Smart Contracts in Cardano

Part I: Functional programming and the eUTXO model

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

Abstract

The purpose of this work is to

- Show how the eUTxO model of blockchain comes about from a consideration of handling state in functional programming
- Exhibit Cardano's validators as verifiable specifications of a contract-method input-output behaviour
- Provide a putative answer to the question: what is a smart contract in Cardano?

This is the first in a three-part series covering these topics. The other parts are (SMC-PartII) and (SMC-PartIII).

Introduction

In this first part of our conceptual revisit of Cardano's eutxo model and smart contracts, we review the functional programming approach to handling state as additional input/output to functions. After recalling the kind of state involved in account-based smart contracts, we show how treating such state functionally leads directly to Cardano's *extended unspent transaction output* (which additionally enhances data to a *resource*). We conclude by recalling Haskell's treatment of state via the *State Monad* and show its equivalence to the additional-(input/output) method.

The second part (SMC-PartII) of this work examines the link between formal specifications of methods via pre and postconditions and Cardano's validators, while the third part (SMC-PartIII) addresses the link between smart contract methods, transaction schemas and their associated validators, rounding up our treatment of smart contracts in Cardano.

Functional programming

The prevailing programming paradigm is *object oriented programming*, a natural evolution of *imperative programming* incorporating encapsulation to support *data abstraction*. Imperative programming evolved from primitive assembly languages, closely tied to the hardware model of CPU and RAM: the machine memory is a large array of updatable locations, which store both data and program. The program is understood as a sequence of instructions which fetch data from memory, perform calculations and update *variables* (references to storage locations), plus any input/output required from external devices.

Functional programming [Thompson (2011)](Bird 2014), on the other hand, is a paradigm where programs are *functional expressions* transforming inputs into outputs. Such expressions are built composing predefined

as well as user-defined functions, and evaluated by *term rewriting*, replacing an instance of the left-hand side of an equation by the corresponding instance of the right-hand side.

Ex: defining `succ` and `double` as

```
succ x = x + 1

double y = 2 * y
```

we may evaluate `double(succ 3)` as follows:

```
double ( succ 3) ~~> double( 3 + 1) ~~> double 4 ~~> 2 * 4 ~~> 8
```

(innermost redex evaluation strategy: strict evaluation)

Or

```
double (succ 3) ~~> 2 * (succ 3) ~~> 2 * (3 + 1) ~~> 2 * 4 ~~> 8
```

(outermost redex evaluation strategy: lazy evaluation)

The two main characteristics of functional programming for our considerations of eUTXO modelling are

- ***Stateless***: functions are evaluated, reading inputs and producing outputs without manipulating any state or storage variables.
- ***Referentially transparent***: the main consequence of statelessness is that we can reason equationally with functional expressions, replacing equals by equals anywhere.

Handling State

Modern functional languages like Haskell evolved from functional calculi, typically some variant of lambda calculus, used as metalanguages to express the denotational semantics of programming languages (Tenent 2023), which gives meaning to programs as functions transforming input state into output state.

In imperative programming, a *storage location* or *variable* can be thought abstractly as an *object* on which we may perform two operations: `read` and `update` (Reynolds 1981).

Example: Consider a `reverse` operation on lists using an *accumulator* to store the intermediate result of reversing the list as we go through it:

```
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
```

In order to emulate the use of the storage variable in a stateless fashion we must emulate the two operations on it:

- To emulate `read` we must provide the contents of the variable as an additional input to the `reverse` function

- To emulate `update` we must produce the updated content as an additional output so that it can be used in later computation.

```
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
```

In general, we have the following bidirectional translation:

$$\left[\mathsf{var\ s : State} \vdash \mathsf{f : I} \longrightarrow \mathsf{O}\right] \iff \left[\vdash \mathsf{f' : (I, State)} \longrightarrow \mathsf{(O, State)}\right]$$

which means that a function in the context/environment of a variable of type State can be represented by a function *in empty context* with extra input and output of type State.

Distributed Ledger and Smart Contracts

In the account-based model of smart contracts on a blockchain (Ethereum), a distributed ledger is a collection of ***accounts*** (identified by an ***address***) with associated information, in particular a *balance* of assets or values held.

```
Account-Ledger = Map Address (Value,State)
```

A ***smart contract*** is an *object* , in the sense of object-oriented programming. Therefore it has an internal *local state*, which includes the *balance*, and *methods*. A smart contract is *deployed* (held) at an (account) address in the ledger.

A ***transaction*** updates the distributed ledger by interacting with smart contracts and user held accounts, calling on their methods.

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

On formalism

We will not indulge in fully formal models of account-based or eutxo-based blockchains and related concepts like transactions, but give only indicative structures which add sufficient precision to our discussion without overloading the reader with details. A more elaborated formalisation of these notions appears in (Brünjes and Gabbay 2020) and a full-fledged Agda coding in Knispel et al. (2024)].

Example: Here is a very simple vesting contract to illustrate concepts. A **benefactor** deposits a certain **amount** to be retrived by a specified **beneficiary***, once a certain **deadline** has been reached (say, when the beneficiary comes of age). Before the deadline, the benefactor may cancel the vesting and recover the deposited amount. A boolean flag **consumed** is used to prevent multiples withdrawals/retrievals.

```
// 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);
    }
}
```

How should we handle these statefull smart contracts in a stateless fashion? Answer: eutxo (extended unspent transaction output) (Chakravarty et al. 2020); our *functional transactions* should have additional input/output to account for state.

As we mentioned, in blockchains, the state of an account, uniquely identified by an *address*, is conveniently split (conceptually) in two components: value/balance and local variables. In Satoshi Nakamoto's original utxo model for Bitcoin, only value is considered. Cardano extends this by adding a ***datum*** to the utxo to account for the local state of a contract object. So the data for an **eutxo** is a tuple consisiting of

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

This is literally the functional interpretation of the "state" embodied in the account-based model: a map from `Address` to `(Value,State)` is realized as a list of triples '[(address,value,state)]' with no repetitions in the first component. That triple is exactly the data content of an eutxo.

Just like in Bitcoin, eutxos in Cardano are ***resources***: they are *produced* by a transaction (as outputs) and can be *consumed* or *spent* once, and only once, by another transaction (as inputs). The resource nature of a eutxo is realized by uniquely identifying it with a `TxOutRef`, which is a pair `(TransactionId,TxIx)`: a transaction identifier of the transaction that produces this output, together with the index which signals its position in the list of outputs of the transaction.

According to this reformulation, a single eutxo would suffice to account for both value and local state at an address, which we can see as the equivalent of an account. However, in line with Bitcoin's utxos, Cardano allows multiple etuxos to be held at a single address, thereby effectively splitting or *sharding* the local state at an address. This sharding has great benefits for efficiency: a transaction can select very precisely those utxos which hold the piece of state of interest to consume, and transactions consuming different utxos can be computed/validated in parallel. We elaborate on transaction execution/validation later on.

Another major benefit of sharding state combined with the resource nature of eutxos is *non-interference*: an eutxo can only be used by a single transaction, so different transactions cannot operate over the same piece of state (Brünjes and Gabbay 2020, Lemma 2.15). This ensures *predictability* of the outcome of transactions, in contrast to the account-based model with shared global state among transactions; a transaction in such model can never guarantee in which initial state it will be executed.

The State Monad

Our treatment of state in functional programming has focused on the traditional and widespread technique of having an additional input/output parameter. Functional programmers familiar with Haskell know of its ubiquitous use of *monads* to encapsulate operations on types and provide a systematic composition of the associated *monadic computations*. In particular, there is a "famous" **State** monad.

Let us recall the 1-1 correspondence between functions that take a pair as argument and functions with "two arguments":

$$\big[(\mathsf{A} \times \mathsf{B}) \longrightarrow \mathsf{C}\big] \Longleftrightarrow \big[\mathsf{A} \longrightarrow (\mathsf{B} \longrightarrow \mathsf{C})\big]$$

given by the functions

```
curry :: ((A,B) -> C) -> (A -> (B -> C))
curry f = \a -> (\b -> f (a,b))

uncurry:: (A -> (B -> C)) -> ((A,B) -> C)
uncurry g = \(a,b) -> g a b
```

which satisfy

```
curry $ uncurry g == g
uncurry $ curry f == f
```

Such a correspondence is called a (*natural*) *isomorphism*. Let us apply this correspondence to a function which manipulates a state `S`

```
f:: (I,S) -> (O,S)  ~~~~~> curry(f):: I -> (S -> (O,S))
```

The expression `S -> (O,S)` is precisely the so-called *State monad* at the type `O`.

```
State:: * -> *
State a :: S -> (a,S)
```

A function $\mathsf{f} : \mathsf{I} \longrightarrow \mathsf{O}$ with state S corresponds to a monadic computation $\mathsf{curry(f)} : \mathsf{I} \longrightarrow \mathsf{State\ O}$

References

Bird, Richard. 2014. *Thinking Functionally with Haskell.* Cambridge University Press.

Brünjes, Lars, and Murdoch J Gabbay. 2020. "UTxO-Vs Account-Based Smart Contract Blockchain Programming Paradigms." In *Leveraging Applications of Formal Methods, Verification and Validation: Applications: 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20–30, 2020, Proceedings, Part III 9*, 73–88. Springer.

Chakravarty, Manuel MT, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Michael Peyton Jones, and Philip Wadler. 2020. "The Extended UTXO Model." In *Financial Cryptography and Data Security: FC 2020 International Workshops, AsiaUSEC, CoDeFi, VOTING, and WTSC, Kota Kinabalu, Malaysia, February 14, 2020, Revised Selected Papers 24*, 525–39. Springer.

Knispel, Andre, Orestis Melkonian, James Chapman, Alasdair Hill, Joosep Jääger, William DeMeo, and Ulf Norell. 2024. "Formal Specification of the Cardano Blockchain Ledger, Mechanized in Agda." In *FMBC 2024*. https://iohk.io/en/research/library/papers/formal-specification-of-the-cardano-blockchain-ledger-mechanized-in-agda/.

Reynolds, John C. 1981. *The Craft of Programming.* Englewood Cliffs, NJ: Prentice-Hall International.

Tenent, Robert. 2023. "Denotational Semantics." In *Handbook of Logic in Computer Science*, edited by S. Abramsky Dov M. Gabbay and T. S. E. Maibaum. Oxford University Press. https://academic.oup.com/book/52970/chapter/421961871.

Thompson, Simon. 2011. *Haskell: The Craft of Functional Programming.* Addison-Wesley.