# UTxO- vs account-based smart contract blockchain programming paradigms

Lars Brünjes[1] and Murdoch J. Gabbay[2]

[1] IOHK
[2] Heriot-Watt University, Scotland, UK

**Abstract.** We implement two versions of a simple but illustrative smart contract: one in Solidity on the Ethereum blockchain platform, and one in Plutus on the Cardano platform, with annotated code excerpts and with source code attached. We get a clearer view of the Cardano programming model in particular by introducing a novel mathematical abstraction which we call Idealised EUTxO. For each version of the contract, we trace how the architectures of the underlying platforms and their mathematics affects the natural programming styles and natural classes of errors. We prove some simple but novel results about alpha-conversion and observational equivalence for Cardano, and explain why Ethereum does not have them. We conclude with a wide-ranging and detailed discussion in the light of the examples, mathematical model, and mathematical results so far.

## 1 Introduction

In the context of blockchain and cryptocurrencies, smart contracts are a way to make the blockchain programmable. That is: a smart contract is a program that runs on the blockchain to extend its capabilities.

For the smart contract, the blockchain is just an abstract machine (database, if we prefer) with which it programmatically interacts. Basic design choices in the blockchain's design can affect the the smart contract programming paradigm which it naturally supports, and this can have far-reaching consequences: different programming paradigms are susceptible to different programming styles, and different kinds of program errors.

Thus, a decision in the blockchain's design can have lasting, unavoidable, and critical effects on its programmability. It is worth being very aware of how this can play out, not least because almost by definition, applications of smart-contracts-the-programming-paradigm tend to be safety-critical.

In this paper we will consider a simple but illustrative example of a smart contract: a fungible tradable token issued by an issuer who creates an initial supply and then retains control of its price—imitating a government-issued fiat currency, but run from a blockchain instead of a central bank.

We will put this example in the context of two major smart contract languages: Solidity, which runs on the Ethereum blockchain, whose native token is *ether*; and Plutus, which runs on the Cardano blockchain, whose native token

is *ada*.[3] We compare and contrast the blockchains in detail and exhibit their respective smart contracts. Both contracts run, but their constructions are different, in ways that illuminate the essential natures of the respective underlying blockchains and the programming styles that they support.

We will also see that the Ethereum smart contract is arguably buggy, in a way that flows from the underlying programming paradigm of Solidity/Ethereum. So even in a simple example, the essential natures of the underlying systems are felt, and with a mission-critical impact.

Definition 1.1. We will use Solidity and Plutus to code a system as follows:

1. An issuer `Issuer` creates some initial supply of a tradable token on the blockchain at some location in `Issuer`'s control; call this the *official portal*.
2. Other parties buy the token from the official portal at a per-token ether/ada *official price*, controlled by the issuer.[4]
   Once other parties get some token, they can trade it amongst themselves (e.g. for ether/ada), independently of the official portal and on whatever terms and at whatever price they mutually agree.
3. The issuer can update the ether/ada official price of the token on the official portal, at any time.
4. For simplicity, we permit just one initial issuance of the token, though tokens can be redistributed as just described.

## 2 Idealised EUTxO

### 2.1 The structure of an idealised blockchain

We start with a novel mathematical idealisation of the EUTxO (Extended UTxO) model on which Cardano is based [1, 2].

Notation 2.1. Suppose $X$ and $Y$ are sets. Then:

1. Write $\mathbb{N} = \{0, 1, 2, \dots\}$ and $\mathbb{N}_{>0} = \{1, 2, 3, \dots\}$.
2. Write $\mathit{fin}(X)$ for the finite powerset of $X$ and $\mathit{fin}_!(X)$ for the pointed finite powerset (the $(X', x) \in \mathit{fin}(X) \times X$ such that $x \in X'$).[5]
3. Write $\mathit{pow}(X)$ for the powerset of $X$, and $X \xrightarrow{\mathit{fin}} Y$ for finite maps from $X$ to $Y$ (finite partial functions).

Definition 2.2. Let the **types** of **Idealised EUTxO** be a solution to the equations in Figure 1.[6] For diagrams and examples see Example 2.6.

[3] Plutus and Cardano are IOHK designs. The CEO and co-founder of IOHK, Charles Hoskinson, was also one of the co-founders of Ethereum.

[4] Think: central bank, manufacturer's price, official exchange rate, etc.

[5] This use of 'pointed' is unrelated to the 'points to' of Notation 2.4.

[6] We write 'a solution of' because Figure 1 does not specify a unique subset for Validator. *Computable* subsets is one candidate, but our mathematical abstraction is agnostic to this choice. This is just the same as function-types not being modelled by *all* functions, or indeed as models of set theory can have different powersets (e.g. depending on whether a powerset includes the Axiom of Choice).

$$
\begin{aligned}
\mathsf{Redeemer} &= \mathsf{CurrencySymbol} = \mathsf{TokenName} = \mathsf{Position} = \mathbb{N} \\
\mathsf{Chip} &= \mathsf{CurrencySymbol} \times \mathsf{TokenName} \\
\mathsf{Datum} \times \mathsf{Value} &= \mathbb{N} \times (\mathsf{Chip} \xrightarrow{fin} \mathbb{N}_{>0}) \\
\mathsf{Validator} &\subseteq pow(\mathsf{Redeemer} \times \mathsf{Datum} \times \mathsf{Value} \times \mathsf{Context}) \\
\mathsf{Input} &= \mathsf{Position} \times \mathsf{Redeemer} \\
\mathsf{Output} &= \mathsf{Position} \times \mathsf{Validator} \times \mathsf{Datum} \times \mathsf{Value} \\
\mathsf{Transaction} &= fin(\mathsf{Input}) \times fin(\mathsf{Output}) \\
\mathsf{Context} &= fin_{!}(\mathsf{Input}) \times fin(\mathsf{Output})
\end{aligned}
$$

**Fig. 1.** Types for Idealised EUTxO

```
data Chip = MkChip
  { cSymbol  :: !CurrencySymbol
  , cName    :: !TokenName }

data Config = MkConfig
  { cIssuer                 :: !PubKeyHash
  , cTradedChip, cStateChip :: !Chip }

tradedChip :: Config → Integer → Value
tradedChip MkConfig{..} n = singletonValue cTradedChip n

data Action =
    SetPrice !Integer
  | Buy      !Integer

transition :: Config → State Integer → Action
           → Maybe (TxConstraints Void Void, State Integer)
transition c s (SetPrice p)                    -- ACTION: set price to p
  | p < 0      = Nothing                       -- p negative? ignore!
  | otherwise  = Just                          -- otherwise
      ( mustBeSignedBy (cIssuer c)             -- issuer signed?
      , s{stateData = p})                      -- set new price!
transition c s (Buy m)                         -- ACTION: buy m chips
  | m ≤ 0      = Nothing                       -- buy negative quantity? ignore!
  | otherwise  = Just                          -- otherwise
      ( mustPayToPubKey (cIssuer c) value'     -- issuer been paid?
      , s{stateValue = stateValue s − sold})   -- sell chips!
 where
  value' = lovelaceValueOf (m ∗ stateData s)   -- final value buyer pays
  sold   = tradedChip c m                      -- no. chips buyer gets

guarded :: HasNative s ⇒ Config → Integer → Integer
        → Contract s Text ()
guarded c n maxPrice =
  void $ withError $ runGuardedStep (client c) (Buy n) $
  λ_ p _ → if p ≤ maxPrice then Nothing else Just ()
```

**Fig. 2.** Plutus implementation of the tradable token

REMARK 2.3. 1. Think of $r \in \mathsf{Redeemer}$ as a key, required as a necessary condition by a validator (below) to permit computation.
2. A chip $c = (d, n)$ is intuitively a currency unit (£, \$, ...), where $d \in \mathsf{CurrencySymbol}$ is assumed to Gödel encode[7] some predicate defining a **monetary policy** (more on this in Remark 2.8(2)) and $n \in \mathsf{TokenName}$ is just a symbol (Ada has special status and is encoded as $(0, 0)$).
3. $\mathsf{Datum}$ is any data; we set it to be $\mathbb{N}$ for simplicity. A value $v \in \mathsf{Value}$ is intuitively a multiset of tokens; $v\,c$ is the number of $c$s in $v$.
   - We may abuse notation and define $v\,c = 0$ if $c \notin dom(v)$.
   - If $dom(v) = \{c\}$ then we may call $v$ a **singleton** value (note $v\,c$ need not equal 1). See line 10 of Figure 2 for the corresponding code.
4. A transaction is a set of inputs and a set of outputs (either of which may be empty). In a blockchain these are subject to consistency conditions (Definition 2.5); for now we just say its inputs 'consume' some previous outputs, and 'generate' new outputs. A context is just a transaction viewed from a particular input (see next item).
5. Mathematically a validator $V \in \mathsf{Validator}$ is the set of $\mathsf{Redeemer} \times \mathsf{Datum} \times \mathsf{Value} \times \mathsf{Transaction}$ tuples it validates *but* in the implementation we intend that $V$ is represented by code `V` such that
   - from `V` we cannot efficiently compute a tuple $t$ such that `V`$(t)$=`True`, and
   - from `V` and $t$, we can efficiently check if `V`$(t)$=`True`.[8]

   We use a *pointed* transaction (Notation 2.1(2)) because a $\mathsf{Validator}$ in an $\mathsf{Output}$ in a $\mathsf{Transaction}$ is always invoked by some particular $\mathsf{Input}$ in a later $\mathsf{Transaction}$ (see Definition 2.5(2&3)), and that invoking input is identified by using $\mathsf{Context}$. If $t \in V$ then we say $V$ **validates** $t$.

NOTATION 2.4. 1. If $tx = (I, O) \in \mathsf{Transaction}$ and $o \in \mathsf{Output}$, say $o$ **appears in** $tx$ and write $o \in tx$ when $o \in O$; similarly for an input $i \in \mathsf{Input}$. We may silently extend this notation to larger data structures, writing for example $o \in \mathit{Txs}$ (Definition 2.9(1)).
2. If $i$ and $o$ have the same position then say that $i$ **points to** $o$.
3. If $tx = (I, O) \in \mathsf{Transaction}$ and $i \in I$ then write $tx@i$ for the context $((I, i), O)$ obtained by pointing $I$ at $i \in I$.

DEFINITION 2.5. A **valid blockchain**, or just **blockchain**, of idealised EUTxO is a finite sequence of transactions $\mathit{Txs}$ such that:

1. Distinct outputs appearing in $\mathit{Txs}$ have distinct positions.
2. Every input $i$ in some $tx$ in $\mathit{Txs}$ points to a unique output in some earlier transaction — it follows from this and condition 1 of this Definition, that distinct inputs appearing in $\mathit{Txs}$ also have distinct positions.
   Write this unique output $\mathit{Txs}(i)$.
3. If $i = (p, k)$ appears in $tx$ in $\mathit{Txs}$ and points to an earlier output $\mathit{Txs}(i) = (p, V, s, v)$, then $(k, s, tx@i) \in V$ (@ from Notation 2.4(3)).

[7] Gödel encoding refers to the idea of enumerating a countable datatype (in some arbitrary way) so that each element is represented by a unique numerical index.
[8] The 'crypto' in 'cryptocurrency' lives here.

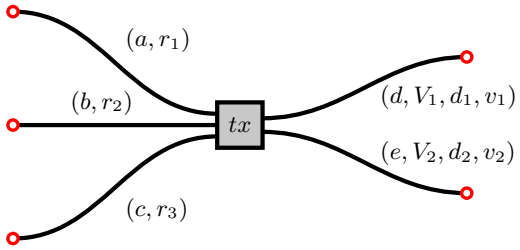


**Fig. 3.** A transaction $tx$ with three inputs and two outputs, positions $a, b, c, d, e$, redeemers $r_i$, validators $V_j$, data $d_j$ and values $v_j$

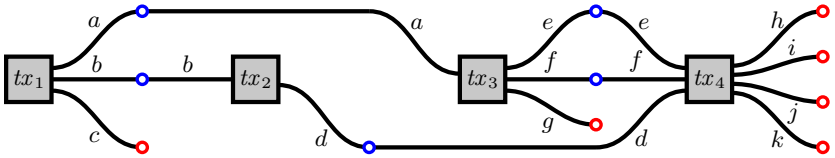


**Fig. 4.** A blockchain $\mathcal{B} = [tx_1, tx_2, tx_3, tx_4]$

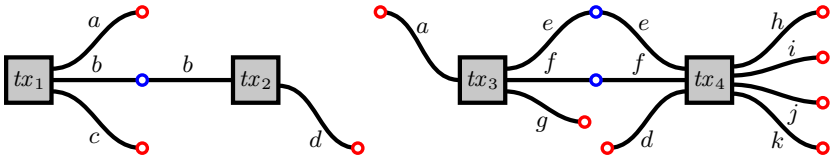


**Fig. 5.** $B$ chopped up as a blockchain $[tx_1, tx_2]$ and a chunk $[tx_3, tx_4]$

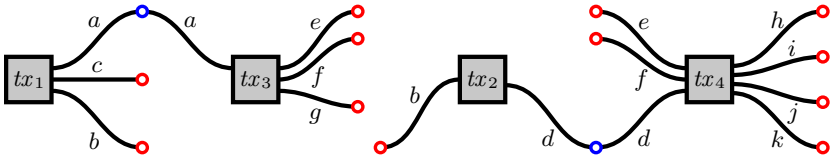


**Fig. 6.** $B$ chopped up as a blockchain $[tx_1, tx_3]$ and a chunk $[tx_2, tx_4]$

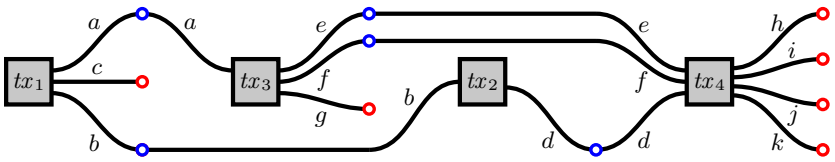


**Fig. 7.** The blockchain $\mathcal{B}' = [tx_1, tx_3, tx_2, tx_4]$

Example 2.6. Example transactions, blockchains, and chunks are illustrated in Figures 3, 4, 5, 6, and 7.

We leave it as an exercise to the reader to verify that: $\mathcal{B}$, $\mathcal{B}'$, $[tx_1, tx_2]$ and $[tx_1, tx_3]$ are blockchains; and $[tx]$, $[tx_3, tx_4]$ and $[tx_2, tx_4]$ are chunks but not blockchains. Also, e.g. $[tx_2, tx_1]$ is neither blockchain nor chunk, though it is a list of transactions, because the $b$-input of $tx_2$ points to the later $b$-output of $tx_1$.

Notation 2.7. *Txs* will range over finite sequences of transactions, and $\mathcal{B}$ will range over blockchains. We write $valid(Txs)$ for the assertion "*Txs* is a blockchain".

Remark 2.8. In words: *A sequence of transactions is a blockchain when every input points to an earlier validating output.* Note that output and inputs of a transaction may be empty, and this matters because any blockchain must contain a so-called *genesis block*, which is a transaction without inputs. We call Figure 1 and Definition 2.5 *Idealised EUTxO* because:

1. The implementation has bells and whistles which we omit for simplicity:
   (a) There is a second type of output for making payments to people.
   (b) Values represent 'real money' and so are preserved—the sum of input values must be equal to the sum of output values—unless they are not preserved, e.g. due to fees (which reduce the sum of outputs) or forging (creating) tokens (which increase it).
   (c) Transactions can be made time-sensitive using *slot ranges* (Remark 2.18); a transaction can only be accepted into a block whose slot is inside the transaction's slot range.

   All of the above is important for a working blockchain but for our purposes it would just add complexity.
2. We continue the discussion in Remark 2.3(2). If we consider a chip $c = (d, n)$, the currency symbol $d$ is not aritrary: it is (a Gödel encoding of) a *monetary policy* predicate. In the implementation, only transactions which satisfy all pertinent monetary policies are admissible.
   This is relevant to our example in Section 3 because we will assume a *state chip* with a monetary policy which enforces that it is affine (zero or one chips on the blockchain; see Remark 3.3). Explaining the mechanics of how this works is outside the scope of this paper; here we just note it exists.
3. In the implementation, a transaction is a pair of a set of inputs and a *list* of outputs. This is because an implementation concerns a run on a particular concrete blockchain $\mathcal{B}$, and we want to assign concrete positions for outputs in $\mathcal{B}$; so with a list the output located at the $j$th output of $i$th transaction could get position $2^i 3^j$.
   However, we care here about the theory of blockchains (plural). It is better to use sets and leave positions abstract, since if positions are fixed by their location in a blockchain then in Theorem 2.17 and the lemmas leading up to it, when we rearrange transactions in a blockchain (e.g. proving an observational equivalence) we could have to track an explicit reindexing of positions in the statement of the results. More on this in Subsection 2.3.

### 2.2 UTxOs and observational equivalence

We will be most interested in Definition 2.9 when $Txs$ and $Txs'$ are blockchains $\mathcal{B}$ and $\mathcal{B}'$. However, it is convenient for Lemma 2.15(1) if we consider the more general case of any finite sequences of transactions $Txs$ and $Txs'$:

Definition 2.9. 1. Call an output $o \in Txs$ **spent** (in $Txs$) when a later input points to it, and otherwise call $o$ **unspent** (in $Txs$).
2. Write $UTxO(Txs)$ for the set of unspent outputs in $Txs$.
3. If $UTxO(Txs) = UTxO(Txs')$ then write $Txs \approx Txs'$ and call $Txs$ and $Txs'$ **observationally equivalent**.

Notation 2.10. Given $Txs$ and a $tx$, write $Txs; tx$ for the sequence of transactions obtained by appending $tx$ to $Txs$. We will mostly care about this when $Txs$ and $Txs; tx$ are blockchains, and if so this will be clear from context.

Lemma 2.11. *Validity (Definition 2.5) is closed under initial subsequences, but not necessarily under final subsequences:*

1. $valid(Txs; tx)$ *implies* $valid(Txs)$.
2. $valid(tx; Txs)$ *does not necessarily imply* $valid(Txs)$.

*Proof.* For part 1: removing $tx$ cannot invalidate the conditions in Definition 2.5. For part 2: it may be that an input in $Txs$ points to an output in $tx$; if we then remove $tx$ we would violate condition 2 of Definition 2.5 that every input must point to a previous output.

A special case of interest is when two transactions operate on non-overlapping parts of the preceding blockchain:

Notation 2.12. Suppose $tx$ and $tx'$ are transactions. Write $tx\#tx'$ when the positions mentioned in the inputs and outputs of $tx$, are disjoint from those mentioned in $tx'$, and in this case call $tx$ and $tx'$ **apart**. Similarly for $tx\#Txs$.

Lemma 2.13. $tx\#tx' \iff tx'\#tx$.

*Proof.* An easy structural fact.

Remark 2.14. In Lemma 2.15 and Theorem 2.17 below, note that:

- $valid(\mathcal{B}; tx)$ (Notations 2.7 and 2.10) can be read as the assertion "it is valid to append the transaction $tx$ to the blockchain $\mathcal{B}$".
- $valid(\mathcal{B}; Txs)$ can be read as "it is valid to extend $\mathcal{B}$ with $Txs$".
- If $tx\#tx'$ then they mention disjoint sets of positions, so they cannot point to one another and the UTxOs they point to are guaranteed distinct.

Lemma 2.15. *1. If* $tx\#tx'$ *then we have* $valid(\mathcal{B}; tx; tx') \iff valid(\mathcal{B}; tx'; tx)$ *and* $\mathcal{B}; tx; tx' \cong \mathcal{B}; tx'; tx$.[9] *Intuitively: if two transactions are apart then it is valid to commute them.*

[9] We don't necessarily know $\mathcal{B}; tx; tx'$ is a blockchain, which is why we stated Definition 2.9 for sequences of transactions.

2. *If* $valid(\mathcal{B}; tx'; tx)$ *then* $valid(\mathcal{B}; tx) \iff tx\#tx'$. *(Some real work happens in this technical result, and we use this work to prove Theorem 2.17.)*

*Proof.* 1. By routine checking Definition 2.5.

2. Suppose $\neg(tx\#tx')$, so some position $p$ is mentioned by $tx$ and $tx'$. If $p$ is in an input in both $tx$ and $tx'$, or an output in both, then $valid(\mathcal{B}; tx'; tx)$ is impossible because each input must point to a unique earlier output, and each output must have a unique position. If $p$ is in an input in $tx$ and an output in $tx'$ then $\neg valid(\mathcal{B}; tx)$, because now this input points to a nonexistent output. If $p$ is in an output in $tx$ and an input in $tx'$ then $valid(\mathcal{B}; tx'; tx)$ is impossible, since each input must point to an *earlier* output.
   Conversely suppose $tx\#tx'$. Then $tx'$ must point only to outputs in $\mathcal{B}$ and removing $tx$ cannot disconnect them and so cannot invalidate the conditions in Definition 2.5.

Remark 2.16. Theorem 2.17 below gives a sense in which UTxO-based accounting is 'stateless'. With the machinery we now have the proof looks simple, but this belies its significance, which we now unpack in English:

Suppose we submit $tx$ to some blockchain $\mathcal{B}$. Then *either* the submission of $tx$ fails and is rejected (e.g. if some validator objects to it)—*or* it succeeds and $tx$ is appended to $\mathcal{B}$.

If it succeeds, then even if other transactions $Txs$ get appended first—e.g. they were submitted by other actors whose transactions arrived first, so that in-between us creating $tx$ and the arrival of $tx$ at the main blockchain, it grew from $\mathcal{B}$ to $\mathcal{B}; Txs$—then the result $\mathcal{B}; Txs; tx$ is *up to observational equivalence* equivalent to $\mathcal{B}; tx; Txs$, which is what *would* have happened *if* our $tx$ had arrived at $\mathcal{B}$ instantly and before the competing transactions $Txs$.

In other words: if we submit $tx$ to the blockchain, then at worst, other actors' actions might prevent $tx$ from getting appended, however, *if* our transaction gets onto the blockchain somewhere, then we obtain our originally intended result up to observational equivalence.

Theorem 2.17. *Suppose* $valid(\mathcal{B}; Txs; tx)$ *and* $valid(\mathcal{B}; tx)$. *Then*

1. $valid(\mathcal{B}; tx; Txs)$ *and*
2. $\mathcal{B}; tx; Txs \cong \mathcal{B}; Txs; tx$.

*Proof.* Using Lemma 2.15(2) $tx\#Txs$. The rest follows using Lemma 2.15(1).

Remark 2.18. The Cardano implementation has *slot ranges* (Remark 2.8(1c)). These introduce a notion of time-sensitivity to transactions which breaks Theorem 2.17(1), because $Txs$ might be time-sensitive. If we enrich Idealised EUTxO with slot ranges then a milder form of the result holds which we sketch as Proposition 2.19 below.

Proposition 2.19. *If we extend our blockchains with slot ranges, which restrict validity of transactions to defined time intervals, then Theorem 2.17 weakens to:*

*If* $valid(\mathcal{B}; tx; Txs)$ *and* $valid(\mathcal{B}; Txs; tx)$ *then* $\mathcal{B}; tx; Txs \cong \mathcal{B}; Txs; tx$.

*Proof.* As for Theorem 2.17, noting that slot ranges are orthogonal to UTxOs, provided the transactions concerned can be appended.

### 2.3 $\alpha$-equivalence and more on observational equivalence

Our syntax for $\mathcal{B}$ in Definition 2.5 is *name-carrying*; outputs are identified by unique markers which—while taken from $\mathbb{N}$; see "Position=$\mathbb{N}$"in Figure 1—are clearly used as *atoms* or *names* to identify binding points on the blockchain, to which at most one later input may bind. Once bound this name can be thought of graphically as an edge from an output to the input that spends it, so clearly the choice of name/position—once it is bound—is irrelevant up to permuting our choices of names. This is familiar from $\alpha$-equivalence in syntax, where e.g. $\lambda a.\lambda b.ab$ is equivalent to $\lambda b.\lambda a.ba$. We define:

DEFINITION 2.20. 1. Write $\mathcal{B} =_\alpha \mathcal{B}'$ and call $\mathcal{B}$ and $\mathcal{B}'$ **$\alpha$-equivalent** when they differ only in their choice of positions of spent output-input pairs (Definition 2.9(1)).[10] It is a fact that this is an equivalence relation.
2. If $\Phi$ is an assertion about blockchains, write "up to $\alpha$-equivalence, $\Phi$" for the assertion "there exist $\alpha$-equivalent forms of the arguments of $\Phi$ such that $\Phi$ is true of those arguments".

Lemma 2.21 checks that observational equivalence interacts well with being a valid blockchain and appending transactions. We sketch its statement and its proof, which is by simple checking. In words it says: *extensionally, a blockhain up to $\alpha$-equivalence is just its UTxOs*:

LEMMA 2.21. *1.* $\mathcal{B}\cong\mathcal{B}' \wedge valid(\mathcal{B};tx) \wedge valid(\mathcal{B}';tx)$ *implies* $\mathcal{B};tx\cong\mathcal{B}';tx$.
*2. If* $\mathcal{B} =_\alpha \mathcal{B}'$ *then* $\mathcal{B} \cong \mathcal{B}'$.
*3. Up to $\alpha$-equivalence, if* $\mathcal{B} \cong \mathcal{B}'$ *then* $valid(\mathcal{B};tx) \iff valid(\mathcal{B}';tx)$.
*4. Up to $\alpha$-equivalence, if* $\mathcal{B} \cong \mathcal{B}' \wedge valid(\mathcal{B};tx)$ *then* $\mathcal{B};tx \cong \mathcal{B}';tx$.

REMARK 2.22. We need $\alpha$-conversion in cases 3 and 4 of Lemma 2.21 because $valid(\mathcal{B};tx)$ might fail due to *accidental name-clash* between the position assigned to a spent output in $\mathcal{B}$, and a position in an unspent output of $tx$. We need $\alpha$-conversion to rename the bound position and avoid this name-clash.

This phenomenon is familiar from syntax, e.g. we know to $\alpha$-convert $a$ in $(\lambda a.b)[b{:=}a]$ to obtain (up to $\alpha$-equivalence) $\lambda a'.a$.

There are many approaches to $\alpha$-conversion: graphs, de Bruijn indexes [4], name-carrying syntax with an equivalence relation as required (used in this paper), the *nominal abstract syntax* of the second author and others [5], and the type-theoretic approach in [6]. Studying what works best for a structural theory of blockchains is future research.

In this paper we have used raw name-carrying syntax, possibly quotiented by equivalence as above; a more sophisticated development might require more. Note the implementation solution discussed in Remark 2.8(3) corresponds to a de Bruijn index approach, and for our needs in this paper, this does *not* solve all problems, as discussed in that Remark (see 'messy reindexing').

[10] Positions of unspent outputs (UTxOs) cannot be permuted. If we permute a UTxO position in $\mathcal{B}$, we obtain a blockchain $\mathcal{B}'$ with a symmetric equivalence to $\mathcal{B}$ but not observationally equivalent to it (much as $-i$ relates to $i$ in $\mathbb{C}$). More on this in [3].

## 3 The Plutus smart contract

Definition 3.1. Relevant parts of the Plutus code to implement Definition 1.1 are in Figure 2. Full source is at https://arxiv.org/src/2003.14271/anc.

Remark 3.2. 1. Chip corresponds to *Chip* from Figure 1.
2. Config stores configuration: the issuer (given by a hash of their public key), the chip traded, and the state chip (see Remark 3.3).
3. tradedChip packages up $n$ of cTradedChip in a value (think: a roll of quarters).
4. Action is a datatype which stores labels of a transition system, which in our case are either 'buy' or 'set price'.
5. State Integer corresponds to Datum in Figure 1. stateData s on line 29 retrieves this datum and uses it as the price of the token.
6. value' (lines 26 and 29) is the amount of ada paid to the Issuer.[11]

Remark 3.3 (State chip). In Remark 2.16 we described in what sense Idealised EUTxO (Figure 1) is stateless. Yet Definition 1.1(3) specifies that Issuer can set a price for the traded chip. This is seems stateful. How to reconcile this?

We create a *state chip* cStateChip, whose monetary policy (Remark 2.8(2)) enforces that it is *affine* and *monotone increasing*, and thus *linear* once created. The Issuer issues an initial transaction to the blockchain which sets up our trading system, and creates a unique UTxO that contains this state chip in its value, with the price in its Datum field.

The UTxO with the state chip corresponds to the *official portal* from Definition 1.1, and its state datum corresponds to the *official price*.

Monetary policy ensures this is now an invariant of the blockchain as it develops, and anybody can check the current price by looking up the unique UTxO distinguished by carrying precisely on cStateChip, and looking at the value in its Datum field.[12] The interested reader can consult the source code.

Remark 3.4 (Expressivity of off-chain code). The Plutus contract is a *state machine*, whose transition function transition is compiled into a Validator function, and so is explicitly *on-chain*. The function guarded lives *off-chain*; e.g. in the user's wallet. It can construct and send a Buy-transaction to (a UTxO with the relevant validator on) the blockchain, after checking that the price is acceptable.

If accepted, the effect of this transaction is independant of concurrent actions of other users, in senses we have made mathematically precise (cf. Remarks 2.16 and 4.4). This gives Plutus off-chain code a power that Solidity off-chain code cannot attain.

[11] We call it value because it directly corresponds with msg.value in Figure 9, whose name is fixed in Solidity. We add a dash to avoid name-clash with value, an existing function from the Plutus Ledger library.

[12] This technique was developed by the IOHK Plutus team.

# 4 The Solidity smart contract

## 4.1 Description

Remark 4.1. The Ethereum blockchain is account-based: it can be thought of as a state machine whose transitions modify a global state of contracts containing functions and data. We propose in Figure 8 a type presentation of Idealised Ethereum, parallel to the Idealised EUTxO of Figure 1. In brief:

1. Sender, Address, and Value are natural numbers $\mathbb{N}$. Datum is intended to be structured data which we Gödel encode for convenience. FunctionName and ContractName are names, which again for simplicity we encode as $\mathbb{N}$.
2. A contract has a name and a finite set of functions (Notation 2.1(2)).
3. A function has a name and maps an input to a blockchain transformer.
4. A blockchain is a finite collection of contracts, to each of which is assigned a state of a value (a balance of ether) and some structured data. We intend that a valid blockchain satisfy some consistency conditions, including that each contract in it have a distinct name.

Thus the contract Changing (line 1 of Figure 9), once deployed, is located on the Ethereum blockchain, with its functions and state. This contract and its state are 'in one place' on the blockchain; not spread out across multiple UTxOs as in Cardano. There is no need for the state-chip mechanism from Remark 3.3.

Remark 4.2. We now briefly read through the code:

1. address is an address for the Issuer. It is payable (it can receive ether) and public (its value can be read by any other function on the blockchain).[13]
2. constructor is the function initialising the contract. It gives issuer (who triggers the contract) all the new token.
3. send is a function to send money to another address. Note that this is not in the Plutus code; this is because we got it 'for free' as part of Cardano's in-built support for currencies (this is also why Idealised EUTxO has the type Value in Figure 1).
4. buy on line 20 is analogue to the Buy transition in Figure 2.

## 4.2 Discussion

Remark 4.3. In line 21 of Figure 9 we calculate the tokens purchased using a division of value (the total sum paid) by price.

In contrast, in line 29 of Figure 2 we calculate the sum by multiplying the number of tokens by the price of each token.

[13] Any data on the Ethereum blockchain is public in the external sense that it can be read off the binary data of the blockchain as a file on a machine running it. However, not all data is public in the internal sense that it can be accessed from any code running on the Ethereum virtual machine.

$$
\begin{aligned}
\mathsf{Sender} = \mathsf{FunctionName} = \mathsf{ContractName} = \mathsf{Datum} &= \mathsf{Value} = \mathbb{N} \\
\mathsf{Transaction} &= \mathsf{ContractName} \times \mathsf{FunctionName} \times \mathsf{Sender} \times \mathsf{Value} \times \mathsf{Datum} \\
\mathsf{Contract} &= \mathsf{ContractName} \times \mathit{fin}(\mathsf{Function}) \\
\mathsf{Function} &\subseteq \mathsf{FunctionName} \times \big((\mathsf{Sender} \times \mathsf{Value} \times \mathsf{Datum}) \to \mathsf{Blockchain} \to \mathsf{Blockchain}\big) \\
\mathsf{Blockchain} &= \mathsf{Contract} \xrightarrow{\mathit{fin}} (\mathsf{Value} \times \mathsf{Datum})
\end{aligned}
$$

**Fig. 8.** Types for Idealised Ethereum

```
contract Changing {
    address payable public issuer;                // issues the token
    uint public price;                            // current price
    mapping (address => uint) public balances;    // tracks who owns how many tokens

    constructor(uint _count, uint _price) public {
        require(_count > 0, "count must be positive");
        require(_price > 0, "price must be positive");
        issuer               = msg.sender;
        price                = _price;
        balances[msg.sender] = _count;
    }

    function send(address _receiver, uint _amount) public {
        require(_amount <= balances[msg.sender], "balance too low");
        balances[msg.sender] -= _amount;
        balances[_receiver]  += _amount;
    }

    function buy() public payable {
        uint _tokens = msg.value / price;
        require(_tokens <= balances[issuer], "not enough tokens");
        issuer.transfer(msg.value);
        balances[issuer]     -= _tokens;
        balances[msg.sender] += _tokens;
    }

    function setPrice(uint _newPrice) public {
        require(msg.sender == issuer, "only issuer can set price");
        price = _newPrice;
    }
}
```

**Fig. 9.** Solidity implementation of the tradable coin

There is a reason for this: we cannot perform the multiplication in the Solidity buy code because we do not actually know the price per token at the time the function acts to transition the blockchain. We can access a price at the time of invoking buy by querying Changing.price(), but by the time that invocation reaches the Ethereum blockchain and acts on it, the blockchain might have undergone transitions, and the price might have changed.

Because Ethereum is stateful and has nothing like Remark 2.16 and Theorem 2.17, this cannot be fixed.

Remark 4.4. One might counter that this could indeed be fixed, just by changing buy to include an extra parameter which is the price that the buyer expects to pay, and if this expectation is not met then the function terminates. However, the issuer controls the contract (issuer = msg.sender on line 9 of Figure 9), including the code for buy, so this safeguard can only exist at the *issuer's* discretion—and our issuer, whether through thoughtlessness or malice, has not included it.

In Cardano the buyer has more control, because by Theorem 2.17 a party issuing a transaction knows (up to observational equivalence) precisely what the inputs and outputs will be; the only question is whether it successfully attaches to the blockchain (cf. Remark 2.16 and Subsection 2.3).

Another subtle error in the Ethereum code is that the / on line 21 is integer division, so there may be a rounding error.[14]

Remark 4.5. The Ethereum code contains two errors, but the emphasis of this discussion is not that it is possible to write buggy code (which is always true) rather, we draw attention to how and why the underlying accounts-based structure of Ethereum invites and provides cover for certain errors *in particular*.

On the other hand, the Plutus code is more complex than the Ethereum code. Plutus is arguably conceptually beautiful, but the pragmatics as currently implemented are fiddly and the amount of boilerplate required to get our Plutus contract running much exceeds that required for our Ethereum contract. This may improve as Plutus matures and libraries improve—Plutus was streamlined as this paper was written, in at least one instance because of a suggestion from this paper[15]—but it remains to be seen if the gap can be totally closed.

So Ethereum is simple, direct and alluring, but can be dangerous, whereas Plutus places higher burdens on the programmer but enjoys some good mathematical properties which can enhance programmability. This is the tradeoff.

Remark 4.6 (Contracts on the blockchain). It is convenient to call the code referred to in Figures 2 and 9 *contracts*, but this is in fact an imprecise use of the term: a contract is an entity on the blockchain, whereas the figures

[14] It would be unheard of for such elementary mistakes to slip into production code; and even if it did happen, it is hardly conceivable that such errors would happen repeatedly across a wide variety of programming languages.

That was sarcasm, but the point may bear repeating: programmer error and programming language design are two sides of a single coin.

[15] —the need for runGuardedStep.

describe *schemas* or *programs* which, when invoked with parameters, attempt to push a contract onto the blockchain.

- For Plutus, each time we invoke the contract schema for some choice of values for Config, the schema tries to creates a transaction on the blockchain which generates a UTxO which carries a state-chip *and* an instance of the contract. The contract is encoded in the monetary policy of the state-chip (which enforces linearity once created) and in the Validator of that UTxO. The issuer's address is encoded in the Validator field of the UTxO. A mathematical presentation of the data structures concerned is in Figure 1.
- For Ethereum, Figure 9 also describes a schema which deploys a contract to the blockchain. Each time we invoke its constructor we generate a specific instance of Changing. Configuration data, given by the constructor arguments, is simply stored as values of the global state variables of that instance, such as issuer (line 2 of Figure 9).

Remark 4.7 (State & possible refinements). In Solidity, state is located in variables in contracts. To query a state we just query a variable in a contract of interest, e.g. price or balances in contract Changing in Figure 9.

In Plutus, state can be more expensive. State is located in unspent outputs; see the Datum and Value fields in Output in Figure 1. In a transaction that queries or modifies state, we consume any UTxOs containing state to which we refer, and produce new UTxOs with new state. Even if our queries are read-only, they destroy the UTxOs read and create new UTxOs.[16]

Suppose a UTxO contains some popular state information, so multiple users submit transactions referencing it. Only one transaction can succeed and be appended and the other transactions will *not* link to the new state UTxO which is generated—even if it contains the same state data as the older version. Instead, the other transactions will fail, because the particular incarnation of the state UTxO to which they happened to link, has been consumed.

This could become a bottleneck. In the specific case of our Plutus contract, it could become a victim of its own success if our token is purchased with high enough frequency, since access to the state UTxO and the tokens on it could become contested and slow and expensive in user's time. The state UTxO becomes, in effect, constantly locked by competing users—though note the the blockchain would still be behaving correctly in the sense that nobody can lose tokens.

For a more scaleable scenario, some library for parallel redundancy with a monetary policy for the state chip(s) implementing a consensus or merging algorithm, might be required. This is future work.

One more small refinement: every token ever bought must go through the contract at the official price—even if it is the issuer trying to withdraw it (recall from Definition 1.1(4) that we allow only one issuance). In practice, the issuer might want to code a way to get their tokens out without going through buy.

[16] This is distinct from a user inspecting the contents of UTxOs from outside the blockchain, i.e. by reading state off the hard drive of their node or Cardano wallet.

Remark 4.8 (Resisting DOS attacks). An aside contrasting how Cardano and Ethereum manage fees to prevent denial-of-service (DOS) attacks: In Cardano, users add transactions to the blockchain for a fee of $a + bx$ where $a$ and $b$ are constants and $x$ is the size of the transaction in bytes. This prevents DOS attacks because e.g. if we flood the blockchain with $n$ transactions containing tiny balances, we pay $\geq na$ in fees. In Ethereum, storage costs a fee (denominated in *gas*); if we inflate balances—which is a finite but unbounded mapping from addresses to integers—with $n$ tiny balances, we may run out of gas.

### 4.3 Summary of the critical points

1. The errors in the Solidity code are not necessarily obvious.
2. The errors cannot be defended against without rewriting the contract, which requires the seller's cooperation (who controls the contract and might even have planted the bug).
3. In Plutus, the *buyer* creates a transaction and determines its inputs and outputs. A transaction might be rejected if the available UTxOs change—but if it succeeds then the buyer knows the outcome. Thus if the user of a contract anticipates an error (or attack) then they can guard against it independently of the contract's designer.
   In Ethereum this is impossible: a buyer can propose a transition to the global virtual machine by calling a function on the Ethereum blockchain, but the designer of the contract designed that function—and because of concurrency, the buyer cannot know the values of the inputs to this function at the time of its execution on the blockchain.

## 5 Conclusions

We hope this paper will provide a precise yet accessible entry point for interested readers, and a useful guide to some of the design considerations in this area.

We have seen that Ethereum is an accounts-based blockchain system whereas Cardano (like Bitcoin) is UTxO-based, and we have implemented a specification (Definition 1.1) in Solidity (Section 4) and in Plutus (Section 3), and we have given a mathematical abstraction of Cardano, *idealised EUTxO* (Section 2). These have raised some surprisingly non-trivial points, both quite mathematical (e.g. Subsection 2.3) and more implementational (e.g. Subsection 4.2), which are discussed in the body of the paper.

The accounts-based paradigm lends itself to an imperative programming style: a smart contract is a *program* that manipulates a global state mapping global variables ('accounts') to values. The UTxO-based paradigm lends itself to a functional programming style: the smart contract is a *function* that takes a UTxO-list as its input, and returns a UTxO-list as its output, with no other dependencies. See Theorem 2.17 and the preceding discussion and lemmas in Subsection 2.2 for a mathematically precise rendering of this intuition, and Subsection 2.3 for supplementary results.

Smart contracts are naturally concurrent and must expect to scale to thousands, if not billions, of users,[17] but the problems inherent to concurrent imperative programming are well-known and predate smart contracts by decades. This is an issue with Ethereum/Solidity and it is visible even in our simple example. Real-life examples, like the infamous DAO hack[18], are clearly related to the problem of transactions having unexpected consequences in a stateful, concurrent environment.

Cardano's UTxO-based structure invites a functional programming paradigm, and this is reflected in Plutus. The price we pay is arguably an increase in conceptual complexity. More generally, in Remarks 4.6 and 4.7 we discussed how state is handled in Plutus and its UTxO model: further clarification of the interaction of state with UTxOs is important future work.

In summary: accounts are easier to think about than UTxOs, and imperative programs are easier to read than functional programs, but this ease of use makes us vulnerable to new classes of errors, *especially* in a concurrent safety-critical context. Such trade-offs may be familiar to many readers with a computer science or concurrency background.

## References

1. M. Chakravarty, J. Chapman, K. MacKenzie, O. Melkonian, M. P. Jones, and P. Wadler, "The extended UTXO model," in *Workshop on Trusted Smart Contracts (Financial Cryptography 2020)*, January 2020.
2. D. Coutts and E. de Vries, "A formal specification of the Cardano wallet," tech. rep., IOHK, July 2018. Version 1.2.
3. M. Gabbay, "Equivariant ZFA and the foundations of nominal techniques," *Journal of Logic and Computation*, 01 2020.
4. N. G. de Bruijn, "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem," *Indagationes Mathematicae*, vol. 5, no. 34, pp. 381–392, 1972.
5. M. J. Gabbay and A. M. Pitts, "A new approach to abstract syntax with variable binding," *Formal Aspects of Computing*, vol. 13, pp. 341–363, July 2001.
6. G. Allais, R. Atkey, J. Chapman, C. McBride, and J. McKinna, "A type and scope safe universe of syntaxes with binding: their semantics and proofs," *PACMPL*, vol. 2, no. ICFP, pp. 90:1–90:30, 2018.
7. A. Kiayias, A. Russell, B. David, and R. Oliynykov, "Ouroboros: A provably secure proof-of-stake blockchain protocol," in *Advances in Cryptology – CRYPTO 2017* (J. Katz and H. Shacham, eds.), (Cham), pp. 357–388, Springer International Publishing, 2017.
8. N. Atzei, M. Bartoletti, and T. Cimoli, "A Survey of Attacks on Ethereum Smart Contracts (SoK)," in *Principles of Security and Trust* (M. Maffei and M. Ryan, eds.), (Berlin, Heidelberg), pp. 164–186, Springer Berlin Heidelberg, 2017.

[17] How to reach distributed consensus in such an environment is another topic, with its own attack surface. Cardano uses Ouroboros consensus [7].

[18] The DAO hack stole approximately 70 million USD from Ethereum, which chose to revert the theft using a hard fork of the Ethereum blockchain [8].