Smart Contracts in Cardano

Claudio Hermida claudio.hermida@gmail.com
https://www.linkedin.com/in/claudiohermida/

Synopsis

- State in functional programming ~~~> eUTXO model
- Specification of contract methods ~~~~> Validator
- What is a smart contract in Cardano?

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract VestingContract {
    address public beneficiary;
    address public benefactor;
    uint public amount;
    uint public deadline;
    bool public consumed;

    constructor(
```

```
        address _beneficiary,
        uint _deadline
    ) payable {
        beneficiary = _beneficiary;
        benefactor = msg.sender;
        amount = msg.value;
        deadline = _deadline;
        consumed = false;
    }



    function claim() public {
        require(msg.sender == beneficiary, "Only the beneficiary can call
this function.");
        require(block.timestamp >= deadline, "Deadline has not passed
yet.");
        require(!consumed, "Funds have already been released.");
        consumed = true;
        payable(beneficiary).transfer(amount);
    }

    function cancel() public {
        require(msg.sender == benefactor, "Only the benefactor can call
this function.");
        require(block.timestamp < deadline, "Deadline has already
passed.");
        require(!consumed, "Funds have already been released.");
        consumed = true;
        payable(benefactor).transfer(amount);
    }
}
```

Account-based blockchain model

```
Account-Ledger = Map Address Value
```

```
smart contract = object ~~~> (state{balance, ...}, methods)
```

```
tx-method : Account-Ledger -> Account-Ledger
```

Functional Programming

- **Stateless**: functions are evaluated, reading inputs and producing outputs without manipulating any state or storage variables.
- **Referentially transparent**: statalessness enables *equational reasoning*, allowing to replace equals by equals anywhere.

Handling state

Imperative programming: a **variable** is an *object* with two methods: `read` and `update`

To emulate such state in a stateless fashion, we must emulate these two operations.

- `read`: we must provide the contents of the variable as an additional *input*
- `update`: we must produce the updated content as an additional *output* so that it can be used in later computation.

```
reverse :: List A -> List A

reverse l =
    let
       var accumulator:: List A
       reverseWithAccumulator :: List A -> List A
           reverseWithAccumulator EmptyList = accumulator -- read state
           reverseWithAccumulator a:l       =
                   accumulator :=  a: accumulator;  -- update state
                   reverseWithAccumulator l
   in
       accumulator := EmptyList;  -- initialize state
       reverseWithAccumulator l
```

```
newType State = List A

reverse l =
let
   reverseWithAccumulator' :: (List A,State) -> (List A, State)

   reverseWithAccumulator'(EmptyList,acc)  = (acc,acc)
                       -- read state => no change on second component
   reverseWithAccumulator' (a:l, acc) =
                               reverseWithAccumulator' (l, a:acc)
                       -- update state => modify second component

   in
       fst $ reverseWithAccumulator (l,EmptyList)
       -- supply initial state,
       -- select first component to get result
```

- **functional transactions** should have additional input/output to account for state
- state is associated to an **address**
- state split **(value, storage)**

∎ the additional input/output representing state is an *eutxo*

eutxo =

address	target address where the utxo is held/locked
value	value locked in this utxo
datum	local state