CARDANO DEVELOPMENT COURSE (Week 3-4)

Module 3: Aiken Intensive.

3.0. Aiken Testing and Debugging

3.1. Aiken Validators (Mint and Spend Handlers)

3.2. Vesting Contract in Aiken

3.3. Minting Contract in Aiken

3.4. One-shot Minting Policy

3.0. Aiken Testing and Debugging

On-chain programming can be quite demanding and bears many similarities to embedded programming. Due to the highly restricted execution environment, programs must be tightly optimized, often leaving minimal opportunity for debugging or error handling.

Aiken aims to ease this process by offering developers additional tools and debugging features. Let's take a closer look at what it provides.

3.0.1. Traces

Your first ally in this journey are traces. Think of a trace as a log message, that is captured by the virtual machine at a specific moment. You can add traces to top-level expressions in Aiken using the trace keyword.

For example:

```
fn is_even(n: Int) -> Bool {
  trace @"is_even"
  n % 2 == 0
}

fn is_odd(n: Int) -> Bool {
  trace @"is_odd"
  n % 2 != 0
}
```

Another example:

```
fn foo(my_list: List<Option<Int>>) -> Bool {
  trace @"foo": my_list
  list.is_empty(my_list)
}
```

Further Reading: https://aiken-lang.org/language-tour/troubleshooting#traces

3.0.2. ? Operator

At their core, on-chain programs are simply predicates—functions that return either True or False. As a result, it's common to structure them using combinations of boolean expressions through logical conjunctions and disjunctions.

However, reasoning about such programs can be challenging, since booleans are inherently "blind." In other words, as complex boolean expressions are evaluated, valuable context about the original conditions often gets lost.

Consider the following simple expression as an example:

```
let must_be_after = True
let must_spend_token = False

must_be_after && must_spend_token
```

This evaluates to False. However, from the result alone, you can't tell which part of the expression caused the failure. When dealing with more complex expressions, being able to trace the exact source of failure becomes crucial for debugging.

That’s where Aiken’s ? operator—also known as the "trace-if-false" operator—comes in. This postfix operator can be added to any boolean expression and will only generate a trace if the expression evaluates to False. It allows you to follow the entire chain of evaluation that led to the failure.

Using it, the example above could be rewritten as:

```
must_be_after? && must_spend_token?
```

This would produce the trace: "must_spend_token ? False".

Pretty useful, right?

Further reading: https://aiken-lang.org/language-tour/troubleshooting#-operator

3.0.3. CBOR diagnostic

All of this is helpful, but sometimes you need more. In certain cases, you want to inspect the value of a specific object at runtime. This is easier said than done, because once an Aiken program is compiled, much of the original context is stripped away—including type information. Even function and variable names are reduced to compact indices, making it quite difficult to examine programs and their values during execution.

Aiken's standard library offers a handy way to inspect any value at runtime by converting it into a String representation. This format, known as CBOR diagnostic notation, is a human-readable syntax similar to JSON but capable of representing binary data as well.

aiken/cbor:

```
pub fn diagnostic(data: Data) -> String
```

Here's a little cheatsheet to help you decipher CBOR diagnostics:

Type	Examples
`Int`	1, -14, 42
`ByteArray`	`h'FF00'`, `h'666f6f'`
`List`	`[]`, `[1, 2, 3]`, `[_ 1, 2, 3]`
`Map`	`{}`, `{ 1: h'FF', 2: 14 }`, `{_ 1: "AA" }`
`Tag`	`42(1)`, `10(h'ABCD')`, `1280([1, 2])`

Example usage:

```
use aiken/cbor

test my_datum_1() {
  let datum = MyDatum { foo: 42, bar: "Hello, World!" }
  cbor.diagnostic(datum) == @"121([42,
h'48656c6c6f2c20576f726c6421'])"
}
```

Further reading:

https://aiken-lang.org/language-tour/troubleshooting#cbor-diagnostic

3.1. Aiken Validators (Mint and Spend Handlers)

In Aiken, you can promote some functions to validator handlers using the keyword `validator`.
Example:

```
validator my_script {
  mint(redeemer: MyMintRedeemer, policy_id: PolicyId, self:
Transaction) {
    todo @"mint logic goes here"
  }

  else(_) {
    fail
  }
}
```

A validator is a named block that contains one or more handlers. The handler name must match Cardano's well-known purposes: mint, spend, withdraw, publish, vote or propose.

In this section, we are going to explore the mint and spend handler in-depth

3.1.1. Mint Handler

The mint handler in Aiken has a format like so:

```
mint(redeemer: MyMintRedeemer, policy_id: PolicyId, self:
Transaction) {
  todo @"mint logic goes here"
}
```

The mint handler accepts three (3) parameters; The `redeemer`, `policy ID`, and `self` which is the transaction info.

- The `policy_id` is essentially the script hash of the validator.

- Policy_id + asset_name will serve as the identifier for assets, tokens, or NFTs minted with the above validator.
- We refer to a minting validator as a minting policy.

3.1.2. Spend Handler

The spend handler in Aiken has a format like so:

```
spend(datum: Option<MyDatum>, redeemer: MySpendRedeemer, utxo:
OutputReference, self: Transaction) {
  todo @"spend logic goes here"
}
```

The spend handler accepts four(4) parameters; The datum, redeemer, policy ID, and self which is the transaction info.

- Out of all allowed handlers, the spend is the only one with 4 parameters.
- The datum is saved on-chain.

This handler governs the movement of UTxOs out the validator.

Further reading: *https://aiken-lang.org/language-tour/validators*

3.2. Vesting Contract in Aiken

Now, let’s explore a vesting contract in Aiken.

Starting with the definition of its interface (i.e. its datum’s shape):

```
use aiken/crypto.{VerificationKeyHash}

pub type VestingDatum {
  /// POSIX time in milliseconds, e.g. 1672843961000
  lock_until: Int,
  /// Owner's credentials
  owner: VerificationKeyHash,
  /// Beneficiary's credentials
  beneficiary: VerificationKeyHash,
}
```

- First, we import the type `VerificationKeyHash` from the library, `aiken/crypto`
- Then we define the type `VestingDatum` with 3 fields;
    - `lock_until`: the duration/lock time
    - `owner`: the credential of the user that locks the value
    - `beneficiary`: the credential of the user that can unlock and claim the value

An additional dependency we will add to this project is `vodka`. To include `vodka` in our Aiken project, let's update our `aiken.toml` file to specify it as a dependency.

```
[[dependencies]]
name = "sidan-lab/vodka"
version = "0.1.1-beta"
source = "github"
```

We will use two specific functions from `vodka`:

- `key_signed`: validates that a specific credential signs a transaction
- `valid_after`: validates that the transaction is executed after a specified time (such as after `lock_until` as specified in our datum type, VestingDatum)

Since we have gotten our pieces together, let's see the complete vesting validator. Note that it uses a spend handler.

```
use cardano/transaction.{OutputReference, Transaction}
use vodka_extra_signatories.{key_signed}
use vodka_validity_range.{valid_after}
use aiken/crypto.{VerificationKeyHash}

pub type VestingDatum {
  /// POSIX time in milliseconds, e.g. 1672843961000
  lock_until: Int,
  /// Owner's credentials
  owner: VerificationKeyHash,
  /// Beneficiary's credentials
  beneficiary: VerificationKeyHash,
}

validator vesting {
  // In principle, scripts can be used for different purpose (e.g.
minting
  // assets). Here we make sure it's only used when 'spending' from a
eUTxO
  spend(
    datum_opt: Option<VestingDatum>,
    _redeemer: Data,
    _input: OutputReference,
    tx: Transaction,
  ) {
    expect Some(datum) = datum_opt
    or {
      key_signed(tx.extra_signatories, datum.owner),
      and {
        key_signed(tx.extra_signatories, datum.beneficiary),
        valid_after(tx.validity_range, datum.lock_until),
      },
    }
  }
  else(_) {
    fail
  }
}
```

What is new in the code above?

- We have extra imports:
    - OutputReference and Transaction
- A validator with a spend handler
- The spend handler accepts 4 parameters:
    - datum_opt: an optional VestingDatum type
    - redeemer: of type Data (since it's preceded with an underscore (_), it means the parameter is not used in the handler)
    - input: UTxO input to the contract/validator (also with an _)
    - tx: the transaction info

Vesting Contract Validation

Starting with this line:

```
expect Some(datum) = datum_opt
```

The code above forces a required datum from datum_opt. The UTxO must have a datum before it can be spent/consumed.

Next:

```
or {
      key_signed(tx.extra_signatories, datum.owner),
      and {
        key_signed(tx.extra_signatories, datum.beneficiary),
        valid_after(tx.validity_range, datum.lock_until),
      },
    }
```

The above is an or block. One of the comma separated conditions must be satisfied for the contract to return True and the UTxO be allowed to be spent.

The first condition:

```
key_signed(tx.extra_signatories, datum.owner),
```

means the owner must sign the transaction

The second condition:

```
and {
        key_signed(tx.extra_signatories, datum.beneficiary),
        valid_after(tx.validity_range, datum.lock_until),
      },
```

is an and block, meaning the beneficiary must sign the transaction, and the transaction must be signed by the beneficiary after some certain date/time.

In a nutshell, the vesting contract allows spending of UTxOs by either:

- The owner signing the transaction
- The beneficiary signing the transaction at a due date

3.3. Minting Contract in Aiken

A minting contract governs the creation and burning(destruction/deletion) of new assets. Whether they are tokens or NFTs.
Let's see a contract that can mint and burn an NFT.

```
use aiken/collection/dict
use cardano/transaction.{Transaction}
use cardano/assets.{PolicyId}

pub type Action {
  Minting
  Burning
}

validator mint_nft() {
  mint(redeemer: Action, policy_id: PolicyId, self: Transaction) {
      // It checks that only one minted asset exists and will fail
otherwise
      expect [Pair(_asset_name, quantity)] = self.mint
        |> assets.tokens(policy_id)
        |> dict.to_pairs()

      when redeemer is {
        Minting -> (quantity == 1)?

        Burning -> (quantity == -1)?
      }
    }
  }
```

To create rules for minting an NFT, we make sure that the quantity being minted and burned must be 1. The `quantity` is destructured from `mint` from the transaction info.

3.4. One-shot Minting Policy

One-shot minting policy is very similar to the minting policy you just saw in Section 3.3. It is an extension of the nft minting policy. See below:

```
use aiken/collection/dict
use aiken/collection/list
use cardano/transaction.{OutputReference, Transaction}
use cardano/assets.{PolicyId}

pub type Action {
  Mint
  Burn
}

validator one_shot(utxo_ref: OutputReference) {
  mint(redeemer: Action, policy_id: PolicyId, self: Transaction) {
    let Transaction {inputs, mint ..} = self

    // It checks that only one minted asset exists and will fail
otherwise
    expect [Pair(_asset_name, quantity)] = mint
      |> assets.tokens(policy_id)
      |> dict.to_pairs()

    // Check if the specified UTxO reference (utxo_ref) is consumed
by any input
    let is_utxo_spent = list.any(inputs, fn(input) {
input.output_reference == utxo_ref })

    when redeemer is {
      Mint ->
        is_utxo_spent? && (quantity == 1)?

      Burn -> (quantity == -1)? // No need to check if output is
consumed for burning
    }
  }
}
```

The One-shot minting validator checks for a single mint: `quantity == 1`, and a single burn: `quantity == -1` for burning just like the nft minting validator.

In addition to checking for a single mint, it checks if the UTxO provided as a parameter to the validator is consumed.

This validator uses the concept that a UTxO can only be spent/consumed once, and once forever. Taking advantage of this, the One-shot minting validator can only mint one unique NFT with a unique policy ID (in this case with any name (asset_name)) only once forever.

Further reading:
https://aiken-lang.org/fundamentals/common-design-patterns#one-shot-minting-policies