contract instance stopped with error: RedeemFailed␣
↪NotEnoughFundsAtAddress
...
```

Here we simply set up targets with two beneficiaries, crash and restart wallet 4, pay sufficient contributions to cover the targets, and then try to redeem the escrow, which seems straightforward enough, and yet the redemption thinks there are not enough funds in the escrow, *even though we just paid them in*!

This failure is a little tricky to debug. A clue is that the *payment* was made by a contract instance that has been restarted, while the *redemption* was made by a contract that has not. Do the payment and redemption actually refer to the same escrow? In fact the targets supplied to the contract instance are not necessarily exactly the same: the contract receives a *list* of targets, but in the model we represented them as a *map*–and converted the list of targets to a map, and back again, when we restarted the contract. That means the *order* of the targets might be different.

Could that make a difference? To find out, we can just *sort* the targets before passing them to the contract instance, thus guaranteeing the same order every time:

```
escrowParams' :: Slot -> [(Wallet,Value)] -> EscrowParams d
escrowParams' s tgts' =
  EscrowParams
    { escrowTargets  = [ payToPaymentPubKeyTarget␣
↪(mockWalletPaymentPubKeyHash w) v
                       | (w,v) <- sortBy (compare `on` fst) tgts' ]␣
↪        -- NEW!!
    , escrowDeadline = scSlotZeroTime def + POSIXTime (getSlot s *␣
↪scSlotLength def)
    }
```

Once we do this, the tests pass. We can also see from the resulting statistics that quite a lot of crashing and restarting is going on:

```
> quickCheck prop_CrashTolerance
+++ OK, passed 100 tests.

Actions (2721 in total):
42.48% Pay
24.99% WaitUntil
13.08% Crash
 9.52% Restart
 6.06% Redeem
 3.01% Init
 0.85% Refund
```

Perhaps it's debatable whether or not the behaviour we uncovered here is a *bug*, but it is certainly a feature–it was not obvious in advance that specifying the same targets in a different order would create an independent escrow, but that is what happens. So for example, if a buyer and seller using an escrow contract to exchange an NFT for Ada specify the two targets in different orders, then they would place their assets in independent escrow that cannot be redeemed until the refund deadline passes. Arguably a better designed contract would sort the targets by wallet, as we have done here, before creating any UTXOs, so that the problem cannot arise.

Exercises

You will find the code discussed here in `Spec.Tutorial.Escrow6`, *without* the addition of `sortBy` to `escrowParams`.

1. Run `quickCheck prop_CrashTolerance` to provoke a test failure. Examine the counterexample and the test output, and make sure you understand how the test fails. Run this test several times: you will see the failure in several different forms, with the same underlying cause. Make sure you understand how each failure arises.

   Why does `quickCheck` always report a test case with *two* target payments–why isn't one target enough to reveal the problem?

2. Add sorting to the model, and verify that the tests now pass.

3. An alternative way to fix the model is *not* to convert the targets to a `Map` in the model state, but just keep them as a list of pairs, so that exactly the same list can be supplied to the contract instances when they are restarted. Implement this change, and verify that the tests still pass.

   Which solution do you prefer? Arguably this one reflects the *actual* design of the contract more closely, since the model makes explicit that the order of the targets matters.

Debugging the Auction contract with model assertions

In this section, we'll apply the techniques we have seen so far to test another contract, and we'll see how they reveal some surprising behaviour. The contract we take this time is the auction contract in `Plutus.Contracts.Auction`. This module actually defines *two* contracts, a seller contract and a buyer contract. The seller puts up a `Plutus.V1.Ledger.Value.Value` for sale, creating an auction UTXO containing the value, and buyers can then bid Ada for it. When the auction deadline is reached, the highest bidder receives the auctioned value, and the seller receives the bid.

Modelling the Auction contract

`Spec.Auction` contains a contract model for testing this contract. The value for sale is a custom token, wallet 1 is the seller, and the deadline used for testing is fixed at slot 101; the generated tests just consist of an `Init` action to start the auction, and a series of `Bid` actions by the other wallets.

```
    data Action AuctionModel = Init | Bid Wallet Integer
        deriving (Eq, Show, Data)
```

The model keeps track of the highest bid and bidder, and the current phase the auction is in:

```
data AuctionModel = AuctionModel
    { _currentBid :: Integer
    , _winner     :: Wallet
    , _endSlot    :: Slot
    , _phase      :: Phase
    } deriving (Show, Eq, Data)

data Phase = NotStarted | Bidding | AuctionOver
    deriving (Eq, Show, Data)
```

It is updated by the `Plutus.Contract.Test.ContractModel.Interface.nextState` method on each bid:

```
    nextState cmd = do
        case cmd of
            Init -> do
                phase .= Bidding
                withdraw w1 $ Ada.toValue Ledger.minAdaTxOutEstimated <>␣
↪theToken
                wait 3
            Bid w bid -> do
                currentPhase <- viewContractState phase
                when (currentPhase == Bidding) $ do
                    current <- viewContractState currentBid
                    leader  <- viewContractState winner
                    withdraw w $ Ada.lovelaceValueOf bid
                    deposit leader $ Ada.lovelaceValueOf current
                    currentBid .= bid
                    winner     .= w
                wait 2
```

Note that when a higher bid is received, the previous bid is returned to the bidder.

We only allow bids that are larger than the previous one (which is why `Plutus.Contract.Test.ContractModel.Interface.nextState` doesn't need to check this):

```
    precondition s Init = s ^. contractState . phase == NotStarted
    precondition s (Bid w bid) =
      -- In order to place a bid, we need to satisfy the constraint where
      -- each tx output must have at least N Ada.
      s ^. contractState . phase /= NotStarted &&
      bid >= Ada.getLovelace (Ledger.minAdaTxOutEstimated) &&
      bid > s ^. contractState . currentBid
```

The most interesting part of the model covers what happens when the auction deadline is reached: in contrast to the `Escrow` contract, the highest bid is paid to the seller automatically, and the buyer receives the token. We model this using the `Plutus.Contract.Test.ContractModel.Interface.nextReactiveState` method introduced in section *Modelling the passage of time*

```
    nextReactiveState slot' = do
      end  <- viewContractState endSlot
      p    <- viewContractState phase
      when (slot' >= end && p == Bidding) $ do
        w   <- viewContractState winner
        bid <- viewContractState currentBid
        phase .= AuctionOver
        deposit w $ Ada.toValue Ledger.minAdaTxOutEstimated <> theToken
        deposit w1 $ Ada.lovelaceValueOf bid
```

Finally we can define the property to test; in this case we have to supply some options to initialize wallet 1 with the token to be auctioned:

```
prop_Auction :: Actions AuctionModel -> Property
prop_Auction script =
    propRunActionsWithOptions (set minLogLevel Info options)␣
↪defaultCoverageOptions
        (\ _ -> pure True)  -- TODO: check termination
        script
```

The only important part here is `options`, which is defined as follows:

```
-- | The token that we are auctioning off.
theToken :: Value
theToken =
    -- This currency is created by the initial transaction.
    someTokenValue "token" 1

-- | 'CheckOptions' that includes 'theToken' in the initial distribution of␣
↪Wallet 1.
options :: CheckOptions
options = defaultCheckOptionsContractModel
    & changeInitialWalletValue w1 ((<>) theToken)
```

Unsurprisingly, the tests pass.

```
> quickCheck prop_Auction
+++ OK, passed 100 tests.

Actions (2348 in total):
85.82% Bid
10.35% WaitUntil
 3.83% Init
```

No locked funds?

Now we have a basic working model of the auction contract, we can begin to test more subtle properties. To begin with, can we recover the funds held by the contract? The strategy to try is obvious: all we have to do is wait for the deadline to pass. So `prop_FinishAuction` is very simple:

```
finishAuction :: DL AuctionModel ()
finishAuction = do
    anyActions_
    finishingStrategy
    assertModel "Locked funds are not zero" (symIsZero . lockedValue)

finishingStrategy :: DL AuctionModel ()
finishingStrategy = do
    slot <- viewModelState currentSlot
    end  <- viewContractState endSlot
    when (slot < end) $ waitUntilDL end

prop_FinishAuction :: Property
prop_FinishAuction = forAllDL finishAuction prop_Auction
```

This property passes too:

```
> quickCheck prop_FinishAuction
+++ OK, passed 100 tests.

Actions (3152 in total):
84.77% Bid
12.25% WaitUntil
 2.98% Init
```

Now, to supply a `Plutus.Contract.Test.ContractModel.Interface.NoLockedFundsProof` we need a general strategy for fund recovery, and a wallet-specific one. Since all we have to do is wait, we can use the *same* strategy as both.

```
noLockProof :: NoLockedFundsProof AuctionModel
noLockProof = defaultNLFP
  { nlfpMainStrategy   = finishingStrategy
  , nlfpWalletStrategy = const finishingStrategy }
```

Surprisingly, *these tests fail*!

```
> quickCheck prop_NoLockedFunds
*** Failed! Assertion failed (after 2 tests and 1 shrink):
DLScript
  [Do $ Init,
   Do $ Bid (Wallet 3) 2000000]

The ContractModel's Unilateral behaviour for Wallet 3 does not match the
actual behaviour for actions:
Actions
 [Var 0 := Init,
  Var 1 := Bid (Wallet 3) 2000000,
  Var 2 := Unilateral (Wallet 3),
  Var 3 := WaitUntil (Slot {getSlot = 101})]
Expected funds of W[1] to change by
  Value (Map [(363d...,Map [("token",-1)])])
but they changed by
  Value (Map [(,Map [("",-2000000)]),(363d...,Map [("token",-1)])])
a discrepancy of
  Value (Map [(,Map [("",-2000000)])])
Expected funds of W[3] to change by
  Value (Map [(363d...,Map [("token",1)])])
but they changed by
  Value (Map [(,Map [("",-2000000)])])
a discrepancy of
  Value (Map [(,Map [("",-2000000)]),(363d...,Map [("token",-1)])])
Test failed.
```

This test just started the auction and submitted a bid from wallet 3, then *stopped all the other wallets* (this is what `Unilateral (Wallet 3)` does), before waiting until the auction deadline. This resulted in a different distribution of funds from the one the model predicts. Looking at the last part of the message, we see that we expected wallet 3 to get the token, *but it did not*; neither did it get its bid back. Wallet 1 did lose the token, though, and in addition lost the 2 Ada required to create the auction UTXO in the first place.

What is going on? The strategy worked in the general case, but failed in the unilateral case, which tells us that *the buyer requires the cooperation of the seller* in order to recover the auctioned token. Why? Well, our description of the contract above was a little misleading: the proceeds of the auction *cannot* be paid out automatically just because the deadline passes; the Cardano blockchain won't do that. Instead, *someone must submit the payout transaction*. In the case of this contract, it's the seller: even though there is no *endpoint call* at the deadline, the seller's off-chain code continues running throughout the auction, and when the deadline comes it submits the payout transaction. So if the seller's contract is stopped, then no payout occurs.

This is not a *very* serious bug, because the *on-chain* code allows anyone to submit the payout transaction, so the buyer could in principle do so. However, the existing off-chain code does not provide an endpoint for this, and so recovering the locked funds would require writing a new version of the off-chain code (or rolling a suitable transaction by hand).

Model assertions, and unexpected expectations.

Looking back at the failed test again, the *expected* wallet contents are actually a little *unexpected*:

```
Actions
 [Var 0 := Init,
  Var 1 := Bid (Wallet 3) 2000000,
  Var 2 := Unilateral (Wallet 3),
  Var 3 := WaitUntil (Slot {getSlot = 101})]
Expected funds of W[1] to change by
  Value (Map [(363d...,Map [("token",-1)])])
but they changed by
  Value (Map [(,Map [("",-2000000)]),(363d...,Map [("token",-1)])])
a discrepancy of
  Value (Map [(,Map [("",-2000000)])])
```

Notice that, even though wallet 3 made a bid of 2 Ada, we *expected* the seller to end up without the token, but *with no extra money*. Wouldn't we expect the seller to end up with 2 Ada?

Because `prop_Auction` passed, then we know that in the absence of a `Unilateral` then the model and the contract implementation agree on fund transfers. But does the model actually predict that the seller gets the winning bid? This can be a little hard to infer from the state transitions themselves; we can check that each action appears to do the right thing, but whether the end result is as expected is not necessarily immediately obvious.

We can address this kind of problem by *adding assertions to the model*. The model tracks the change in each wallet's balance since the beginning of the test, so we can add an assertion, at the point where the auction ends, that checks that the seller loses the token and gains the winning bid. We just a little code to `Plutus.Contract.Test.ContractModel.Interface.nextReactiveState`:

```
    nextReactiveState slot' = do
      end  <- viewContractState endSlot
      p    <- viewContractState phase
      when (slot' >= end && p == Bidding) $ do
        w   <- viewContractState winner
        bid <- viewContractState currentBid
        phase .= AuctionOver
        deposit w $ Ada.toValue Ledger.minAdaTxOutEstimated <> theToken
        deposit w1 $ Ada.lovelaceValueOf bid
        -- NEW!!!
        w1change <- viewModelState $ balanceChange w1  -- since the start of␣
↪the test
        assertSpec ("w1 final balance is wrong:\n  "++show w1change) $
          w1change == toSymValue (inv theToken <> Ada.lovelaceValueOf bid) ||
          w1change == mempty
```

If the boolean passed to `Plutus.Contract.Test.ContractModel.Interface.assertSpec` is `False`, then the test fails with the first argument in the error message.

> **Note:** We do have to allow for the possibility that the auction never started, which is why we include in the assertion the possibility that wallet 1's balance remains unchanged. Without this, the tests fail.

Now `prop_Auction` fails!

```
> quickCheck prop_Auction
*** Failed! Falsified (after 27 tests and 24 shrinks):
Actions
```

(continues on next page)

(continued from previous page)

```
 [Init,
  Bid (Wallet 3) 2000000,
  WaitUntil (Slot {getSlot = 100})]
assertSpec failed: w1 final balance is wrong:
  SymValue {symValMap = fromList [], actualValPart = Value (Map␣
↪[(363d...,Map [("token",-1)])])}
```

Note: The balance change is actually a `Plutus.Contract.Test.ContractModel.Symbolics.SymValue`, not a `Plutus.V1.Ledger.Value.Value`, but as you can see it *contains* a `Plutus.V1.Ledger.Value.Value`, which is all we care about right now.

Even in this simple case, the seller does not receive the right amount: wallet 1 lost the token, but received no payment!

The reason has to do with the minimum Ada in each UTXO. When the auction UTXO is created, the seller has to put in 2 Ada along with the token. When the auction ends, one might expect that 2 Ada to be returned to the seller. But it can't be: *it is needed to create the UTXO that delivers the token to the buyer*! Thus the seller receives 2 Ada (from the buyer's bid) in this example, but this only makes up for the 2 Ada deposited in the auction UTXO, and the seller ends up giving away the token for nothing.

This is quite surprising behaviour, and arguably, the contract should require that the buyer pay the seller 2 Ada *plus* the winning bid, so that the stated bid is equal to the seller's net receipts.

Note: Model assertions can be tested without running the emulator, by using `Plutus.Contract.Test.ContractModel.Interface.propSanityCheckAssertions` instead of `Plutus.Contract.Test.ContractModel.Interface.propRunActions_`. This is very much faster, and enables very thorough testing of the model. Since other tests check that the implementation correponds to the model, then this still gives us valuable information about the implementation.

Crashing the auction

Is the auction crash tolerant? To find out, we just declare that `Bid` actions are available when the corresponding buyer contract is running, define the restart arguments, and the crash-tolerant property (which just replicates the definition of `prop_Auction` with a different type).

```
instance CrashTolerance AuctionModel where
  available (Bid w _) alive = (Key $ BuyerH  w) `elem` alive
  available Init      alive = True

  restartArguments _ BuyerH{}  = ()
  restartArguments _ SellerH{} = ()

prop_CrashTolerance :: Actions (WithCrashTolerance AuctionModel) -> Property
prop_CrashTolerance =
  propRunActionsWithOptions (set minLogLevel Critical options)␣
↪defaultCoverageOptions
        (\ _ -> pure True)
```

Perhaps unsurprisingly, this property fails:

```
> quickCheck prop_CrashTolerance
*** Failed! Assertion failed (after 17 tests and 11 shrinks):
```

(continues on next page)

(continued from previous page)

```
Actions
 [Init,
  Crash SellerH,
  WaitUntil (Slot {getSlot = 100})]
Expected funds of W[1] to change by
  Value (Map [])
but they changed by
  Value (Map [(,Map [("",-2000000)]),
↪(363d3944282b3d16b239235a112c0f6e2f1195de5067f61c0dfc0f5f,Map [("token",-
↪1)])])
a discrepancy of
  Value (Map [(,Map [("",-2000000)]),
↪(363d3944282b3d16b239235a112c0f6e2f1195de5067f61c0dfc0f5f,Map [("token",-
↪1)])])
Test failed.
```

We already know that the auction payout is initiated by the seller contract, so if that contract is not running, then no payout takes place. (Although there are no bids in this counterexample, a payout is still needed–to return the token to the seller). That is why this test fails.

But this is actually not the only way the property can fail. The other failure (which generates some rather long contract logs) looks like this:

```
> quickCheck prop_CrashTolerance
*** Failed! Assertion failed (after 13 tests and 9 shrinks):
Actions
 [Init,
  Crash SellerH,
  Restart SellerH]
Contract instance log failed to validate:
... half a megabyte of output ...
Slot 6: 00000000-0000-4000-8000-000000000004 {Wallet W[1]}:
          Contract instance stopped with error: StateMachineContractError␣
↪(SMCContractError (WalletContractError (InsufficientFunds "Total: Value␣
↪(Map [(,Map [(\"\",9999999997645750)])]) expected: Value (Map␣
↪[(363d3944282b3d16b239235a112c0f6e2f1195de5067f61c0dfc0f5f,Map [(\"token\",
↪1)])])"))))
Test failed.
```

In other words, after a crash, *the seller contract fails to restart*. This is simply because the seller tries to put the token up for auction when it starts, and *wallet 1 no longer holds the token*–it is already in an auction UTXO on the blockchain. So the seller contract fails with an `Wallet.Emulator.Error.InsufficientFunds` error. To continue the auction, we would really need another endpoint to resume the seller, which the contract does not provide, or a parameter to the seller contract which specifies whether to start or continue an auction.

Coverage of the Auction contract

We can generate a coverage report for the `Auction` contract just as we did for the `Escrow` one. The interesting part of the report is:

The auction is defined as a Plutus state machine, which just repeats an `auctionTransition` over and over again. We can see that the state machine itself, and most of the transition code, is covered. However, the `Bid` transition has only been tested in the case where new bid is higher than the old one. Indeed, the tests are designed to respect that precondition. Moroever, the last clause in the `case` expression has not been tested at all–but this is quite OK, because

it returns `Nothing` which the state machine library interprets to mean "reject the transaction". So the uncovered code *could* only be covered by failing transactions, which the off-chain code is designed not to submit.

Exercises

The code discussed here is in `Spec.Auction`.

1. Test the failing properties (`prop_NoLockedFunds` and `prop_CrashTolerance`) and observe the failures.
2. Add the model assertion discussed in *Model assertions, and unexpected expectations.* to the code, and `quickCheck prop_SanityCheckAssertions` to verify that it fails. Change the assertion to say that the seller receives 2 Ada *less* than the bid, and verify that it now passes.

Becoming Level 1 Certification Ready

Level 1 certification of plutus smart contracts relies on the machinery we have discussed in this tutorial. First things first we are going to have a look at the `Plutus.Contract.Test.Certification` module.

This module defines a type `Plutus.Contract.Test.Certification.Certification` paramtereized over a type `m` that should be a `Plutus.Contract.Test.ContractModel.Interface.ContractModel`. This is a record type that has fields for:

1. a `PlutusTx.Coverage.CoverageIndex`,
2. two different types of `Plutus.Contract.Test.ContractModel.Interface.NoLockedFundsProof` (a standard full proof and a light proof that does not require you to provide a per-wallet unilateral strategy),
3. the ability to provide a specialized error whitelist,
4. a way to specify that we have an instance of `Plutus.Contract.Test.ContractModel.CrashTolerance.CrashTolerance` for `m`,
5. unit tests in the form of a function from a `Plutus.Contract.Test.Coverage.CoverageRef` to a `TestTree` (see `Plutus.Contract.Test.checkPredicateCoverage` for how to construct one of these), and
6. named dynamic logic unit tests.

Fortunately, understanding what we need to do to get certification-ready at this stage is simple. We just need to build a `Plutus.Contract.Test.Certification.Certification` object. For example of how to do this, check out `Spec.GameStateMachine.certification` and `Spec.Uniswap.certification`.

You can run level 1 certification locally using the `Plutus.Contract.Test.Certification.Run.certify` function - but at this stage you may find it difficult to read the output of this function. Don't worry! A certification dashboard is on the way!

Exercises

1. Build a certification object for the `Auction` and `Escrow` contracts.

5.3 How-to guides

5.3.1 How to get started with the Plutus Platform

We provide a template repository (deprecated) that you can use to get started quickly. The repository *README* provides up-to-date instructions for how to set it up.

Further reading

This would be a good time to try one of the *tutorials*.

5.3.2 How to write a scalable Plutus app

Every dapp has its own requirements for throughput and performance, often quantified in terms of number of concurrent users, number of events per time window, and so forth. When building a dapp, developers need to find a design that achieves the desired performance within the constraints of the underlying blockchain.

The Cardano blockchain uses the *extended UTXO model with scripts* to represent the ledger state. The UTXO model with its graph structure is fundamentally different from the account model used by some existing smart-contract enabled blockchains. As a result, the design patterns that work for dapps on account-based blockchains do not translate directly to Cardano. We need new design patterns, because the underlying representation of the data is different.

In this document we discuss the structure of the UTXO model and its implications for scalability, arriving at a list of *scalability guidelines* that can guide the design of distributed applications on Plutus. We use the *example of a decentralised exchange* to illustrate the pros and cons of the different approaches.

The building blocks

Let's look at the building blocks of the ledger's scripting features.

Transaction outputs

At its core, the on-chain state of our app is captured in *transaction outputs*. The extended UTXO model has two types of transaction outputs: Script outputs and public key outputs. Public key outputs contain the hash of a public key. Script outputs contain two hashes: The hash of a validator script and the hash of a data value. Both types of outputs hold crypto currency values.

They differ in the kind of witness that is required to spend them. A public key output can be spent by a transaction that carry a signature of the private key that corresponds to the output's public key. A script output can be spent by a transaction that carries the validator script which hashes to the script hash, and two pieces of data - the redeemer and the datum. The latter must be the value that hashes to the output's datum hash. In addition, any transaction attempting to spend a script output must meet the requirements set out in the validator script.

Output type	Witness	Address	Data
Public key	Signature	Public key hash	N/A
Script	composite	Script hash	Datum hash

The address of an output is used to group similar outputs together on the ledger. If we have a validator script then we can look at the ledger to see which outputs are currently at the script's address, that is, which outputs are locked by the hash of our validator script.

Each transaction output is uniquely identified by two pieces of data: The hash of the transaction that produced the output, and the index of this output in the transaction's list of outputs.

State of transaction outputs

It is tempting to think of the datum hash of a script output as the *state* of the output. After all, the datum hash is a piece of data that must be supplied with the spending transaction, and it doesn't affect the script address. This is a bit misleading because the datum hash of a transaction output never changes. The datum hash is determined by the transaction that produces the output, and it is immutable.

There is only one bit of information that changes: Whether the output has been spent. Every transaction output starts in the **unspent** state, then it may transition to the **spent** state. The transition happens when a transaction that spends the output is appended to the blockchain.

Note: Reality is slightly more complicated: Since transactions can be rolled back, the state of a transaction output can change back to *unspent* if the spending transaction gets rolled back. Only when a certain number of blocks have been added is the spending transaction firmly committed and the state of the output cannot change back to *unspent* anymore. Rollbacks are reflected in the *Plutus Application Backend (PAB)*'s interface for *dealing with blockchain events*, and they need to be considered when thinking about the business logic of your applications.

Changing the state of our app

Let's think of the on-chain state of our application as a **set of unspent transactions outputs** (a subset of the global UTXO set that is maintained by the ledger). There are no hard restrictions on how many different outputs or addresses our app can have – a one-off trade between two parties only needs a single output, while a complex distributed application with governance, tokens and so forth might involve multiple Plutus scripts and a large number of unspent transaction outputs.

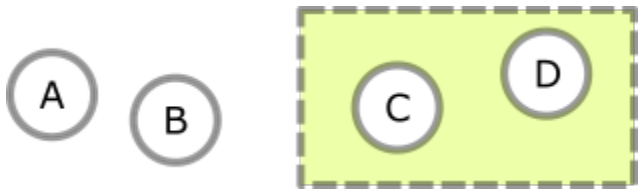


Fig. 8: A UTXO set with four unspent outputs, two of which belong to our app (green box).

Any transaction that changes the set of UTXOs that belong to our app also changes the state of the app. Transactions can change our application's set of UTXOs by adding new outputs to it, or by spending existing unspent outputs.

The number of new outputs that can be added to our application state in a single block is ultimately limited by the block size. The block size is a protocol parameter that can be adapted over time to match the growth in smart contract usage. The number of outputs that can be removed from our application state in a block is limited both by the block size *and* by the number of outputs that are currently unspent.

To produce a script output we only need to provide the hashes of the script and the datum, whereas to spend a script output we need to provide the script, datum and redeemer values in full. Transactions that produce script outputs therefore tend to be smaller (and cheaper) than transactions that spend them. In addition, a transaction that only produces script outputs and doesn't spend them cannot fail due to UTXO congestion on script outputs.

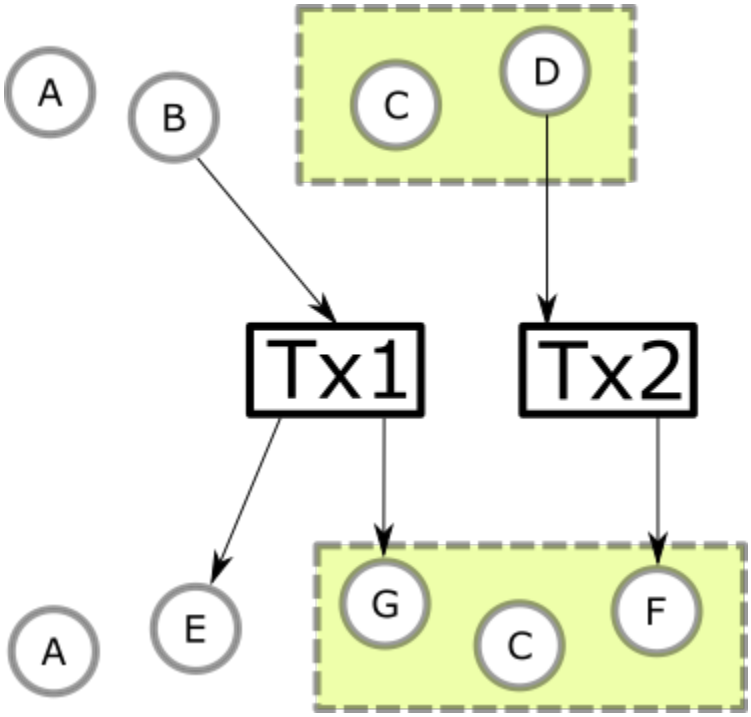


Fig. 9: A block with two transactions, Tx1 and Tx2, both changing the on-chain state of our app. Adding the block to the chain caused outputs (B) and (D) to become spent, and (E), (F), and (G) to be created in the unspent state.

UTXO congestion

Two transactions conflict if they try to spend the same unspent output. When this happens, only one of the transactions is added to the ledger. The other transaction is rejected. The author of the rejected transaction must build a new transaction spending a different output and try again. If many users are trying to spend the same output there can quickly arise a situation where most users spend a lot of time waiting, because their clients all try to spend the same output. Almost all of them will fail and try again in the next block.

Congestion can happen on any type of output, but the chances of it happening to public key outputs are low, because the private key required to spend the output is usually only known to a single wallet, which can keep track of which outputs it has attempted to spend. For example, let's assume the user wants to make a payment and run a Plutus script in two different transactions. When the wallet has constructed and submitted the payment transaction, it remembers the public key inputs that were used to fund it. Then when the wallet balances the Plutus transaction it knows not to use the same public key inputs again, even if the inputs are still technically unspent at that time (while the payment transaction is in the mempool).

Script outputs are more likely to fall victim to UTXO congestion *if* they can be spent by more than one party. To avoid UTXO congestion we should therefore design our system such that the number of simultaneous attempts made to spend the same script output is as low as possible. What does this mean for the state of our distributed application?

We need to minimise the number of transactions that are trying to spend the same script output. At the same time, we should design the system so that the access patterns which require relatively high throughput can be realised exclusively by producing script outputs, not by spending script outputs.

Minting Policy Scripts

Another way to run Plutus scripts on the ledger is by creating tokens with a custom minting policy. From a scalability perspective, minting scripts are great because they do not consume a script input. They aren't subject to UTXO congestion on script outputs, while allowing us to run a script in the transaction that *produces* the tokens. Seeing the token on the ledger is therefore evidence that the minting policy script has been executed successfully (as opposed to seeing a script output on the ledger, which can be produced without running any scripts at all). Whenever we need to run a Plutus script in our application we should ask ourselves if we can make this script a minting policy, and only use validators if we absolutely have to store some information or crypto currency value in a transaction output.

Scalability guidelines

The discussion of the UTXO model above can be summarised in three guiding principles for avoiding bottlenecks in your app:

1. **Minimise the number of transactions that are trying to spend the same script output.** The number of entities (users) that try to spend a given script output at a single time should be small. It should certainly not grow with the total number of concurrent users of the system. A good distributed app design ensures that the number of UTXOs that make up the application state grows with the number of active users, and that each user interacts with a small subset of the application's UTXOs only.
2. **Decouple the spending of script outputs from producing script outputs.** Transactions that don't spend script outputs are not liable to UTXO congestion on script outputs.
3. **Use minting policy scripts and tokens.** Minting policies are Plutus scripts that can be run without spending a script output. Besides being useful for NFTs and other currency-like applications, tokens created by Plutus minting policies can act as *evidence* that some event happened in the past. For example, we could write a state machine that produces a token in its last transition. This token can then be used as proof that the state machine has finished, long after the last output has been spent. In this way, minting policies could be used to implement certain forms of oracles.

Examples

Decentralised exchange

A decentralised exchange (DEX) can be realised either as an automated market maker (AMM) contract or using an order book. The AMM approach results in one UTXO per liquidity pair. This is fine for rarely-traded pairs, but pairs that have even close to one trade per block will soon run into UTXO congestion issues. Frequently traded pairs are better off with an order book model. Each order (bid/ask) is represented as a single UTXO. Creating a new order only requires adding a script output, so it cannot be subject to UTXO congestion. Matching orders is performed by a service that scans the blockchain for script outputs, maintains an order book and creates spending transactions when a match has been made. This is an example decoupling the spending of script outputs from producing them (Guideline 2). It is an instance of the *order book pattern*.

The basic idea could be extended in many different ways. For example, minting policies can be used to enforce payment for market makers or to create governance tokens. If the code was open sourced, anyone could run a match making service and earn fees, thus creating incentives for fast settlement. This would result in a truly decentralised exchange, because the match making could be performed by anyone without central coordination.

Marlowe

Marlowe is implemented using the Plutus state machine libraries. The number of concurrent users on a given Marlowe instance is fixed and limited, and it rarely exceeds a handful. Updates that require spending and producing the instance UTXO happen with a frequency of much less than once per block. The chances of UTXO congestion happening on a Marlowe contract instance are therefore small (Guideline 1). If they do happen, they only affect a single instance of Marlowe, and not the entire system.

Summary

Apart from the *guidelines*, the main lesson of this article is that Plutus apps need to be designed with the UTXO ledger in mind. Porting an existing contract from an account-based blockchain such as Ethereum is likely to result in *UTXO congestion* if the entire on-chain state of the app is kept in a single unspent output.

5.3.3 How to handle blockchain events

The state of a Cardano dapp often spans *many UTXOs* and it can change with every new block that's added to the chain. Our *off-chain code* needs to react to state changes, for example by informing users that a trade has been settled, or by constructing new transactions in response to actions of other participants. Since many smart contracts are time sensitive, we want to respond to these events as quickly as possible.

With the *Plutus Application Backend (PAB)* we can write reactive off-chain code that deals promptly with blockchain events, using an easy-to-consume interface that wraps some of the complexity of the *distributed ledger*.

Transaction output lifecycle

Transaction outputs are either spent or unspent. They start out as *unspent*, when the transaction that produces them is added to the chain. Later on, another transaction might use them as inputs, so their state changes to *spent*. Once an output has been spent, it can never be "un-spent", or spent a second time – that's a fundamental property of the ledger. However, for certain time after a block of transactions was first appended to the blockchain, it is possible for it to disappear again as result of a *rollback*.

Transaction states

In the presence of rollbacks, transactions have three states that they can switch between: Unknown, tentatively confirmed, and committed.

If we want to respond to a new transaction as quickly as possible (for example, by spending one of its outputs), we must be prepared for the possibility that the transaction is rolled back, invalidating our own transaction.

Note: The fact that rolling back a transaction invalidates all transactions that spend the rolled-back transaction's outputs can be useful for combining multiple actions in a group of transaction that should all be accepted or rejected together.

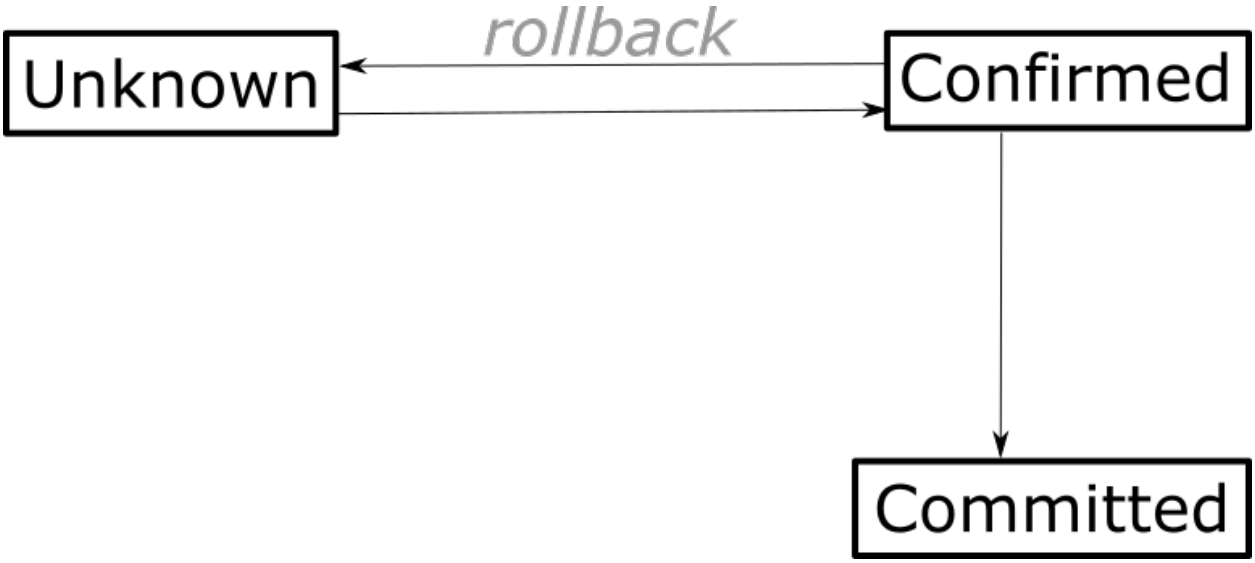


Fig. 10: The state of transactions as observed by the PAB, and possible transitions between them. When the transaction is deep enough in the blockchain, the state changes to *committed* and does not change anymore.

PAB functions for listening to state changes

The PAB has a function for the state of a transaction to change:

```
-- | Wait for the status of a transaction to change
awaitTxStatusChange ::
  forall w s e.
  (AsContractError e)
  => TxId
  -> Contract w s e TxStatus
```

`Plutus.Contract.Request.awaitTxStatusChange` returns a `Plutus.ChainIndex.Types.TxStatus` value with the new state of the transaction.

Note: If you do not want to deal with rollbacks in your application, you can keep calling *awaitTxStatusChange* until the status is *DefinitelyConfirmed*. This will eventually happen for all valid transactions.

In addition, we can use the following functions to wait for outputs to be spent or to appear at a given address on the chain:

```
{-| Wait until one or more unspent outputs are produced at an address.
-}
awaitUtxoProduced ::
  forall w s e .
  (AsContractError e)
  => CardanoAddress
  -> Contract w s e (NonEmpty ChainIndexTx)
```

```
{-| Wait until the UTXO has been spent, returning the transaction that spends it.
-}
awaitUtxoSpent ::
  forall w s e.
```

(continues on next page)

(continued from previous page)

```
  (AsContractError e)
  => TxOutRef
  -> Contract w s e ChainIndexTx
```

With this functions we can implement off-chain code that reacts quickly to on-chain events.

5.3.4 How to analyse the cost and size of Plutus scripts

Running Plutus scripts on a validating node uses CPU time and RAM space, which are paid for by transaction fees. When building a decentralised application in Plutus we need to keep an eye on the size of the transactions that we submit to the network.

The Plutus libraries give us some tools for measuring the resource consumption of our scripts.

Resource use of Plutus scripts

There are two types of resources used by Plutus transactions. First we have the runtime cost – the amount of CPU and RAM used to actually run the script. Then there is the network cost – the size of the transaction, which determines network load and storage need when the transaction is added to the blockchain.

The *Plutus.Trace.Emulator.Extract* module lets us analyse both types of cost for transactions that are produced by the Plutus *emulator*.

`Plutus.Trace.Emulator.Extract.writeScriptsTo` is a function that, given an emulator trace, produces a JSON file for each transaction that is created during that trace.

```
{-| Run an emulator trace and write the applied scripts to a file in Flat format
    using the name as a prefix.
-}
writeScriptsTo
    :: Extract.ScriptsConfig -- ^ Configuration
    -> String -- ^ Prefix to be used for file names
    -> EmulatorTrace a -- ^ Emulator trace to extract transactions from
    -> EmulatorConfig -- ^ Emulator config
    -> IO (Sum Int64, ExBudget) -- Total size and 'ExBudget' of extracted scripts
```

The `Plutus.Trace.Emulator.Extract.Command` argument selects one of two modes of `Plutus.Trace.Emulator.Extract.writeScriptsTo`. The mode determines what kind of data is written to the folder specified in `Plutus.Trace.Emulator.Extract.scPath`.

1. `Plutus.Trace.Emulator.Extract.Scripts` writes the validator scripts, one for each script input that is validated as part of the emulator trace. Here we have the choice between fully applied validators and unapplied validators. *Fully applied* means that we get a *Plutus Core* (PLC) program that can be evaluated to the unit value (or an error). This is the program that the node actually runs when validating the script input, and it is used for determining the script execution cost. *Unapplied* results in the PLC program of the unapplied validator. This tells us the size of the serialised Plutus script that we need to attach to the spending transaction. In both cases the CPU and memory budget in ExUnits will be displayed in the terminal (see sample output below).
2. `Plutus.Trace.Emulator.Extract.Transactions` writes all partial transactions that are sent to the (emulated) wallet for balancing before they are submitted to the network. Each partial transaction results in a JSON file. The *transaction* field of the JSON object contains the actual transaction in the text envelope format used by *cardano-api*. Since the transaction body is hex encoded, we can look at the length of the *cborHex* field and divide it by two in order to get the size of the partial transaction in bytes. Note that the final transaction will be slightly larger, because some additional inputs and outputs will be added by the wallet.

Examples

To see `Plutus.Trace.Emulator.Extract.writeScriptsTo` in action you can run the *plutus-use-cases-scripts* command that is part of the *plutus-use-cases* package in our repository.

Validator scripts

```
cabal run plutus-use-cases-scripts -- ./tmp scripts
```

results in the following output:

```
Writing scripts (fully applied) to: ./tmp
Writing script: ./tmp/auction_1-1.flat (Size: 3.7kB, Cost: ExCPU 309803992, ExMemory␣
↪789488)
Writing script: ./tmp/auction_1-2.flat (Size: 9.1kB, Cost: ExCPU 1122022080, ExMemory␣
↪3410856)
Writing script: ./tmp/auction_1-3.flat (Size: 9.1kB, Cost: ExCPU 1126876612, ExMemory␣
↪3408894)
Writing script: ./tmp/auction_1-4.flat (Size: 3.9kB, Cost: ExCPU 395045625, ExMemory␣
↪989992)
Writing script: ./tmp/auction_2-1.flat (Size: 3.7kB, Cost: ExCPU 309803992, ExMemory␣
↪789488)
Writing script: ./tmp/auction_2-2.flat (Size: 9.1kB, Cost: ExCPU 1122022080, ExMemory␣
↪3410856)
Writing script: ./tmp/auction_2-3.flat (Size: 9.2kB, Cost: ExCPU 1267324633, ExMemory␣
↪3853688)
Writing script: ./tmp/auction_2-4.flat (Size: 9.4kB, Cost: ExCPU 1376566955, ExMemory␣
↪4153874)
Writing script: ./tmp/auction_2-5.flat (Size: 9.1kB, Cost: ExCPU 1126876612, ExMemory␣
↪3408894)
```

Note: The program writes out fully applied validators by default. Fully applied validators are larger than unapplied validators because they contain not just the validator code itself but also all arguments, including the `Plutus.V1.Ledger.Contexts.ScriptContext`. The script context can be quite large as it is a representation of the entire transaction body.

Running the program in the unapplied validator mode gives us a more realistic picture:

```
cabal run plutus-use-cases-scripts -- ./tmp scripts --unapplied-validators
Writing scripts (unapplied) to: ./tmp
Writing script: ./tmp/auction_1-1-unapplied.flat (Size: 2.9kB, Cost: ExCPU 309803992,␣
↪ExMemory 789488)
Writing script: ./tmp/auction_1-2-unapplied.flat (Size: 8.1kB, Cost: ExCPU 1122022080,
↪ ExMemory 3410856)
Writing script: ./tmp/auction_1-3-unapplied.flat (Size: 8.1kB, Cost: ExCPU 1126876612,
↪ ExMemory 3408894)
Writing script: ./tmp/auction_1-4-unapplied.flat (Size: 2.9kB, Cost: ExCPU 395045625,␣
↪ExMemory 989992)
Writing script: ./tmp/auction_2-1-unapplied.flat (Size: 2.9kB, Cost: ExCPU 309803992,␣
↪ExMemory 789488)
Writing script: ./tmp/auction_2-2-unapplied.flat (Size: 8.1kB, Cost: ExCPU 1122022080,
↪ ExMemory 3410856)
Writing script: ./tmp/auction_2-3-unapplied.flat (Size: 8.1kB, Cost: ExCPU 1267324633,
↪ ExMemory 3853688)
```

(continues on next page)

(continued from previous page)

```
Writing script: ./tmp/auction_2-4-unapplied.flat (Size: 8.1kB, Cost: ExCPU 1376566955,
↪ ExMemory 4153874)
Writing script: ./tmp/auction_2-5-unapplied.flat (Size: 8.1kB, Cost: ExCPU 1126876612,
↪ ExMemory 3408894)
(...)
```

Now the script sizes are more realistic.

Partial transactions

```
cabal run plutus-use-cases-scripts -- ./tmp transactions -p ./plutus-use-cases/
↪scripts/protocol-parameters.json
```

results in

```
Writing transactions to: ./tmp
Writing partial transaction JSON: ./tmp/auction_1-1.json
Writing partial transaction JSON: ./tmp/auction_1-2.json
Writing partial transaction JSON: ./tmp/auction_1-3.json
Writing partial transaction JSON: ./tmp/auction_1-4.json
Writing partial transaction JSON: ./tmp/auction_2-1.json
Writing partial transaction JSON: ./tmp/auction_2-2.json
Writing partial transaction JSON: ./tmp/auction_2-3.json
(...)
```

Each file contains the partial transaction and some additional information that the wallet uses for balancing.

```
{
    "transaction": {
        "cborHex": "84a500800d800(...)",
        "description": "",
        "type": "Tx AlonzoEra"
    },
    "signatories": [],
    "inputs": [
        {
            "txIn": "0636250aef275497b4f3807d661a299e34e53e5ad3bc1110e43d1f3420bc8fae
↪#6",
            "txOut": {
                "address": "addr1vy6aahffs2sreuu70h8q8jpen98lmmpwc6cy788j6s8xrgcpajqhn
↪",
                "value": {
                    "lovelace": 100000000
                }
            }
        }
    ]
}
```

5.3.5 Plutus Scripts

What is a Plutus spending script?

This is a type of Plutus script that is required to validate the spending of a tx output at its own script address. The tx output at the Plutus script address *must* be associated with a datum hash, otherwise, the tx output will be unspendable! The purpose of this datum hash is to encode the state of the contract. A demonstration of this can be seen in lecture #7 of the Plutus Pioneers Program. The Plutus spending script expects a datum and a redeemer in order to successfully validate the spending of the tx output at its own script address; note that the redeemer is considered to be the user input. These are supplied in the transaction being submitted along with the Plutus script itself and specified transaction execution units.

The transaction execution units are an upper bound or budget of what will be spent to execute the Plutus script. If you don't specify a high enough value to cover script execution, your transaction will still be successful (provided it is a valid tx) but you will lose your collateral. The collateral is the transaction input(s) you specify to be consumed if your script fails to execute. There is a protocol parameter `collateralPercent` that determines what percentage of your inputs you must supply as collateral.

*Note that in order to use a tx input as collateral, it***cannot***reside at a script address; it must reside at a 'normal' payment address and it cannot contain any multi-assets.*

NB: All variable assignments can be looked up in cardano-node/scripts/plutus/example-txin-locking-plutus-script.sh

An example of using a Plutus spending script

Below is an example that shows how to use a Plutus spending script. This is a step-by-step process involving:

- the creation of the `AlwaysSucceeds` Plutus txin script
- sending ADA to the Plutus script address
- spending ADA at the Plutus script address

In this example we will use the AlwaysSucceeds Plutus spending script. In order to execute a Plutus spending script, we require the following:

- Collateral tx input(s) - these are provided and are forfeited in the event the Plutus script fails to execute.
- A Plutus tx output with accompanying datum hash. This is the tx output that sits at the Plutus script address. It must have a datum hash, otherwise, it is unspendable.
- The Plutus script serialized in the text envelope format. `cardano-cli` expects Plutus scripts to be serialized in the text envelope format.

Creating the `AlwaysSucceeds` Plutus spending script

The plutus-example executable will automagically generate several Plutus scripts in the CLI-compatiable text envelope format.

Run the following commands:

```
cd plutus-example

cabal run exe:plutus-example
```

This will output `always-succeeds-txin.plutus` in the `generated-plutus-scripts` dir.

Setting up a local Alonzo node cluster

There is a convenient script that will set up an Alonzo cluster immediately on your local machine.

Run the following command:

```
cabal install cardano-cli
cabal install cardano-node
./cardano-node/scripts/byron-to-alonzo/mkfiles.sh alonzo
```

Follow the instructions displayed in the terminal to start your Alonzo cluster.

Sending ADA to the script address

In order to require a Plutus script to validate the spending of a tx ouput, we must put the tx output at the script address of the said Plutus script. However, before we do that, we must create a datum hash:

```
> cardano-cli transaction hash-script-data --script-data-value 42
> 9e1199a988ba72ffd6e9c269cadb3b53b5f360ff99f112d9b2ee30c4d74ad88b
```

In this example, the script we are using always succeeds so we can use any datum hash. We calculate the script address as follows:

```
> cabal run exe:plutus-example
> cardano-cli address build --payment-script-file generated-plutus-scripts/always-
↪succeeds-txin.plutus  --testnet-magic 42
> addr_test1wzeqkp6ne3xm6gz391874va4ujgl4kr0e46pf3ey8xsu3jsgkpcj2
```

Now, we should create the tx that will send ADA to the script address of our `AlwaysSucceeds` script:

```
cardano-cli transaction build-raw \
  --alonzo-era \
  --fee 0 \
  --tx-in $txin \
  --tx-out "addr_test1wzeqkp6ne3xm6gz391874va4ujgl4kr0e46pf3ey8xsu3jsgkpcj2+$lovelace
↪" \
  --tx-out-datum-hash␣
↪9e1199a988ba72ffd6e9c269cadb3b53b5f360ff99f112d9b2ee30c4d74ad88b \
  --out-file create-datum-output.body

cardano-cli transaction sign \
  --tx-body-file create-datum-output.body \
  --testnet-magic 42 \
  --signing-key-file $UTXO_SKEY \
  --out-file create-datum-output.tx
```

Spending ADA at the script address

Now that there is ADA at our script address, we must construct the appropriate transaction in order to spend it.

`$plutusutxotxin` - This is the tx input that sits at the Plutus script address (NB: It has a datum hash). `$plutusrequiredtime` and `$plutusrequiredspace` - These make up the Plutus script execution budget and are part of the `$txfee` `tx-in-redeemer-value` - We must also supply a redeemer value even though the Plutus script will succeed regardless of the redeemer.

```
cardano-cli transaction build-raw \
  --alonzo-era \
  --fee "$txfee" \
  --tx-in $plutusutxotxin \
  --tx-in-collateral $txinCollateral \
  --tx-out "$dummyaddress+$spendable" \
  --tx-in-script-file $plutusscriptinuse \
  --tx-in-datum-value 42  \
  --protocol-params-file pparams.json\
  --tx-in-redeemer-value 42 \
  --tx-in-execution-units "($plutusrequiredtime, $plutusrequiredspace)" \
  --out-file test-alonzo.body

cardano-cli transaction sign \
  --tx-body-file test-alonzo.body \
  --testnet-magic 42 \
  --signing-key-file "${UTXO_SKEY}" \
  --out-file alonzo.tx
```

If there is ADA at `$dummyaddress` then the Plutus script was successfully executed. Conversely, if the Plutus script failed, the collateral input would have been consumed.

You can use the example-txin-locking-plutus-script.sh in conjunction with mkfiles.sh alonzo script to automagically run the `AlwaysSucceeds` script.

5.4 Troubleshooting

5.4.1 Error codes

To reduce code size, on-chain errors only output codes. Here's what they mean:

- Ledger errors
  - `L0: Input constraint`
  - `L1: Output constraint`
  - `L2: Missing datum`
  - `L3: Wrong validation interval`
  - `L4: Missing signature`
  - `L5: Spent value not OK`
  - `L6: Produced value not OK`
  - `L7: Public key output not spent`
  - `L8: Script output not spent`

  - `L9: Value minted not OK`
  - `La: MustPayToPubKey`
  - `Lb: MustPayToOtherScript`
  - `Lc: MustHashDatum`
  - `Ld: checkScriptContext failed`
  - `Le: Can't find any continuing outputs`
  - `Lf: Can't get any continuing outputs`
  - `Lg: Can't get validator and datum hashes`
  - `Lh: Can't get currency symbol of the current validator script`
  - `Li: DecodingError`
- State machine errors
  - `S0: Can't find validation input`
  - `S1: State transition invalid - checks failed`
  - `S2: Thread token not found`
  - `S3: Non-zero value allocated in final state`
  - `S4: State transition invalid - constraints not satisfied by ScriptContext`
  - `S5: State transition invalid - constraints not satisfied by ScriptContext`
  - `S6: State transition invalid - input is not a valid transition at the current state`
  - `S7: Value minted different from expected`
  - `S8: Pending transaction does not spend the designated transaction output`
- Currency errors
  - `C0: Value minted different from expected`
  - `C1: Pending transaction does not spend the designated transaction output`

5.5 Architectural Decision Records

We document our architectural and design decisions for all of our components. In order to do that, there is practice called architectural decision records ("ADR"), that we can integrate into our workflow. An architectural decision record (ADR) is a document that captures an important architectural decision made along with its context and consequences.

The goals are:

- making decisions transparent to internal/external stakeholders and contributors.
- getting feeback on decisions that we're about to make or have made
- providing external contributors a framework to propose architectural changes

- providing a big picture of all major decisions that were made

The general process for creating an ADR is:

1. cloning the repository
2. creating a new file with the format *<ADR_NUMBER>-<TITLE>.rst* in the directory *doc/adr*
3. adding the ADR in the table of content tree of the Readthedocs
4. committing and pushing to the repository

5.5.1 ADR 1: Record architectural decisions

Date: 2022-06-08

Authors

koslambrou <konstantinos.lambrou@iohk.io>

Status

Accepted

Context

We are in search for a means to document our architectural and design decisions for all of our components. In order to do that, there is practice called architectural decision records ("ADR"), that we can integrate into our workflow.

This does not replace actual architecture documentation, but provides people who are contributing:

- the means to understand architectural and design decisions that were made
- a framework for proposing changes to the current architecture

For each decision, it is important to consider the following factors:

- what we have decided to do
- why we have made this decision
- what we expect the impact of this decision to be
- what we have learned in the process

As we're already using rST, Sphinxdoc and readthedocs, it would be practical to integrate these ADRs as part of our current documentation infrastructure.

Decision

- We will use ADRs to document, propose and discuss any important or significant architectural and design decisions.
- The ADR format will follow the format described in *Implications* section.
- We will follow the convention of storing those ADRs as rST or Markdown formatted documents stored under the *docs/adr* directory, as exemplified in Nat Pryce's adr-tools. This does not imply that we will be using *adr-tools* itself, as we might diverge from the proposed structure.
- We will keep rejected ADRs
- We will strive, if possible, to create an ADR as early as possible in relation to the actual implementation.

Implications

ADRs should be written using the template described in the *ADR template* which comes from Chapter 6.5.2 (*A Template for Documenting Architectural Decisions*) of *Documenting Software Architectures: Views and Beyond (2nd Edition)*.

However, the mandatory sections are *Title*, *Status*, *Issue/Context*, *Decision*, *Implications/Consequences*. The rest are optional.

Another good reference is the article Architecture Decision Records by Michael Nygard (Nov. 15, 2011).

ADR template

What follows is the ADR format (adapted from the book).

Section	Description
Title	These documents have names that are short noun phrases. For example, "ADR 1: Deployment on Ruby on Rails 3.0.10" or "ADR 9: LDAP for Multitenant Integration"
Authors	List each author's name and email.
Status	State the status of the decision, such as "draft" if the decision is still being written, as "proposed" if the project stakeholders haven't agreed with it yet, "accepted" once it is agreed. If a later ADR changes or reverses a decision, it may be marked as "deprecated" or "superseded" with a reference to its replacement. (This is not the status of implementing the decision.)
Issue (or context)	This section describes the architectural design issue being addressed. This description should leave no questions as to why this issue needs to be addressed now. The language in this section is value-neutral. It is simply describing facts.
Decision	Clearly state the solution chosen. It is the selection of one of the positions that the architect could have taken. It is stated in full sentences, with active voice. "We will ..."
Tags	Add one or more tags to the decision. Useful for organizing the set of decision.
Assumptions	Clearly describe the underlying assumptions in the environment in which a decision is being made. These could be cost, schedule, technology, and so on. Note that constraints in the environment (such as a list of accepted technology standards, an enterprise architecture, or commonly employed patterns) may limit the set of alternatives considered.
Argument	Outline why a position was selected. This is probably as important as the decision itself. The argument for a decision can include items such as implementation cost, total cost of ownership, time to market, and availability of required development resources.
Alternatives	List alternatives (that is, options or positions) considered. Explain alternatives with sufficient detail to judge their suitability; refer to external documentation to do so if necessary. Only viable positions should be described here. While you don't need an exhaustive list, you also don't want to hear the question "Did you think about... ?" during a final review, which might lead to a loss of credibility and a questioning of other architectural decisions. Listing alternatives espoused by others also helps them know that their opinions were heard. Finally, listing alternatives helps the architect make the right decision, because listing alternatives cannot be done unless those alternatives were given due consideration.
Implications (or consequences)	Describe the decision's implications. For example, it may • Introduce a need to make other decisions • Create new requirements • Modify existing requirements • Pose additional constraints to the environment • Require renegotiation of scope • Require renegotiation of the schedule with the customers

5.5.2 ADR 2: Repository Standardization

Date: 2022-07-06

Authors

Lorenzo Calegari <lorenzo.calegari@iohk.io>

Status

Draft

Context

IOG is undertaking a company-wide effort to restructure and standardize its repositories, favoring mono-repos and enforcing shared GitOps and DevOps processes. Parallel to this, a new CI infrastructure is being developed.

Examples of this are:

- input-output-hk/cardano-world
- input-output-hk/ci-world
- input-output-hk/atala-world

This initiative appears to be championed by the SRE team who are the creators of divnix/std. Indeed *std* is at the heart of the standardization dream.

Decision

- Standardization of the repositories has been deemed a worthwhile endeavour, though of very low priority.
- Phase 1 of the standardization process will be carried out in parallel with *Move Marconi to a separate repository*. A separate repository will be created for Marconi, and from the very beginning it will use *std*. This way the benefits, limitations and integration costs of *std* can be experienced and measured, and an informed, definitive decision on standardizing *plutus-core* and *plutus-apps* themselves can be made.

Argument

In short, *std* aims to answer the one critical question that pops in the mind of newcomers and veterans alike:

What can I do with this repository?

In practice, *std* is flake-based nix library code that provides a strongly-but-sensibly-opinionated top-level interface for structuring all your nix code.

This is wonderful news for the owner of the repository's nix code, but what about every other stakeholder? Especially developers who don't care/know about nix?

Contributors of a standardized codebase will be gifted with a TUI to discover and interact with the repository, which is probably something that is long overdue as an industry-level best-practice.

Who wouldn't want to clone a repository, type *std* and be presented with a TUI that gives you an interactive tour of the repository's artifacts, together with a list of all possible DevOps and GitOps actions (build, test, develop, run, deploy, benchmark, publish, package, monitor, . . .) in addition to any other action that you may define.

And for power users and automators, there is an equivalent CLI to the TUI. This makes *README* files obsolete to an extent. A TUI/CLI combo represents the best conceivable solution in terms of user experience (only a GUI could top that perhaps).

In conclusion, the advantages of standardizing the repositories are:

- Enforce a shared mental model for internal and external teams to effortlessly reason about the codebase.
- Provide a TUI/CLI to more easily discover, interact with, and contribute to the repository, with the goal to provide a superior user experience to all stakeholders.
- Refactor all existing Nix code into a supposedly far better structure. *std* seems to solve the "import problem" by automatically parsing the directory structure and threading all derivations into a globally accessible top-level scope, drastically reducing the average length of paths in the dependency graph, both at the file level and at the term/variable level. This all translates into cleaner, more maintainable code.

Implications

The plutus repositories now exhibit a large amount of duplicated nix (and configuration) code, as a result of the split into *plutus-core* and *plutus-apps*.

While introducing *std* will not in itself help reduce duplication, the refactoring process will involve identifying and isolating shared components that can be later packaged and separated into library code.

The goal is to standardize both repositories, by introducing *std* and refactoring all existing nix code accordingly.

The SRE team has also created several other satellite repositories containing reusable nix code to support this process, though it is unclear at this stage whether these are relevant to standardizing *plutus-core* and *plutus-apps*.

Such repositories include:

- nixagii
- devshell-capsules
- kladoi

The standardization process would follow the 4 Layers SRE Mental Model, which begins by introducing *std* in Layer 1 (binary packaging). Layers 2-3-4 (which is mostly DevOps) will be postponed to a later date, once the migration to the new CI systems has been officially approved and initiated.

5.5.3 ADR 3: Move Marconi into a separate repository

Date: 2022-06-08

Authors

Lorenzo Calegari <lorenzo.calegari@iohk.io>

Status

Draft

Context

Marconi is a Haskell executable and library that lives in *plutus-chain-index*.

It is desirable to move it into a separate repository for the following reasons:

- Better visibility and easier to discover
- It wants to update the version of its *cardano-api* dependency independently of the version used by *plutus-apps*
- It is a farily independent component, therefore it warrants its own repository

However, creating a separate repository would be rather costly. It would involve a great deal of duplication, due to the way our current nix code is structured, not to mention the added complexity and overhead inherent in maintaining a separate codebase.

Decision

- We will put Marconi in a separate Github repository
- Until we resolve the issues with creating a separate Github repository (see Context), we will keep Marconi as a separate project in *plutus-apps*

Implications

- A nix flake will be added in *plutus-apps* so that users will be able to obtain the Marconi executable trivially
- The possibility to specify a separate version of *cardano-api* just for Marconi, **while staying in plutus-apps**, will be explored
- As a very low priority task, a new repository *will* be created for Marconi, which will use *std* from the start (see *Repository Standardization*)

Related Decisions

Repository Standardization

5.5.4 ADR 4: Making a case for Marconi

Date: 2022-07-26

Author(s)

Radu Ometita <radu.ometita@iohk.io>

Status

Proposed

Context

Plutus off-chain code oftentimes needs access to indexed portions of the blockchain. The plutus-chain-index project is the initial solution meant to deliver access to this kind of data. However, after release, a couple of shortcomings were identified which prompted the development of an indexing solution that is based on a different set of architectural and functional constraints.

A lot of the shortcomings are connected to the exploratory type of development that we used to deliver the plutus-chain-index which was prompted by the lack of a clear specification and a lack of concern for non-functional and quality assurance requirements. The top-down design resulted in a monolithic and fairly complex architecture which made the code difficult to reuse, compose and understand.

Some of the problems we identified due to the above-mentioned approaches are:

A. The use of an effect system (the *freer-simple* package) makes the code fairly complex and difficult to understand (quite a few type-level computations are happening). The separation between syntax and semantics imposed by the library also complicates matters for no clear reason (for example, if we write two semantics, one for pure code used for testing and one for production code, then there would be a lot of production code that would not be tested).

B. We cannot customise the indexed set of data, the plutus-chain-index provides only all-or-nothing indexing. While this can be addressed, the architecture makes it an uphill battle.

C. The implicit assumption that there is only one index running caused issues when we made the Plutus Application Backend collect and index information requested by smart contracts. Now we have two components that index information from the blockchain, but they are not synchronised. Querying the plutus-chain-index about transactions received from the Plutus Application Backend may result in no data returned, since the plutus-chain-index indexes data slower than the PAB.

D. The lack of non-functional requirements resulted in software that uses an unreasonable amount of resources and results in slow synchronisation speeds. And since everything is monolithic it is difficult to turn off indexing of data which is not required by our customers there is no way to limit the required resources.

E. The same lack of a specification and non-functional requirements makes the testing feel ad-hoc and like an afterthought.

The Chain Index was meant to be a software application that supports the execution of smart contracts. And, in that, it succeeded. However, we found that our customers would rather have a library of functionality that they can customize to do the following:

- to build their own indexers,
- to work only with the data that they care about for their application,
- to use whatever storage engine they prefer, and
- to support only the queries that they need to support.

So when we took all the feedback into account we decided that a redesign of the indexing solution using a much simplified and modular design is a worthwhile enterprise.

We continue by introducing some of the design principles that guided us in the specification of Marconi.

Design principles of Marconi

We follow the Algebra Driven Design approach for Marconi components, so from the get-go, we will have a checked specification for the software that we develop.

The specification is based on a simplified model which should help with documenting how everything works without getting into the more complex details.

Having a set of property-based tests to validate that the implementation conforms to the specification also means that the correctness of the implementation does not rely on type-level checks or complicated term-level machinery (we could even verify the correctness of a Rust implementation by leveraging the Rust to Haskell FFI).

Because we have no reliance on type-level checks or complicated architectural patterns to validate the software (we use the specification and property tests for that), the code is much easier to understand, document and extend.

Indexing solution

The indexing solution has the following basic requirements: it needs to deal with rollbacks as elegantly as possible and provide a way to compromise between memory, disk and CPU usage.

On the Cardano blockchain, there are frequent rollbacks, but they can only span a maximum of 2160 blocks (and most of them are < 10 blocks). We call the 2160 number the security parameter K (and we denote it by 'K' henceforth).

Indexers are a store which is updated by events created from each block. The problem introduced by rollbacks is that we need to undo all state changes when a rollback occurs.

We opted for a design where we keep K blocks in memory as the list of events that are fed into the function that stores them once they go beyond the K limit.

This architectural decision has some desirable effects:

1. Managing rollbacks is very simple and fast. We drop the events that were rolled back. (No need to undo the application of blocks on the state stored on disk, which would be necessary if we were to store everything on disk as fast as possible).
2. Making 'K' configurable makes the design already quite scalable. Developers do not usually need to guard themselves against rollbacks by K blocks so they can choose to store 10 events in memory allowing for chain desynchronisation in the unlikely event that a rollback occurs beyond the 10 blocks limit.
3. In case of a restart recovery is very simple. If the selected K parameter is properly set, we store only fully confirmed transactions so there is nothing to do other than resume operation.

And some less desirable effects:

1. We must keep K events in memory, which (depending on how large events are) can waste some memory. Our educated guess is that this is a reasonable compromise, but depending on how large events can get that may not be the case for your use case.
2. Queries are more involved as we need to scan events in memory and the state persisted on disk.

Query and storage

The indexed data is accessible through queries. There are no constraints on the format of queries or results. Both are identified by a type variable that the indexer exposes and the implementation of the result and query datatypes and the store and query functions can be provided by the user. One of the complications of this query implementation is that a query has to run on the merged data from memory and disk.

The possibility of defining the query and store functions allows us to associate any kind of storage type to the indexers, though, right now we are only using SQLite.

Identification of events

We need a way to provide an answer to the question: How much of the stream has been consumed by the indexer? We choose to do that by associating a sequence number to incoming blocks, and carrying it along the stream of events. Having a way to answer this question is connected to the following features which we plan to implement:

1. Synchronisation of multiple indexers (queries have a validity interval)
2. Resume functionality (we need to know from which slot to resume)
3. Handling of rollbacks (now there is explicit handling of rollbacks)

More information will become available in the next few sprints.

Event streams

To support PAB functionality which subscribes to a source for a set of event types, we need a way to produce events from indexers.

They are also very useful for contracts that want to track rollbacks. Rollbacks are invisible from the point of view of the indexed data, but it may be the case that the internal state of a contract needs to know that the state has been reverted.

5.5.5 ADR 5: PAB and indexing component integration

Date: 2022-07-27

Author(s)

koslambrou <konstantinos.lambrou@iohk.io>

Status

Proposed

Context

Let's start with the problematic example (copy-paste of the current *PubKey* contract in *plutus-use-cases*).

```
-- | Lock some funds in a 'PayToPubKey' contract, returning the output's address
--   and a 'TxIn' transaction input that can spend it.
pubKeyContract
    :: forall w s e.
    ( AsPubKeyError e
    )
    => PaymentPubKeyHash
    -> Value
    -> Contract w s e (TxOutRef, Maybe ChainIndexTxOut, TypedValidator PubKeyContract)
pubKeyContract pk vl = mapError (review _PubKeyError   ) $ do
    -- Step 1
    let inst = typedValidator pk
        address = Scripts.validatorAddress inst
        tx = Constraints.mustPayToTheScriptWithDatumHash () vl
    ledgerTx <- mkTxConstraints (Constraints.typedValidatorLookups inst) tx
                >>= submitUnbalancedTx . Constraints.adjustUnbalancedTx

    -- Step 2
    _ <- awaitTxConfirmed (getCardanoTxId ledgerTx)

    -- Step 3
    let refs = Map.keys
               $ Map.filter ((==) address . txOutAddress)
               $ getCardanoTxProducedOutputs ledgerTx
    case refs of
        []                  -> throwing _ScriptOutputMissing pk
        [outRef] -> do
            -- Step 4
            ciTxOut <- unspentTxOutFromRef outRef
            pure (outRef, ciTxOut, inst)
        _                   -> throwing _MultipleScriptOutputs pk
```

Here's an outline of the contract's steps:

1. Creates a transaction and submits it to the node
2. Waits for transaction to be confirmed
3. Finds the first UTXO of that transaction (return type *TxOutRef*)
4. Queries the plutus-chain-index to get the *ChainIndexTxOut* out of that *TxOutRef*

The problem is that the *ciTxOut* variable in step 4 will almost always result in *Nothing*.

Why? Here's some context.

The PAB listens to the local node and stores blockchain information in memory such as the status of transactions, the status of transaction outputs, the last synced slot, the current slot, etc., in a variable of type *BlockchainEnv*. The *awaitTxConfirmed* is actually querying the state of *BlockchainEnv* and waits until the status of the transaction transitions to *Confirmed*.

Meanwhile, plutus-chain-index (our main indexing component at the time of this writing) is also listening to incoming blocks from the local node and indexes them into a database. The indexed data can be queried using the REST API interface.

This brings up the main issue: the PAB and plutus-chain-index each listen to the same source of information (a local Cardano node), but each index the information at different speeds. For a dApp developer writing off-chain code using

the Contract API, there is no abstraction for handling multiple sources of truth.

Currently, in the best case scenario (fully synced PAB and plutus-chain-index), plutus-chain-index will always trail behind the in-memory storage of the PAB by a few seconds. Therefore, even in this scenario, querying the plutus-chain-index with *unspentTxOutFromRef* in the above contract has a high probability of returning *Nothing*.

Decisions

The best solution is probably a combination of the *Alternative solutions* described below. However, we will mainly choose the *Query functions should interact with a single source of truth* solution.

- We will replace *plutus-chain-index* with *Marconi* as PAB's indexing component
- We will move out the blockchain information indexed by PAB in Marconi
- We will add new indexers in Marconi in order to replicate the information indexed by *plutus-chain-index*
- We will adapt the architecture of Marconi (which will become our new indexing component) to support waiting queries
- Since we suppose that indexing component should be in the same machine as the PAB, then we will use Marconi as a library to index and query the indexed blockchain information without relying on an HTTP API

Alternative solutions

In this section, we describe all the ways to deal with the problem. Note that final decision might include one or more of these alternate solutions.

Make sure all components are in sync

We can make sure that all indexing components are syncing at the same speed. For example, if PAB syncs from the local node and arrives at slot 100, then it needs to wait for the chain-index to also arrive at slot 100. Only then can it respond to a Contract request.

A simply way to achieve this behavior is to change the implementation of Contract API handler functions (like *utxosAt*, *unspentTxOutFromRef*, etc.) by waiting for the component to be in sync with all other components. For example, let's take the *utxosAt* request which basically queries UTXOs from a given address using *plutus-chain-index*. We could, before querying, wait for *plutus-chain-index* to be in sync with all components which index blockchain data (PAB, Blockfrost).

Pros:

- All the indexed information is consistent with each other

Cons:

- As fast as the slowest indexing component
- Tight coupling between the indexing components meaning that if the Contract only uses chain-index requests without using requests from other indexing components, the chain-index will still have to wait for all other components to be in sync with each other

Add indexing specific functions in the Contract API

In this scenario, we would need to split Contract API requests which interact with an external indexing component to the ones that use the PAB. Currently, we have *awaitTxConfirmed* which uses the indexed information in the PAB to wait for a transaction status to change to *Confirmed*. On top of that, we can have *awaitTxIndexed* or *awaitTxOutIndexed* which will wait for the information to be indexed in the external indexing component.

Pros:

- Limits design change on the PAB
- More control given to the user of the Contract API

Cons:

- Adds an undesired complexity to the Contract API
- We'll need to add a bunch of functions (e.g., *currentNodeSlot*, *currentMarconiSlot*, *awaitMarconiTxConfirmed,* `*awaitScrollsTxConfirmed*, etc.) for each new indexing component we want to support

Query functions should interact with a single source of truth

In this scenario, we make the design decision that the Contract API should only interact with a single indexing component. Thus, any blockchain information currently stored in the PAB should be moved to the indexing component. Also, combining indexing components would need to be integrated in the single indexing solution that's connected to the PAB.

Pros:

- Simplest in design to implement (other than manual work to move code)
- No modification to the Contract API
- Augments PAB's cohesion, because it's responsability will be limited to contract instance management

Cons:

- The design of the indexing component will need to be changed to support waiting queries (like the *awaitTxConfirmed* from PAB)
- Still under the assumption that the indexing component is in sync with the PAB in order to use some querying functions

Implications

Having a single source of truth indexer will augment the PAB's cohesion and greatly simpy the *plutus-pab* codebase. Also, we will not encounter the situation where we have multiple indexers that index at different speeds.

However, there is still the problem that the indexing solution might not be in sync with the local node. Therefore, we need to make another decision on how to deal with it.

Notes

This problem manifested itself in the Github issue #473 and there was a temporary fix in the PR #496. However, the proper solution to the issue would be the implementation of this ADR.

This ADR has been discussed here: #550.

5.5.6 ADR 6: Common Contract API

Date: 2022-07-12

Authors

Gergely Szabo <gergely@mlabs.city>

koslambrou <konstantinos.lambrou@iohk.io>

Status

Proposed

Context

There are multiple implementations of a Plutus Application Backend (PAB) external of IO Global, and also other tools related to Plutus smart contracts. Some of them are using the same contract interface as the official implementation, but some of them use a different interface. However, as the ecosystem evolves, it would be beneficial to create a well defined standard, that other off-chain tools can use as a reference, or as an interface to implement.

Currently, as we are getting close to the Vasil hardfork, testing tools and Plutus Application backend tools are at a hurry to update their dependencies and get to a Vasil compliant/compatible state. However, tools that are depending on *plutus-apps* are blocked by the PAB development. This initiative was born out of this context, but could solve other problems as well.

The Contract API (defined in *plutus-apps/plutus-contract*) is using the *freer-simple* effect system to define all the contract effects. This already allows us to separate the interface from the implementation, and to have multiple implementations/interpreters for one interface. Currently, there are two implementations for the Contract API:

- one for the plutus-apps emulator (inside *plutus-apps/plutus-contract*)
- one for plutus-apps' Plutus Application Backend (inside *plutus-apps/plutus-pab*)

Therefore, we can leverage this separatation of interface and implementation in order to move the interface out of *plutus-apps*.

Decision

- We will split the *plutus-apps/plutus-contract* package into two parts: the Contract API (*plutus-contract*) and the emulator (*plutus-contract-emulator*).
- We will create effects for the constraints-based transaction builder library (*plutus-apps/plutus-ledger-constraints*) in the Contract API. Currently, the interface and the implementation in the transaction builder library are tightly coupled. Therefore, we need to decouple them.

- We will create a separate repository with the contract effects and types (the splitted *plutus-contract*). By moving the Contract API out of the plutus-apps monorepository, any tool could update to newer version to their discretion. Without many dependencies, many tools could utilize the Contract API without having to depend on the whole plutus-apps monorepo.
- We (the Plutus Tools at IO Global) will continue to be the main maintainers of this new repository. However, a new ADR will need to be created if we ever decide to make this a community driven project.
- TODO: What about governance? How do we decide which interface changes are accepted? ADRs? Who ultimately accepts and rejects them?

Argument

We speed up the development of off-chain tools, by loosening up some of tightly coupled dependencies, so these external projects can move more freely. This would also mean that the cost of the interface update would be reduced, so we could see more features added to the standard, and the PAB API following the capabilities of Cardano more closely. As an added benefit, community involvement with the API could also greatly improve.

A standard API for all Plutus contacts would help keeping the ecosystem on the same track with their implementation. As more and more off-chain tools implement the same contract interface in the future, it will be relatively easy to switch between different Plutus Application Backend implementations, or to use multiple of these tools at the same time without a need for serious code rewrites.

The implementation of the Contract API interface would track a specific version of the Contract API interface. We would then need to regularly update the implementation given any interface changes.

Implications

- We will need to decide if we should make this a community driven project. If so, we will also need to make a decision about governance. How do we decide which interface changes are accepted? Do we use ADRs? Who ultimately accepts and rejects them?

Notes

This ADR has been discussed here: #586.

5.5.7 ADR 7: Support reference inputs in constraint library

Date: 2022-08-09

Author(s)

koslambrou <konstantinos.lambrou@iohk.io>

Status

Accepted

Context

After the Vasil HF, the Cardano blockchain will support *reference* inputs by adding a new field in the transaction data type. With reference inputs, transactions can take a look at UTXOs without actually spending them.

Thus, we need to adapt our transaction constraint data type (`TxConstraints`) to support referencing UTXOs.

Decision

- We will add the data constuctor `MustReferenceOutput TxOutRef` to the `TxConstraints` data type.
- The PlutusV1 on-chain implementation of this new constraint will simply return `False`. However, *cardano-ledger* throws a phase-1 validation error if transactions that use the some of the new features (reference inputs, inline datums and reference scripts) try to execute PlutusV1 scripts. See the Babbage era ledger specification. Therefore, the only way to get a phase-2 validation error would be to use this constraint on-chain in a PlutusV1 script, without using any of the new Babbage era features off-chain.
- The PlutusV2 on-chain implementation of this new constraint will check that the provided `TxOutRef` is part of the `ScriptContext`'s reference inputs.

Argument

At first glance, we might think that we need two data constructors for reference inputs such as `MustReferencePubKeyOutput` and `MustReferenceScriptOutput` in contrast to the existing `MustSpendPubKeyOutput` and `MustSpendScriptOutput` constraints. However, we do not need to make the distinction between public key outputs and script outputs because we're not spending the output, therefore, we don't need to provide a redeemer nor the actual script as a witness to the transaction input.

Notes

This ADR has been addressed in the PRs #640 and #661.

5.5.8 ADR 8: Support inline datums in constraint library

Date: 2022-08-14

Author(s)

koslambrou <konstantinos.lambrou@iohk.io>

Status

Proposed

Context

In Babbage era (available after the Vasil HF), the Cardano blockchain will support *inline* datums by changing the `TxOut` data type.

In Alonzo era, a `TxOut` was able to store arbitrary data called the datum. However, only the hash of the datum was stored, not the actual datum.

With inline datums available in Babbage era, transaction outputs can either contain the hash of the datum or the actual datum. Thus, we need to adapt our transaction constraint data type (`TxConstraints`) to support this new feature.

Decision

- We will replace the `Datum` parameter in `TxConstraints`'s data constructor `MustPayToPubKeyAddress` with `Plutus.V2.Ledger.Api.OutputDatum`. In the offchain implementation of the constraint, we will use this new data constructor parameter to support either adding the datum in the datum witness set (by using the datum lookups to resolve the hash) or inline it in the transaction output. In the PlutusV1 on-chain implementation of the constraint, we will return `False` if the datum value matches `OutputDatum Datum` because the ledger forbids using Babbage era features with PlutusV1. The PlutusV2 on-chain implementation of the constraint is trivial.
- We will modify the data constructor interface, on-chain implementation and off-chain implementation of `MustPayToOtherScript` similarly to `MustPayToPubKeyAddress`.
- We will modify the off-chain implementation of the data constructor `MustSpendScriptOutput` in order to support inline datums. Currently, the script output's datum is added in the transaction's datum witness set. However, if the datum is inlined in the script output, then it is already witnessed. Therefore, we don't need to add it in the datum witness set.

Argument

The main decision was to find out which data type will replace `Datum` in the interface of `MustPayToPubKeyAddress` and `MustPayToOtherScript`. The decision to use `Plutus.V2.Ledger.Api.OutputDatum` was mainly because of the constraint library's main design: the parameters of `TxConstraints`'s data constructor must work with the on-chain as well as the off-chain implementation. Therefore, we decided to use `OutputDatum` which we know works in on-chain code because this type is used in `Plutus.V2.Ledger.Api.ScriptContext`.

Notes

5.5.9 ADR 9: Support reference scripts in constraint library

Date: 2022-08-15

Author(s)

koslambrou <konstantinos.lambrou@iohk.io>

Status

Accepted

Context

In Babbage era (available after the Vasil HF), the Cardano blockchain will support "reference scripts" by changing the `TxOut` data type. Reference scripts are used to attach arbitrary scripts to transaction outputs and are used to satisfy script requirements during validation, rather than requiring the spending transaction to do so. Thus, we need to adapt our transaction constraint data type (`TxConstraints`) to support this new feature.

Decision

- We will add `Maybe ScriptHash` as a new data constructor parameter for the constraints `MustPayToPubKeyAddress`, `MustPayToOtherScript`, `ScriptOutputConstraint` in `TxConstraints`. In the off-chain implementation of those constraints, if a reference script hash is provided, we will need to find the actual script in the lookups table so that we can include it in the transaction output. In the PlutusV1 on-chain implementation of the constraint, we will return `False` if a reference script is provided because the ledger forbids using Babbage era features with PlutusV1. The PlutusV2 on-chain implementation of the constraint is trivial.
- We will modify the off-chain implementation of `MustSpendScriptOutput` and `ScriptInputConstraint` in order to add support for witnessing a script by actually providing it, or by pointing to the reference input which contains the script.

Argument

The main decision was to find out which data type will represent reference scripts. Similarly to *ADR 8: Support inline datums in constraint library*, the decision to use `Maybe ScriptHash` was mainly because of the constraint library's main design that data types need to work with on-chain as well as off-chain implementation.

Notes

This ADR has been addressed in PR #662, #666 and #678

5.5.10 ADR 10: Rolling back on-disk data for Marconi indexers

Date: 2022-08-05

Author(s)

Ometita Radu <radu.ometita@iohk.io>

Status

Draft

Context

Off-chain code needs access to indexed portions of the blockchain. Currently, we have a working solution in the form of the chain-index and PAB (which both index information). The big problem with the current solution is its lack of reuse (or modularity) capability.

We attempt to fix that problem with Marconi where we currently have a generic indexer that stores volatile information in memory and blocks that have been fully committed (are old enough to guarantee that they will not be rolled back) on disk. The K blockchain parameter represents how many blocks deep the blockchain becomes immutable (no rollbacks can occur beyond K blocks).

We currently need to keep K blocks in memory to be able to perform rollbacks. However, the K parameter can be adjusted for indexers which land us in the unfortunate position of saying that there may be data corruption in the case where the number of rollbacked blocks is larger than the number of blocks stored in memory. While this is both detectable and unlikely to happen we think that our current solution can prevent it without any significant drawbacks.

Decision

After receiving feedback on the initial implementation of the PAB and chain-index we needed a generic way of indexing information where we can control the amount of memory the indexer uses. The first version of indexers store the volatile blocks in memory and persist them to disk whenever they become older than the K parameter.

We make a distinction between the volatile blocks which are stored in memory as events (and are derived from blocks). We fold these events into the aggregated on-disk data structure for which we do not require to keep multiple versions (rollbacks cannot happen for this data structure).

We improve on that idea by allowing part of the volatile blocks to be stored on disk. While this is not required at the API level, the usage pattern would be to have a set of events, as well as the aggregated data structure stored on disk. The compromise here is that the more data is on the disk, the more we will need to work with the disk and the slower the indexing process will become. The advantage would be reduced memory usage.

Events

Events need to contain information about the slot number and block id when they were produced.

The slot number information is used in case of rollbacks (when we only get the old slot number that we need to rollback to) and for resuming the operation of the index, in which case we need both the slot numbers and block ids.

Note that to support resume from disk we need to always have at least one event persisted on disk which contains the slot number and block id from which we are supposed to resume operation. In case there are more than one events stored on disk we can use those as resume alternatives in the case of the ChainSync protocol.

Queries

We also extend the queries with a bit more structure that will make specifying query validity intervals possible. The validity is important in the case where we want to query several indexers and we would like them to require to have processed all the information up to some slot number or be between some slot interval.

At this point all query results are synchronous. We have plans to extend this functionality, but these plans are based on updating the notification system for indexers which will be described in a further ADR.

API Design

We want the API for our users to be as flexible as possible, so some of our previously mentioned design patterns are not captured by the API, but rather by its implementation.

Data types

- The *Events* data type contains the following fields:
  - Slot numbers (a data type that supports ordering)
  - Block id (a data type that supports equality checking)
  - e (type variable standing for the event)
- The *Query* data type contains the following fields:
  - Validity interval (can be any interval defined by using the slot numbers or a special value that turn off checking for validity)
  - q (type variable standing for the query)
- Result
  - Slot number at which the query was ran
  - r (type variable standing for the query result)

Functions

- The *Query* function takes the following parameters:
  - Indexer - The indexer that we are using to run the query.
  - Validity interval - The interval under which the query needs to be ran.
  - The query (q type variable) - The user-defined query.
  - The query result
- The *Store* function takes the following parameters:
  - Indexer - The indexer for which we run the function
  - Does not return anything useful
- The *Resume* function takes the following parameters:
  - Indexer - The indexer we need to query for the last consumed slot numbers
  - Returns a list of slot numbers and block ids

Runtime parameters

- Minimum events retained (this should be the previously mentioned K parameter)

  Currently the main cardano network guarantees that there will be no rollbacks beyond 2160 blocks. This would be that parameter.

- Maximum in-memory events (should be less than K)

  How many events do we want to keep in memory. For the main network this should be any number lower that 2160. The larger the number the less frequent writing to disk is and the more RAM is used.

Extension mechanisms

A. Storage engine

You can customise the query and store functions which run in some generic monad to use whatever backend is best for the job. We currently use SQLite, but that is more for convenience than anything else.

B. Query intervals

If you want to specify an interval for your queries (which is highly encouraged) then you need to have in memory (or on disk) sufficient information to reconstruct the state at the given slot number. The information required is contained in the event (which includes the slot number). By storing more than K events you can extend the query interval as much as you need. In extreme, you can store events without ever aggregating and deleting them, in which case your queries can span the whole blockchain.

Implementation

This is the way we suggest people implement storage for the indexers:

```
| Memory |       Disk          |
|--------|--------|------------|
| Events | Events | Aggregate |
```

To support the resume function we need to always have at least one event stored on disk. This is an invariant that an implementation can keep by ensuring that the number of in-memory events is less than K.

Since the number of events stored in memory is constant we can keep on using a ring buffer backed by the vector library.

Events are moved into storage whenever the in-memory buffer becomes full. When they are moved into storage we also need to decide what we are folding into the stored aggregated data structure. We should never fold any events that are newer than K blocks.

We suggest using type families for the implementation due to the functional dependencies between the handler type and the monad that the indexer runs in, as well as the dependency between the query type and the result type (and in the future the notification type).

5.5.11 ADR 11: Support return and total collateral when building transactions

Date: 2022-08-30

Author(s)

koslambrou <konstantinos.lambrou@iohk.io>

Status

Proposed

Context

In Babbage era (available after the Vasil HF), Cardano transactions will contain new collateral related fields: "return collateral" and "total collateral" collateral. Return collateral (also called "collateral output") and total collateral are detailed in CIP-40.

In summary, return collateral is a special output (basically of type `TxOut`) that becomes available in case there is a failed phase-2 validation. In addition, we have the new total collateral field which explicitly says how much collateral (in lovelace) is going to be actually consumed in the case of phase-2 validation failure.

Decision

- We will add the `txReturnCollateral` and the `txTotalCollateral` fields in the *Ledger.Tx.Internal.Tx* data type.
- We will modify the `Wallet.Emulator.Wallet.handleBalance` function in *plutus-contract* to set the correct return and total collateral for an `UnbalancedTx` (of type `Either CardanoBuildTx EmulatorTx`). In either type of transaction, we would compute the `txTotalCollateral` while estimating the fee with the formula `quot (txfee txb * (collateralPercent pp)) * 100` and then set `txReturnCollateral` with the formula `sum collateralInputs - txTotalCollateral`.

Argument

As the user would want to pay the least amount of collateral, we made the decision to modify the balancing algorithm to automatically set the return collateral to the highest possible value.

Alternatives

The main alternative would have been to add a new constraint such as `MustReturnCollateral TxOut` in the constraints library to allow the users to specify the return collateral themselves. However, as explained in the *Argument* section, users would always want to pay the least amount of collateral. Therefore we don't expect the need to set the return collateral manually.

Notes

5.5.12 ADR 12: Commit to data types in cardano-api

Date: 2022-10-03

Author(s)

koslambrou <konstantinos.lambrou@iohk.io>

Status

Draft

Context

Since the genesis of the `plutus-apps` repository, the components have been historically using the types in the `plutus-ledger-api` package (which is now part of the `plutus` repository) in the off-chain part of a Plutus application.

This was desirable in order to start designing a way to build Plutus applications before the `cardano-ledger` actually supported the Alonzo-era features. Of course, this resulted in the unintended consequence that we used `TxInfo` types (types that are designed to be used in Plutus scripts) in off-chain code. This wouldn't have been a problem if there was a 1:1 relationship between on-chain and off-chain types. However, that presumption is wrong.

Let's take the example of the `TxOut` representation of `plutus-ledger-api` for `PlutusV2`.

```
data TxOut = TxOut {
  ...
  txOutReferenceScript :: Maybe ScriptHash
  }
```

As we can see, the `TxOut` can optionally store the `ScriptHash` of the referenced script. However, that is *not* the adequate representation of a `TxOut` in a transaction given the `cardano-ledger` specification. The off-chain `TxOut` should instead be:

```
data TxOut = TxOut {
  ...
  txOutReferenceScript :: Maybe Script
  }
```

where the reference script field can store that *actual* script, not just the hash.

This proved that we need to start moving away from *plutus-ledger-api* types in the off-chain part of Plutus applications, especially in components like the emulator and the chain indexer.

Decision

- We will create a `cardano-api-extended` cabal project, which will contain features and utilities on top of the `cardano-api` package. A similar idea has emerged with hydra-cardano-api. This package will contain:
  - type synonyms for working with the latest era
  - a simplified and working transaction balancing function (mainly the `Ledger.Fee.makeAutoBalancedTransaction` in `plutus-apps`)
  - validation rules (most of what's in the current `Ledger.Validation`)

  The `cardano-api-extended` package will re-export the modules from the hydra-cardano-api package which contain type synonyms for working with the latest era.
- We will remove our data type representation of a Cardano transaction (`Ledger.Tx.Internal.Tx` in `plutus-ledger`) and fully commit to `Cardano.Api.Tx.Tx era` (or `Cardano.Api.Tx.TxLatestEra`) in the codebase.
- We will replace any use of `plutus-ledger-api` types by `cardano-api` and `cardano-api-extended` types whenever we work with the off-chain part of Plutus applications. For instance, the `plutus-contract` emulator and types in the `Plutus.Contract.Request` module of `plutus-contract` will be updated to use `cardano-api` types. However, the data types in `Ledger.Tx.Constraints.TxConstraints` will continue to use `plutus-ledger-api` types because the constraints are used to generate both Plutus scripts and transactions. Therefore, there should be no breaking change on the API for writing Plutus applications.
- We will improve `cardano-api` through `cardano-api-extended` and regularly push changes upstream when possible.
- We will restructure the `Ledger.Tx.CardanoApi` module in `plutus-ledger` and move functions in `cardano-api-extended`.
- We will enhance the `plutus-contract` emulator by being able to balance and submit `cardano-api` transactions.
- We will modify the `plutus-contract` emulator to fully use the `cardano-ledger` transaction validation rules, and we will remove our custom validation rules (module `Ledger.Index` in `plutus-ledger`).

Argument

The `cardano-api` package is expected to be the supported entry point for clients to interact with Cardano chain in Haskell in the foreseeable future. Therefore, we should extensively use this package and upstream changes as much as possible so that users not using the packages in `plutus-apps` can still have a good experience writing Plutus applications using `cardano-api`.

The main arguments for creating the `cardano-api-extended` package instead of using hydra-cardano-api directly are:

- faster iterative development for extending `cardano-api`
- existing plans to upstream the hydra-cardano-api in `cardano-api`

Alternatives

Define our own data types for off-chain use

This is currently what the `plutus-apps` repository is partially doing. The main problem is that this requires significant maintenance work, especially when the cardano-ledger specification changes between eras.

Implications

- This decision *should not* impact the user-facing API of our libraries. All the changes should be internal. Changes to the public-facing API should be part of a separate ADR.
- Any orphan instances that we currently have in `plutus-ledger` will need to be moved to `cardano-api-extended`.
- The Marconi-related packages will need to work with `cardano-api` types instead of `plutus-ledger-api` types, as Marconi is a full off-chain component. This implies removing the `plutus-ledger` dependency that we currently have in `marconi`.

Notes

5.5.13 ADR 13: Transaction validity time range fix

Date: 2022-10-19

Author(s)

koslambrou <konstantinos.lambrou@iohk.io>

Status

Draft

Context

The following code samples were executed with `cabal repl plutus-ledger` on the plutus-apps commit hash 172873e87789d8aac623e014eff9a39364c719ae.

Currently, the `plutus-ledger-constraint` library has the `MustValidateIn` constraint which

1) validates that a given `POSIXTimeRange`` contains the `TxInfo`'s validity range
2) creates a transaction with the provided `POSIXTimeRange`

The implementation of 1) is trivial. However, a major issue arises for the implementation of 2). Setting the validity interval of a Cardano transaction is done by specifing the slot of the lower bound and the slot of the upper bound. Therefore, the `MustValidateIn` constraint needs to convert the provided `POSIXTimeRange` to essentially a `(Maybe Slot, Maybe Slot)`. The problem is that there are many ways to convert a `POSIXTime` to a `Slot`.

Currently, provided a `POSIXTimeRange`, `plutus-contract` does the following:

- convert the time range to a slot range with `Ledger.TimeSlot.posixTimeRangeToContainedSlotRange :: POSIXTimeRange -> SlotRange`

- convert the `SlotRange` to `(Cardano.Api.TxValidityLowerBound, Cardano.Api.TxValidityUpperBound)` (essentially a `(Maybe Slot, Maybe Slot)`)

The issue with these conversion is that the `POSIXTimeRange` and `SlotRange` intervals are type synonyms of the `PlutusLedgerApi.V1.Interval.Interval a` datatype which has has a "Closure" flag for each of the bounds.

Therefore, the conversions yields a discrepency when *cardano-ledger* converts the `(Cardano.Api.TxValidityLowerBound, Cardano.Api.TxValidityUpperBound)` to a `POSIXTimeRange` when creating the `TxInfo`.

Let's show some examples to showcase the issue.

```
> let sc = SlotConfig 1000 0
> let interval = (Interval (LowerBound (Finite 999) False) (UpperBound PosInf True))
> let r = posixTimeRangeToContainedSlotRange sc interval
> r
Interval {ivFrom = LowerBound (Finite (Slot {getSlot = 0})) False, ivTo = UpperBound␣
↪PosInf True}
> let txValidRange = toCardanoValidityRange r
> txValidRange
Right (TxValidityLowerBound ValidityLowerBoundInBabbageEra (SlotNo 1),
↪TxValidityNoUpperBound ValidityNoUpperBoundInBabbageEra)
```

When creating the `TxInfo`, `cardano-ledger` will convert the previous `cardano-api` validity slot range to:

```
(Interval (LowerBound (Finite 1000) True) (UpperBound PosInf True))
```

In practical reasoning, `LowerBound (Finite 999) False` and `LowerBound (Finite 1000) True` are equal considering the precision of 1000 milliseconds per slot. However, given `Interval` semantics, these are not the same values. Therefore, if the constraint `mustValidateIn interval` is used both to create a transaction and inside a Plutus script (corresponds to the check `interval `contains` txInfoValidRange scriptContextTxInfo`), then the Plutus script will yield `False`.

We can identify a similar behavior with the upper bound.

```
> let sc = SlotConfig 1000 0
> let interval = (Interval (LowerBound NegInf True) (UpperBound (Finite 999) True))
> let r = posixTimeRangeToContainedSlotRange sc interval
> r
Interval {ivFrom = LowerBound NegInf True, ivTo = UpperBound (Finite (Slot {getSlot =␣
↪0})) True}
> let txValidRange = toCardanoValidityRange r
> txValidRange
Right (TxValidityNoLowerBound,TxValidityUpperBound ValidityUpperBoundInBabbageEra␣
↪(SlotNo 1))
```

When creating the `TxInfo`, `cardano-ledger` will convert the previous `cardano-api` validity slot range to:

```
(Interval (LowerBound NegInf True) (UpperBound (Finite 1000) False))
```

Again, a Plutus script with `interval `contains` txInfoValidRange scriptContextTxInfo` will yield `False`.

Additionnaly, the current behavior makes it hard to reason about how a `POSIXTime` gets translated into a `Slot` when creating a transaction. Ultimately, a DApp developer should have control over how his `POSIXTime` gets translated to a `Slot`.

Decision

- We will create the following datatype:

```
-- | ValidityInterval is a half open interval. Closed (inclusive) on the bottom,␣
↪open
-- (exclusive) on the top. A 'Nothing' on the bottom is negative infinity, and a
↪'Nothing'
-- on the top is positive infinity.
data ValidityInterval a = ValidityInterval
  { invalidBefore :: !(Maybe a) -- ^ Inclusive lower bound or negative infinity
  , invalidHereafter :: !(Maybe a) -- ^ Exclusive upper bound or positive infinity
  }
```

- We will add the following constraint and smart constructor:

```
data TxConstraint =
  ...
  MustValidateInTimeRange !(ValidityInterval POSIXTime)

mustValidateInTimeRange :: !(ValidityInterval POSIXTime) -> TxConstraints
```

- We will remove the `MustValidateIn` constraint and deprecate the the `mustValidateIn` smart constructor which will be replaced by `mustValidateInTimeRange`.
- We will create the smart constructor

```
mustValidateInSlotRange :: !(ValidityInterval Slot) -> TxConstraints
```

  which will translate the provide validity slot range into a `POSIXTimeRange` using `Ledger.TimeSlot.posixTimeRangeToContainedSlotRange`.

Argument

- The new `mustValidateInTimeRange` constraint will solve the discrepency between the way the validity constraint range converts a `POSIXTime` to a `Slot` and how `cardano-ledger` converts the `Slot` to `POSXITime` when creating the `TxInfo`.
- However, it won't solve the issues when the provided `POSIXTimeRange` is not an unit of 1000 milliseconds. For this scenario, we provide the `mustValidateInSlotRange` which will always create `POSIXTimeRange` that is an unit of 1000 milliseconds.
- Another benefit of the `mustValidateInSlotRange` constraint is to give control to the users on how to convert their times in `POSIXTime` to a `Slot`.

Implications

- We will have to update the `plutus-use-cases` examples to use `mustValidateInSlotRange` when creating transactions, but still use `POSIXTime` or `POSIXTimeRange` when defining the parameters (inputs) of the use cases. Same for end-users.

Alternatives

Add `MustValidateInSlotRange` constraint

If we decide to go in the direction of only specifying slots when creating transaction, then a logical solution would be replace the `MustValidateInTimeRange` constraint by `MustValidateInSlotRange (Maybe Slot) (Maybe Slot)`. However, the main issue with this solution is that this constraint would not work in a Plutus script, because there is no way to convert the `POSIXTimeRange` validity range of a `TxInfo` to a `(Maybe Slot) (Maybe Slot)`.

Remove `mustValidateInTimeRange`

By defining `mustValidateInSlotRange`, we could decide to completly remove `mustValidateInTimeRange` and force users to work with slots. However, unless we get clear feedback from end-users, we will keep `mustValidateInSlotRange` until new evidence says otherwise.

Alter `mustValidateInTimeRange`

Another alternative solution would be to keep `mustValidateInTimeRange`, but with additonnal parameters which would specify how to convert the `(Maybe POSIXTime, Maybe POSIXTime)` to a `(Maybe Slot, Maybe Slot)`. For example, given the lower (or upper) bound of the `POSIXTimeRange`, do we convert it to the closest slot? Or do we convert it to the lower (or upper) bound slot that includes the `POSIXTime`? This can potentially be discussed in a future ADR if there is value for end-users.

Notes

This ADR is motivied by the SealedBidAuction bug fix in the PR #767.

This ADR has been implemented here: #878.

5.5.14 ADR 14: Marconi Query Interface

Date: 2022-10-27

Author(s)

Kayvan Kazeminejad <kayvan.kazeminejad@iohk.io>

Status

Draft

Context

Marconi provides a general solution for indexing the blockchain data. The query interface adds reporting capabilities on top of Marconi for both Haskell and non-Haskell applications.

Decision

- We will build the interface on top of the Marconi indexer with minimal impact on the Marconi infrastructure
- The query interface may be used both as an executable or a Haskell library
- The query interface supports JSON-RPC 2.0 on top of HTTP
- The query interface provides reporting of both memory and disk storage of indexers as described in Marconi implementation

Implications

- No changes to the marconi infrastructure
- we will remain with SQLite as our storage engine

5.5.15 ADR 15: Time conversion semantic change

Date: 2022-11-19

Author(s)

koslambrou <konstantinos.lambrou@iohk.io>

Status

Draft

Context

Currently, PAB users need to provide the `SlotConfig` in the configuration file, which is passed through to the `Contract` API, which users can use to convert between a `Slot` and a `POSIXTime`. However, the current `SlotConfig` representation supposes that the slot length is the same for all epochs in the Cardano blockchain, which is not the case. For example, during the Byron era, the slot length was 20s, while from Shelley era and onwards, the slot length is 1s. Therefore, the functions from the `Ledger.TimeSlot` module in `plutus-ledger` do not compute the conversion between `Slot` and `POSIXTime` the right way. The current easiest way to compute the time conversions is to query the local Cardano node on the consensus layer, which requires the `ouroboros-consensus` dependency.

Decision

- We will deprecate the `Ledger.TimeSlot.SlotConfig` type and all functions in the `Ledger.TimeSlot` module using the `SlotConfig`. The only viable functions are the ones that convert between `Data.Time` types and plutus types (types related to `TxInfo`).
- We will copy the `Ledger.TimeSlot` module in the emulator (ideally rename it) and keep it as an internal module. Any functions not used by the emulator will be removed.
- We will move the `Ledger.Params` module inside the emulator as an internal module and modify the `Params` datatype name to `EmulatorParams`.
- We will modify the `Plutus.Contract.Request.getParams` function to `Plutus.Contract.Request.getProtocolParameters`. This implies modifying the name of Contract effect `GetParamsReq/GetParamsResp`.
- We will create two pairs of effects in `Plutus.Contract.Effect`:

```
data PABReq =
  ...
  | SlotToUTCTimeIntervalReq SlotNo
  | UTCTimeToSlotReq UTCTime
  ...

 data PABResp =
  ...
  | SlotToUTCTimeIntervalResp (UTCTime, SlotLength) -- An alternative can be␣
↪(UTCTime, UTCTime)
  | UTCTimeToSlotResp SlotNo
  ...
```

- We will implement the emulator effect interpreter by simply using the `SlotConfig` for the conversions.
- We will implement the PAB effect interpreter by using the local node. There are multiple steps to implement this:
  - At startup, the PAB will query the `EraHistory` from the local node and store it in its local environment.
  - We will implement the PAB interpreter by using the `EraHistory` alongside the consensus functions `wallclockToSlot` and `slotToWallclock`. Here's an example function of how to use them:

```
-- Calculate slot number which contains a given timestamp
utcTimeToSlotNo
  :: SystemStart
  -> EraHistory CardanoMode
  -> Time.UTCTime
  -> Either PastHorizonException SlotNo
utcTimeToSlotNo systemStart (EraHistory _ interpreter) time = do
  let relativeTime = toRelativeTime systemStart time
  (slotNo, _, _) <- interpretQuery interpreter $ wallclockToSlot relativeTime
  pure slotNo

slotStart
  :: SystemStart
  -> EraHistory CardanoMode
  -> SlotNo
  -> Either PastHorizonException Time.UTCTime
slotStart systemStart (EraHistory _ interpreter) slotNo = do
  (relativeTime, _) <- interpretQuery interpreter $ slotToWallclock slotNo
  pure $ fromRelativeTime systemStart relativeTime
```

However, we will also add an additional step. If the conversion returns `PastHorizonException`, then there is a good probability that the `EraHistory` is out of date. The reason is that `EraHistory` only encodes era information from the moment the user ran the query and it cannot predict the future. In that case, if the `PastHorizonException` is returned, we will re-query the `EraHistory` of the local node, replace the old value in the PAB environment, and retry the conversion. If it fails again, we return the error message.

Argument

The solution to create new effects in `Plutus.Contract.Effects` has the nice property that the implementation can differ depending on the environment. It allows us to keep using `SlotConfig` in the emulator, while using the existing implementation provided by `ouroboros-consensus` when using a real Cardano node.

Implications

- We will have to update the `plutus-use-cases` examples to use those new conversion functions. The user will **not** use `SlotConfig` to convert between slots and UTC time. He will instead need to use the new effects defined by the `Contract` API.

Alternatives

Changing the representation of `SlotConfig` to the correct one

A good solution would be to reuse the `Summary` datatype from `ouroboros-consensus` which has the correct representation. However, the emulator does not depend on `ouroboros-consensus` and adding it would incur a large dependency footprint for such a simple need. Additionnaly, consensus doesn't (and doesn't want to) expose the `Summary`, which is internal to the `Interpreter` datatype, which in turn is returned by `cardano-api` when querying the local node for the `EraHistory`. Thus, even if we copy-pasted the `Summary` datatype in the emulator, we would still need to find a way to query the `Summary` of `ouroboros-consensus` and convert it to our own `Summary`.

The ideal solution would be coordinate with the maintainers of `ouroboros-consensus` to move out the `Summary` datatype in a (small) consensus core module and find a way to reconstruct the `Summary` when querying the local node. This should be done in the long term, but it is not our current focus.

Directly use EpochInfo in the emulator and Contract API

Another thought of solution would be to replace the use of `SlotConfig` which `EpochInfo`. However, we need `EpochInfo` to be an instance of `FromJSON/ToJSON`, which is not possible because its data contructor parameters are functions.

Notes

5.5.16 ADR 16: Self-contained cardano-node emulator component

Date: 2022-11-23

Author(s)

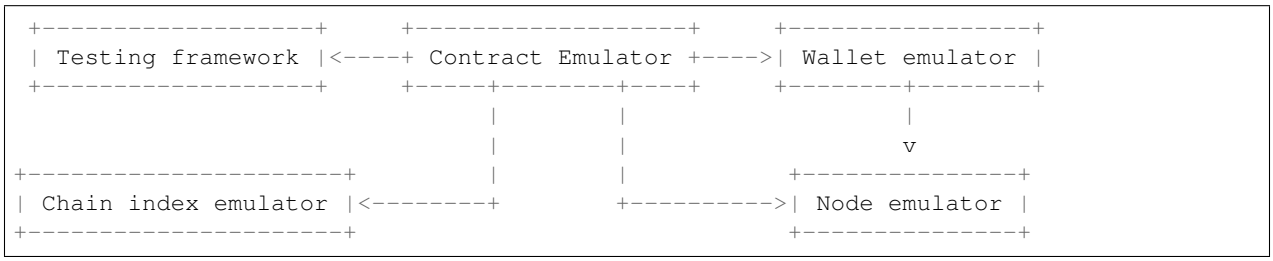
koslambrou <konstantinos.lambrou@iohk.io>

Status

Draft

Context

The current `plutus-contract` Haskell package allows developers to write Plutus applications using the `Contract` Monad. On top of that, the package contains a Plutus application contract emulator, a way to run those contracts on a simulated environment so that they can be tested. The emulator in question contains multiple components like the testing framework (including the `ContractModel`), the wallet emulator, a chain-index emulator and the node emulator.:



The main reason we can't use a real wallet or node backend is because they are not fast enough to be able to run many test cases using property tests with the `ContractModel`.

Now, we believe the node emulator to be a useful separate component that other testing framework can leverage for being able to write fast test cases.

Decision

- We will create a new Haskell package named `cardano-node-emulator`.
- We will move node related functionality into this new package. Here are some modules (or parts of a module) that will need to be moved over.
  - The `Ledger.Validation` module which validates transactions using `cardano-ledger` should only be used by `cardano-node-emulator` and should not be exposed.
  - The `Ledger.Fee` module, which calculates the fees for a given transaction, should be internal to `cardano-node-emulator`.
  - The `Ledger.Generators` module, which contains generators for constructing blockchains and transactions for use in property-based testing, should be internal to `cardano-node-emulator`.

- The `Ledger.TimeSlot.SlotConfig` datatype should only be used by the `cardano-node-emulator`. An end user should not use this representation in a real world scenario. See *ADR 15: Time conversion semantic change* for more details.
- The `Wallet.Emulator.Chain` module in `plutus-contract` should be moved in `cardano-node-emulator`.
- The `Ledger.Params` which allows to configuration the network parameters should be moved over to `cardano-node-emulator`.

Argument

Splitting out the node emulator in a separate Haskell component allows to better scope it's dependency footprint. We don't want the list of dependencies to be too large in order to keep it as a lightweight component.

Notes

This ADR is addressed in PR #831.

5.5.17 ADR 17: End-to-end testing strategy for Plutus, cardano-ledger-api and cardano-node-client

Date: 2022-12-02

Author(s)

james-iohk <james.browning@iohk.io>

Status

Draft

Context

End-to-end testing of Plutus Core functionality is currently performed by a combination of automation and exploratory approaches, both are performed on public *preview* and *pre-prod* testnets using a real node. Automation test scenarios for Plutus are currently being run as part of the wider cardano-node-tests test suite, which uses a Python wrapper for `cardano-cli`. Those tests focus on general ledger/node/cli functionality and only cover a few key scenarios for Plutus functionality, such as TxInfo and SECP256k1 builtins.

There is also ongoing development work to separate the functionality of cardano-api out into two packages:

- cardano-ledger-api handles the building and balancing of transactions.
- `cardano-node-client` will live in cardano-node and handle the submitting of balanced transactions and querying the ledger state.

Both of these packages are in early stage development and will require end-to-end test coverage.

This document outlines the decisions and arguments for an **additional** approach to end-to-end test automation using a framework written in Haskell.

The exploratory testing approach is not in the scope of this document.

Decision

- We will create a new end-to-end testing framework written in Haskell called `plutus-e2e-tests` that will initially be a package in plutus-apps, see *argument 1*.
- We will use cardano-testnet for configuring and initialising local test network environments, see *argument 2*.
- We will initially use `cardano-api` for building and balancing transactions, and to submit balanced transactions and for querying the ledger, see *argument 3*.
- When available, we will use `cardano-ledger-api` instead of `cardano-api` for building and balancing transactions, see *argument 4*.
- When available, we will use `cardano-node-client` instead of `cardano-api` to submit balanced transactions and for querying the ledger state to make test assertions, see *argument 5*.
- We will prioritise `Plutus` test coverage over `cardano-node`, see *argument 6*.
- We will start by creating a few tests with the node/ledger apis without depending on `plutus-apps` and then assess whether we want to use the Contract API and other off-chain tooling going forwards, see *argument 7*.
- We will continue adding a subset of Plutus tests to `cardano-node-tests`, see *argument 8*.

Types of Plutus tests for the `plutus-e2e-tests` Haskell framework

All `Plutus` end-to-end testing requirements will be covered by `plutus-e2e-tests`. In summary, with access to the Haskell and Plutus interfaces and reduced friction from using a single programming language we are likely improve test coverage at this level. For example, builtin functions and error scenarios.

Although we will be building out the end-to-end test coverage, it is more efficient to have fewer and broader test scenarios at this level and a greater number of tests at the lower unit and integration levels for stressing particular features and covering negative scenarios and edge cases.

Examples

- Any Plutus Core builtin function. These may already be tested extensively in the lower unit/property/integration levels but there's value in having some coverage at the end-to-end level too.
- Use cases that go beyond testing features in isolation. Bringing together various functionality helps demonstrate the capability of more realistic Plutus applications.
- Functionality introduced by a new Plutus version. This could mean that `ScriptContext` changes to accommodate an extended transaction body.
- Functionality of old Plutus versions tested with each supporting script version. Tests for old Plutus version functionality will also be run using the newer script versions.

Argument

1. The primary aim is to satisfy all of `Plutus` (core) end-to-end testing requirements, although, this is an opportunity to also get coverage of other packages being developed such as `cardano-testnet`, `cardano-ledger-api` and `cardano-node-client`. We can configure external packages and their versions using CHaPs so there is no need for `plutus-e2e-tests` to have its own repo. It will initially be be a package in `plutus-apps`.
2. There are a few options for configuring and starting a private testnet (see *local testnet* notes section). We will use `cardano-testnet` to enable dynamic configuration in Haskell, which makes it easier to design tests that can also run in the emulated environment. Also, this is the approach officially supported by the node team.

3. The plan is to start building tests with `cardano-api` because neither `cardano-ledger-api` or `cardano-node-client` are at the required stage of development. This allows us to immediately proceed with building out the framework and test suite.
4. The ambition of `cardano-ledger-api` is to be the go-to api for building transactions for application developers. The UX and overall quality of this component will benefit from being included in these end-to-end tests because of the high-level perspective applied during its design and development. When ready we will begin incorporating it as a replacement for `cardano-api`.
5. `cardano-node-client` will eventually replace `cardano-api` as a interface with consensus. As the expected means to submit and query for application developers, it is a vital we include it under test in `plutus-e2e-tests`. When ready we will begin incorporating it as a replacement for `cardano-api`.
6. Although `cardano-ledger-api` and `cardano-node-client` are under test it isn't feasible to expect thorough coverage of all ledger and node functionality, such as staking and update proposals, because the primary focus is to satisfy end-to-end testing requirements for `Plutus`. Fortunately, much of that functionality is already being covered by `cardano-node-tests`.
7. Initially, a few tests will be created without depending on Plutus-apps. This entails building transactions with `cardano-ledger-api` and waiting for on-chain events using `cardano-node-client` without use of the Contract api or the constraints library. This approach allows us to build specific transactions, which is especially useful for testing edge-cases and error scenarios that off-chain tooling may prohibit. However, this approach will require more boilerplate code and this could negatively impact readability of the tests. Having assessed this approach, we may then decide to depend on `Plutus-apps` for the Contract api, which would give a uniform interface for off-chain code such as different node backends (private and public testnets, and emulator) and chain-indexer queries (`cardano-node-client` or Marconi in future). It should also reduce the amount of boilerplate code and provide additional features such as trace logging.
8. There's value continuing to test `cardano-cli` with Plutus transactions for specific cli flags and the cli's error handling with script evaluation. Some examples of tests that should be covered:
    - Cli flags that require use of Plutus scripts E.g. tx-out-reference-script-file or calculate-plutus-script-cost
    - Cli behaviour when script evaluation passes. This could be displaying expected fee correctly.
    - Cli behaviour when script evaluation fails. This can be how different types errors are formatted.
9. At some point, we may wish to incorporate the cardano-node-emulator as an alternative to `cardano-testnet`. This would enable us to run property based tests due to the node being much faster without consensus. With CHaP, `cardano-node-emulator` would be released as a separate component, so no need to depend on `Plutus-apps`.
10. We reserve the option of including additional packages to test from `Plutus-apps` at a later stage.

Pros of building and maintaining our own test framework

- Plutus tools team will have full ownership of the end-to-end test environment and its priorities for `Plutus`.
- Plutus scripts can be defined alongside the tests. `cardano-node-tests` requires pre-compiled scripts.
- Tests will demonstrate how these Haskell packages can be used together to guide Plutus application development using the node apis. Particularly useful for less experienced Haskell developers.
- Possible to define tests once and run at different levels. For example, on private or public testnets and with `cardano-node-emulator` emulated node.
- Benefits from use of all `Plutus` apis. For example, using PlutusTx to produce scripts using a typed interface, and optionally the Contract monad from `Plutus-apps`.
- Have the opportunity to add more components under test at a later stage, such as Marconi or a PAB.

- `cardano-cli` would not be a dependency for Plutus test coverage so no risk of being blocked by its stage of development.
- Less dependence on repetitive manual approach for regression testing because tests can be planned and implemented in parallel with feature implementation and integration.
- Plutus team can implement and review majority of tests in Haskell rather than Python, which is likely to be the team's preference. Also won't need to review as many tests in `cardano-node-tests`.
- Less friction caused by cross-team: planning, dependencies and expectations. Plutus team won't need to wait for node test team to implement the tests. It's likely that other node/cli features will often be prioritised.
- This approach will improve our high-level perspective of each component and help guide UX improvements.
- Now that some `plutus` tests exist in `cardano-node-tests` the process for adding new tests will be relatively straightforward, for some it's mostly a copy/paste job. This means less work to support some duplicate tests in both frameworks.
- Node team are not pressured to focus on Plutus scenarios, they retain control of their priorities.

Cons of building and maintaining our own test framework

- `cardano-node-tests` is well established and already has useful features such as: running tests in different eras, transitioning between eras, reporting, and measuring deviations in script cost.
- It could be quicker for us to get going to reusing the bash scripts `cardano-node-tests` have. See *local testnet* notes section for other examples of spinning up a local testnet.
- We could continue getting `plutus` end-to-end test coverage without the need to build our own framework because the node test team will continue to maintain theirs regardless.
- Plutus team will still be required to support the node test team with defining and reviewing a subset of Plutus tests in `cardano-node-tests`.
- Node test team may grow, less delays in getting Plutus tests implemented by a Python developer.
- The tests using `cardano-cli` already provide some assurance that downstream components are working correctly, so there will be some duplication of test coverage by having an additional framework.

Additional Considerations

- Business stakeholders will want to see test results to think about producing and storing a report. It would be to open source this along with the tests, like `cardano-node-tests` have done.
- At first, tests will be run on a private testnet but we must consider how these tests can also be run on a public testnet. For example, initial wallet balances and utxos will need to be handled dynamically because we'd only have control over these in the private testnet.
- Seeing as `cardano-ledger-api` and `cardano-node-client` are still in early stages of production it would make sense not to block creation of `plutus-e2e-tests`. We can begin using `cardano-api` and switch over when ready.
- End-to-end tests can be slow to execute and as the suite grows we may want to run a subset at more frequent intervals. For example, we run tests for the latest Plutus version nightly but older tests/versions are run weekly, or for tags/release only.

Alternatives

Instead of creating a new repository it is possible the end-to-end tests could live in `Plutus-apps`. Although, because the components under test span other repositories it would be restrictive and additional work at the time when dependencies are updated in `Plutus-apps`, see *argument 1*.

We could use bash scripts to spin up a local testnet, which is the approach teams such as Djed and Hydra took. Although, the decision is to use `cardano-testnet`, see argument *argument 2*.

Notes

This ADR document should be moved out of `Plutus-apps` and into the new end-to-end test repository once created.

Benchmarking hasn't been covered above because we already have a team dedicated to testing cardano-node performance that includes some Plutus scripts. It is an automated approach using `cardano-cli`.

Other places spinning up a local testnet

- https://github.com/woofpool/cardano-private-testnet-setup
- https://github.com/input-output-hk/mithril/mithril-test-lab
- https://github.com/input-output-hk/hydra/hydra-cluster
- https://github.com/input-output-hk/cardano-node/tree/master/scripts/byron-to-alonzo
- https://github.com/input-output-hk/cardano-js-sdk/tree/master/packages/e2e/local-network
- https://github.com/input-output-hk/cardano-wallet/blob/master/lib/wallet/exe/local-cluster.hs
- https://github.com/mlabs-haskell/plutip

5.6 Reference

5.6.1 Examples

Full examples of Plutus Applications can be found in the `plutus-apps` repository. The source code can be found in the `src` and the tests in the `test` folder.

The examples are a mixture of simple examples and more complex ones, including:

- A crowdfunding application
- A futures application
- A stablecoin
- A uniswap clone

Important: Make sure to look at the same version of the *plutus-apps* repository as you are using, to ensure that the examples work.

5.6.2 Cost model parameters

The cost model for Plutus Core scripts has a number of parameters. These are listed and briefly described below. All of these parameters are listed in the Cardano protocol parameters and can be individually adjusted.

For more details on the meaning of the parameters, consult IOHK [[2]].

Table 1: Machine parameters

Operation	Parameter name	Note
apply	cekApplyCost-exBudgetCPU	Constant CPU cost
apply	cekApplyCost-exBudgetMemory	Constant memory cost
builtin	cekBuiltinCost-exBudgetCPU	Constant CPU cost
builtin	cekBuiltinCost-exBudgetMemory	Constant memory cost
con	cekConstCost-exBudgetCPU	Constant CPU cost
con	cekConstCost-exBudgetMemory	Constant memory cost
delay	cekDelayCost-exBudgetCPU	Constant CPU cost
delay	cekDelayCost-exBudgetMemory	Constant memory cost
force	cekForceCost-exBudgetCPU	Constant CPU cost
force	cekForceCost-exBudgetMemory	Constant memory cost
lam	cekLamCost-exBudgetCPU	Constant CPU cost
lam	cekLamCost-exBudgetMemory	Constant memory cost
startup	cekStartupCost-exBudgetCPU	Constant CPU cost
startup	cekStartupCost-exBudgetMemory	Constant memory cost
var	cekVarCost-exBudgetCPU	Constant CPU cost
var	cekVarCost-exBudgetMemory	Constant memory cost

Table 2: Builtin parameters

Builtin function	Parameter name	Note
addInteger	addInteger-cpu-arguments-intercept	Linear model intercept for the CPU calculation
addInteger	addInteger-cpu-arguments-slope	Linear model coefficient for the CPU calculation
addInteger	addInteger-memory-arguments-intercept	Linear model intercept for the memory calculation
addInteger	addInteger-memory-arguments-slope	Linear model coefficient for the memory calculation
appendByteString	appendByteString-cpu-arguments-intercept	Linear model intercept for the CPU calculation
appendByteString	appendByteString-cpu-arguments-slope	Linear model coefficient for the CPU calculation
appendByteString	appendByteString-memory-arguments-intercept	Linear model intercept for the memory calculation
appendByteString	appendByteString-memory-arguments-slope	Linear model coefficient for the memory calculation
appendString	appendString-cpu-arguments-intercept	Linear model intercept for the CPU calculation
appendString	appendString-cpu-arguments-slope	Linear model coefficient for the CPU calculation
appendString	appendString-memory-arguments-intercept	Linear model intercept for the memory calculation
appendString	appendString-memory-arguments-slope	Linear model coefficient for the memory calculation
bData	bData-cpu-arguments	Constant CPU cost
bData	bData-memory-arguments	Constant CPU cost

continues on next page

Table 2 – continued from previous page

Builtin function	Parameter name	Note
blake2b_256	blake2b-cpu-arguments-intercept	Linear model intercept for the CPU calculation
blake2b_256	blake2b-cpu-arguments-slope	Linear model coefficient for the CPU calculation
blake2b_256	blake2b-memory-arguments	Constant memory cost
chooseData	chooseData-cpu-arguments	Constant CPU cost
chooseData	chooseData-memory-arguments	Constant memory cost
chooseList	chooseList-cpu-arguments	Constant CPU cost
chooseList	chooseList-memory-arguments	Constant memory cost
chooseUnit	chooseUnit-cpu-arguments	Constant CPU cost
chooseUnit	chooseUnit-memory-arguments	Constant memory cost
consByteString	consByteString-cpu-arguments-intercept	Linear model intercept for the CPU calculation
consByteString	consByteString-cpu-arguments-slope	Linear model coefficient for the CPU calculation
consByteString	consByteString-memory-arguments-intercept	Linear model intercept for the memory calculation
consByteString	consByteString-memory-arguments-slope	Linear model coefficient for the memory calculation
constrData	constrData-cpu-arguments	Constant CPU cost
constrData	constrData-memory-arguments	Constant memory cost
decodeUtf8	decodeUtf8-cpu-arguments-intercept	Linear model intercept for the CPU calculation
decodeUtf8	decodeUtf8-cpu-arguments-slope	Linear model coefficient for the CPU calculation
decodeUtf8	decodeUtf8-memory-arguments-intercept	Linear model intercept for the memory calculation
decodeUtf8	decodeUtf8-memory-arguments-slope	Linear model coefficient for the memory calculation
divideInteger	divideInteger-cpu-arguments-constant	Constant CPU cost (argument sizes above diagonal)
divideInteger	divideInteger-cpu-arguments-model-arguments-intercept	Linear model intercept for the CPU calculation (argument sizes on or below diagonal)
divideInteger	divideInteger-cpu-arguments-model-arguments-slope	Linear model coefficient for the CPU calculation (argument sizes on or below diagonal)
divideInteger	divideInteger-memory-arguments-intercept	Linear model intercept for the memory calculation (argument sizes on or below diagonal)
divideInteger	divideInteger-memory-arguments-minimum	Constant memory cost (argument sizes above diagonal)
divideInteger	divideInteger-memory-arguments-slope	Linear model coefficient for the memory calculation (argument sizes on or below diagonal)
encodeUtf8	encodeUtf8-cpu-arguments-intercept	Linear model intercept for the CPU calculation below diagonal
encodeUtf8	encodeUtf8-cpu-arguments-slope	Linear model coefficient for the CPU calculation
encodeUtf8	encodeUtf8-memory-arguments-intercept	Linear model intercept for the memory calculation
encodeUtf8	encodeUtf8-memory-arguments-slope	Linear model coefficient for the memory calculation
equalsByteString	equalsByteString-cpu-arguments-constant	Constant CPU cost (arguments different sizes)
equalsByteString	equalsByteString-cpu-arguments-intercept	Linear model intercept for the CPU calculation (arguments same size)

continues on next page

Table 2 – continued from previous page

Builtin function	Parameter name	Note
equalsByteString	equalsByteString-cpu-arguments-slope	Linear model coefficient for the CPU calculation (arguments same size)
equalsByteString	equalsByteString-memory-arguments	Constant memory
equalsData	equalsData-cpu-arguments-intercept	Linear model intercept for the CPU calculation
equalsData	equalsData-cpu-arguments-slope	Linear model coefficient for the CPU calculation
equalsData	equalsData-memory-arguments	Constant memory cost
equalsInteger	equalsInteger-cpu-arguments-intercept	Linear model intercept for the CPU calculation
equalsInteger	equalsInteger-cpu-arguments-slope	Linear model coefficient for the memory calculation
equalsInteger	equalsInteger-memory-arguments	Constant memory cost
equalsString	equalsString-cpu-arguments-constant	Constant CPU cost (arguments different sizes)
equalsString	equalsString-cpu-arguments-intercept	Linear model intercept for the CPU calculation (arguments same size)
equalsString	equalsString-cpu-arguments-slope	Linear model coefficient for the CPU calculation (arguments same size)
equalsString	equalsString-memory-arguments	Constant memory cost
fstPair	fstPair-cpu-arguments	Constant CPU cost
fstPair	fstPair-memory-arguments	Constant memory cost
headList	headList-cpu-arguments	Constant CPU cost
headList	headList-memory-arguments	Constant memory cost
iData	iData-cpu-arguments	Constant CPU cost
iData	iData-memory-arguments	Constant memory cost
ifThenElse	ifThenElse-cpu-arguments	Constant CPU cost
ifThenElse	ifThenElse-memory-arguments	Constant memory cost
indexByteString	indexByteString-cpu-arguments	Constant CPU cost
indexByteString	indexByteString-memory-arguments	Constant memory cost
lengthOfByteString	lengthOfByteString-cpu-arguments	Constant CPU cost
lengthOfByteString	lengthOfByteString-memory-arguments	Constant memory cost
lessThanByteString	lessThanByteString-cpu-arguments-intercept	Linear model intercept for the CPU calculation
lessThanByteString	lessThanByteString-cpu-arguments-slope	Linear model coefficient for the CPU calculation
lessThanByteString	lessThanByteString-memory-arguments	Constant memory cost
lessThanEqualsByteString	lessThanEqualsByteString-cpu-arguments-intercept	Linear model intercept for the CPU calculation
lessThanEqualsByteString	lessThanEqualsByteString-cpu-arguments-slope	Linear model coefficient for the CPU calculation
lessThanEqualsByteString	lessThanEqualsByteString-memory-arguments	Constant memory cost
lessThanEqualsInteger	lessThanEqualsInteger-cpu-arguments-intercept	Linear model intercept for the CPU calculation
lessThanEqualsInteger	lessThanEqualsInteger-cpu-arguments-slope	Linear model coefficient for the CPU calculation

continues on next page

Table 2 – continued from previous page

Builtin function	Parameter name	Note
lessThanEqualsInteger	lessThanEqualsInteger-memory-arguments	Constant memory cost
lessThanInteger	lessThanInteger-cpu-arguments-intercept	Linear model intercept for the CPU calculation
lessThanInteger	lessThanInteger-cpu-arguments-slope	Linear model coefficient for the CPU calculation
lessThanInteger	lessThanInteger-memory-arguments	Constant memory cost
listData	listData-cpu-arguments	Constant CPU cost
listData	listData-memory-arguments	Constant memory cost
mapData	mapData-cpu-arguments	Constant CPU cost
mapData	mapData-memory-arguments	Constant memory cost
mkCons	mkCons-cpu-arguments	Constant CPU cost
mkCons	mkCons-memory-arguments	Constant memory cost
mkNilData	mkNilData-cpu-arguments	Constant CPU cost
mkNilData	mkNilData-memory-arguments	Constant memory cost
mkNilPairData	mkNilPairData-cpu-arguments	Constant CPU cost
mkNilPairData	mkNilPairData-memory-arguments	Constant memory cost
mkPairData	mkPairData-cpu-arguments	Constant CPU cost
mkPairData	mkPairData-memory-arguments	Constant memory cost
modInteger	modInteger-cpu-arguments-constant	Constant CPU cost (argument sizes above diagonal)
modInteger	modInteger-cpu-arguments-model-arguments-intercept	Linear model intercept for the CPU calculation (argument sizes on or below diagonal)
modInteger	modInteger-cpu-arguments-model-arguments-slope	Linear model coefficient for the CPU calculation (argument sizes above diagonal)
modInteger	modInteger-memory-arguments-intercept	Linear model intercept for the memory calculation
modInteger	modInteger-memory-arguments-minimum	Constant memory cost (argument sizes above diagonal)
modInteger	modInteger-memory-arguments-slope	Linear model coefficient for the memory calculation (argument sizes on or below diagonal)
multiplyInteger	multiplyInteger-cpu-arguments-intercept	Linear model intercept for the CPU calculation
multiplyInteger	multiplyInteger-cpu-arguments-slope	Linear model coefficient for the CPU calculation
multiplyInteger	multiplyInteger-memory-arguments-intercept	Linear model intercept for the memory calculation
multiplyInteger	multiplyInteger-memory-arguments-slope	Linear model coefficient for the memory calculation
nullList	nullList-cpu-arguments	Constant CPU cost
nullList	nullList-memory-arguments	Constant memory cost
quotientInteger	quotientInteger-cpu-arguments-constant	Constant CPU cost (argument sizes above diagonal)
quotientInteger	quotientInteger-cpu-arguments-model-arguments-intercept	Linear model intercept for the CPU calculation (argument sizes on or below diagonal)
quotientInteger	quotientInteger-cpu-arguments-model-arguments-slope	Linear model coefficient for the CPU calculation (argument sizes on or below diagonal)
quotientInteger	quotientInteger-memory-arguments-intercept	Linear model intercept for the CPU calculation (argument sizes on or below diagonal)

continues on next page

Table 2 – continued from previous page

Builtin function	Parameter name	Note
quotientInteger	quotientInteger-memory-arguments-minimum	Constant memory cost (argument sizes above diagonal)
quotientInteger	quotientInteger-memory-arguments-slope	Linear model coefficient for the memory calculation (argument sizes on or below diagonal)
remainderInteger	remainderInteger-cpu-arguments-constant	Constant CPU cost (argument sizes above diagonal)
remainderInteger	remainderInteger-cpu-arguments-model-arguments-intercept	Linear model intercept for the CPU calculation (argument sizes on or below diagonal)
remainderInteger	remainderInteger-cpu-arguments-model-arguments-slope	Linear model coefficient for the CPU calculation (argument sizes on or below diagonal)
remainderInteger	remainderInteger-memory-arguments-intercept	Linear model intercept for the memory calculation (argument sizes on or below diagonal)
remainderInteger	remainderInteger-memory-arguments-minimum	Constant memory cost (argument sizes above diagonal)
remainderInteger	remainderInteger-memory-arguments-slope	Linear model coefficient for the memory calculation (argument sizes on or below diagonal)
sha2_256	sha2_256-cpu-arguments-intercept	Linear model intercept for the CPU calculation
sha2_256	sha2_256-cpu-arguments-slope	Linear model coefficient for the CPU calculation
sha2_256	sha2_256-memory-arguments	Constant memory cost
sha3_256	sha3_256-cpu-arguments-intercept	Linear model intercept for the CPU calculation
sha3_256	sha3_256-cpu-arguments-slope	Linear model coefficient for the CPU calculation
sha3_256	sha3_256-memory-arguments	Constant memory cost
sliceByteString	sliceByteString-cpu-arguments-intercept	Linear model intercept for the CPU calculation
sliceByteString	sliceByteString-cpu-arguments-slope	Linear model coefficient for the CPU calculation
sliceByteString	sliceByteString-memory-arguments-intercept	Linear model intercept for the memory calculation
sliceByteString	sliceByteString-memory-arguments-slope	Linear model coefficient for the memory calculation
sndPair	sndPair-cpu-arguments	Constant CPU cost
sndPair	sndPair-memory-arguments	Constant memory cost
subtractInteger	subtractInteger-cpu-arguments-intercept	Linear model intercept for the CPU calculation
subtractInteger	subtractInteger-cpu-arguments-slope	Linear model coefficient for the CPU calculation
subtractInteger	subtractInteger-memory-arguments-intercept	Linear model intercept for the memory calculation
subtractInteger	subtractInteger-memory-arguments-slope	Linear model coefficient for the memory calculation
tailList	tailList-cpu-arguments	Constant CPU cost
tailList	tailList-memory-arguments	Constant memory cost
trace	trace-cpu-arguments	Constant CPU cost
trace	trace-memory-arguments	Constant memory cost
unBData	unBData-cpu-arguments	Constant CPU cost
unBData	unBData-memory-arguments	Constant memory cost
unConstrData	unConstrData-cpu-arguments	Constant CPU cost
unConstrData	unConstrData-memory-arguments	Constant memory cost
unIData	unIData-cpu-arguments	Constant CPU cost
unIData	unIData-memory-arguments	Constant memory cost

continues on next page

Table 2 – continued from previous page

Builtin function	Parameter name	Note
unListData	unListData-cpu-arguments	Constant CPU cost
unListData	unListData-memory-arguments	Constant memory cost
unMapData	unMapData-cpu-arguments	Constant CPU cost
unMapData	unMapData-memory-arguments	Constant memory cost
verifySignature	verifySignature-cpu-arguments-intercept	Linear model intercept for the CPU calculation
verifySignature	verifySignature-cpu-arguments-slope	Linear model coefficient for the CPU calculation
verifySignature	verifySignature-memory-arguments	Constant memory cost

5.6.3 Glossary

active endpoint An endpoint that is active on a contract application instance. Indicates that the contract application instance is waiting for input. The set of active endpoints is part of the state of the contract application instance and changes over time.

address The address of an UTXO says where the output is "going". The address stipulates the conditions for unlocking the output. This can be a public key hash, or (in the Extended UTXO model) a script hash.

Cardano The blockchain system upon which the Plutus Platform is built.

contract application An application written against the contract application API, which runs in the PAB.

contract application API The API that provides an interface between a contract application and the PAB. Also allows the contract to declare contract endpoints that will be forwarded on to PAB clients via the application interface.

contract application instance A configured, running instance of a contract application. Configuration and initialization may require additional parameters to be set by the user. Has its state and lifecycle managed by the PAB.

contract endpoint An interface point exposed by a contract application as part of its own API. These are forwarded on by the PAB to the wallet frontend or other clients.

contract executable A compiled executable of a contract application. These are what are actually distributed to users and run by the PAB.

currency A class of token whose minting is controlled by a particular monetary policy script. On the Cardano ledger there is a special currency called Ada which can never be minted and which is controlled separately.

datum The data field on script outputs in the Extended UTXO model.

emulator An in-process (single thread) emulated blockchain for testing and analysing Plutus apps.

endpoint A potential request made by a contract application for user input. Every endpoint has a name and a type.

Extended UTXO Model The ledger model which the Plutus Platform relies on.

This is implemented in the Alonzo hard fork of the Cardano blockchain.

See What is a ledger?

minting A transaction which mints tokens creates new tokens, providing that the corresponding minting policy script is satisfied. The amount minted can be negative, in which case the tokens will be destroyed instead of created.

minting context A data structure containing a summary of the transaction being validated, and the current minting policy which is being run.

minting policy script A script which must be satisfied in order for a transaction to mint tokens of the corresponding currency.

Hydra A Layer 2 scalability solution for Cardano. See Chakravarty *et al.* [[1]].

distributed ledger

ledger See What is a ledger?

Marlowe A domain-specific language for writing financial contract applications.

multi-asset A generic term for a ledger which supports multiple different asset types natively.

off-chain code The part of a contract application's code which runs off the chain, usually as a contract application.

on-chain code The part of a contract application's code which runs on the chain (i.e. as scripts).

PAB client API The API that the PAB provides to allow PAB clients to interact with contract application instances. Contract endpoints which are exposed by running instances can be called via the client API.

PAB client A program which interacts with a contract application instance via the PAB's client API. Examples of PAB clients include:

1. Wallet frontends such as Daedalus.
2. Other user software which uses the contract application as part of a wider system.

Plutus Application An application written using the Plutus Application Framework.

pab

Plutus Application Backend (PAB) The component which manages Plutus Applications that run on users' machines. It handles:

1. Interactions with the node
2. Interactions with the wallet backend
3. Interactions with the wallet frontend
4. State management
5. Tracking historical chain information

Plutus Core The programming language in which scripts on the Cardano blockchain are written. Plutus Core is a small functional programming language—a formal specification is available with further details. Plutus Core is not read or written by humans, it is a compilation target for other languages.

See What is Plutus Foundation?

Plutus IR An intermediate language that compiles to Plutus Core. Plutus IR is not used by users, but rather as a compilation target on the way to Plutus Core. However, it is significantly more human-readable than Plutus Core, so should be preferred in cases where humans may want to inspect the program.

Plutus Platform The combined software support for writing contract applications, including:

1. Plutus Foundation, and
2. The Plutus Application Framework

See What is Plutus Platform?

Plutus SDK The libraries and development tooling for writing contract applications in Haskell.

Plutus Tx The libraries and compiler for compiling Haskell into Plutus Core to form the on-chain part of a contract application.

redeemer The argument to the validator script which is provided by the transaction which spends a script output.

rollback The result of the local node switching to the consensus chain. See *What is a rollback?*.

schema The set of all endpoints of a contract application.

script A generic term for an executable program used in the ledger. In the Cardano blockchain, these are written in Plutus Core.

script output A UTXO locked by a script.

token A generic term for a native tradeable asset in the ledger.

transaction output Outputs produced by transactions. They are consumed when they are spent by another transaction. Typically, some kind of evidence is required to be able to spend a UTXO, such as a signature from a public key, or (in the Extended UTXO Model) satisfying a script.

UTXO An unspent *transaction output*

utxo congestion The effect of multiple transactions attempting to spend the same *transaction output*. See *UTXO congestion*.

validator script The script attached to a script output in the Extended UTXO model. Must be run and return positively in order for the output to be spent. Determines the address of the output.

validation context A data structure containing a summary of the transaction being validated, and the current input whose validator is being run.

5.6.4 Bibliography

5.6.5 Elsewhere

- Haddock generated documentation

BIBLIOGRAPHY

[1] Manuel M. T. Chakravarty, Sandro Coretti, Matthias Fitzi, Peter Gazi, Philipp Kant, Aggelos Kiayias, and Alexander Russell. Hydra: fast isomorphic state channels. Technical Report, Cryptology ePrint Archive, Report 2020/299, 2020. URL: https://eprint.iacr.org/2020/299.

[2] IOHK. Plutus platform technical report. Technical Report, IOHK, 2019. Available at https://github.com/input-output-hk/plutus.

INDEX