An Abstract Model of UTxO-based Cryptocurrencies with Scripts

Joachim Zahnentferner
Input Output HK
Hong Kong
Email: chimeric.ledgers@protonmail.com

Abstract

In [1], an abstract accounting model for UTxO*-based cryptocurrencies has been presented. However, that model considered only the simplest kind of transaction (known in Bitcoin as* pay-to-pubkey-hash*) and also abstracted away all aspects related to authorization. This paper extends that model to the general case where the transaction contains validator (a.k.a.* scriptPubKey*) scripts and redeemer (a.k.a.* scriptSig*) scripts, which together determine whether the transaction's fund transfers have been authorized.*

1. Introduction

The *Chimeric Ledgers* paper [1] introduced abstract models for UTxO-based transactions and account-based transactions, as well as for ledgers that may contain both kinds of transactions, but authorization mechanisms were left aside. This paper is a short addendum that extends the model for UTxO-based cryptocurrencies with an abstract authorization mechanism that encompasses Bitcoin scripts.

The contributions of this paper are:

- Definition of an abstract model for UTxO-based cryptocurrencies with authorization scripts;
- Discussions of the relation between this abstract model and Bitcoin.

For the sake of self-containment, parts of sections 2 and 3 from the *Chimeric Ledgers* paper [1] are reproduced here in a summarized form. Nevertheless, readers unfamiliarized with the *Chimeric Ledgers* paper are encouraged to read it first.

Reference implementations of all concepts defined here are available in Scala (github.com/input-output-hk/chimeric-ledgers-spec-scala) and Haskell (github.com/input-output-hk/plutus-prototype/tree/master/docs/model/UTxO.hsproj).

2. Preliminaries

In the formalism presented here, the technical details of the underlying blockchain are disregarded, because only the ledger of transactions (i.e. the data stored in the blockchain) is of interest. A *ledger* is assumed to be a list of transactions.

Definition 1. A *ledger* is a list of valid transactions:

$$\mathsf{Ledger} \stackrel{\mathsf{def}}{=} \mathsf{List}[\mathsf{Transaction}]$$

Notations: A record data type with fields φ_1, ..., φ_n of types T_1, ..., T_n is denoted $(\varphi_1 \colon T_1, \ldots, \varphi_n \colon T_n)$. If t is a value of a record data type T and φ is the name of a field of T, then $t.\varphi$ denotes the value of φ for t. A list λ of type $\mathsf{List}[T]$ is either the empty list $[]$ or a list $e :: \lambda'$ with *head* e of type T and *tail* λ' of type $\mathsf{List}[T]$. $[e_1, \ldots, e_n]$ is an abbreviation for $e_1 :: \ldots :: e_n :: []$. $\lambda(i)$ denotes the i-th element of λ (with the head being the 0-th element, by convention). The concatenation of two lists λ_1 and λ_2 is denoted $\lambda_1 ::: \lambda_2$. The length of a list λ is denoted $|\lambda|$. A list of integers from n to m, including n and m, is denoted $[n..m]$. The standard equality symbol $(=)$ is used to state that two values are equal. The definitional equality symbol $(\stackrel{\mathsf{def}}{=})$ is used to define the new constant or function symbol on the left side. An anonymous function that takes a tuple as argument may be denoted as $(a_1, \ldots, a_n) \Rightarrow \ldots$. For instance, the k-th projection of a tuple may be denoted $(a_1, \ldots, a_n) \Rightarrow a_k$. A cryptographic collision-resistant hash of an object c is denoted $c^{\#}$.

3. The Model without Scripts

The definitions for UTxO-based cryptocurrencies, as presented in [1], are reproduced below without any change. Explanatory commentary has been omitted here, but can be found in [1].

Definition 2. The datatype for UTxO-*based transactions* is defined as:

$$\begin{aligned}\mathsf{UtxoTx} \stackrel{\text{def}}{=} (&\mathit{inputs}\colon \mathsf{Set}[\mathsf{Input}],\\ &\mathit{outputs}\colon \mathsf{List}[\mathsf{Output}],\\ &\mathit{forge}\colon \mathsf{Value}, \mathit{fee}\colon \mathsf{Value})\end{aligned}$$

The datatype for *outputs* is:

$$\mathsf{Output} \stackrel{\text{def}}{=} (\mathit{address}\colon \mathsf{Address}, \mathit{value}\colon \mathsf{Value})$$

where value is the value[1] of the output and $\mathit{address}$ is the address that owns it. The datatype for *inputs* is:

$$\mathsf{Input} \stackrel{\text{def}}{=} (\mathit{id}\colon \mathsf{Id}, \mathit{index}\colon \mathsf{Int})$$

where id is the id of a previous transaction to which this input refers, and index indicates which of the referred transaction's outputs should be spent.

Definition 3. The function $\mathrm{tx}\colon \mathsf{Input} \to \mathsf{Ledger} \to \mathsf{Option}[\mathsf{UtxoTx}]$, when applied to an input i and a ledger λ, retrieves a transaction t contained in λ such that $t^{\#} = i.id$, if such a t exists. The function $\mathrm{out}\colon \mathsf{Input} \to \mathsf{Ledger} \to \mathsf{Option}[\mathsf{Output}]$ returns $\mathrm{tx}(i).get.inputs(i.index)$, if this exists. And finally, the function $\mathrm{value}\colon \mathsf{Input} \to \mathsf{Ledger} \to \mathsf{Option}[\mathsf{Value}]$, returns $\mathrm{out}(i, \lambda).get.value$, if this exists.

Definition 4. The *unspent outputs* of a transaction can be computed by applying the following function:

$$\begin{aligned}&\mathrm{unspentOutputs}\colon \mathsf{UtxoTx} \to \mathsf{Set}[\mathsf{Input}]\\ &\mathrm{unspentOutputs}(t) \stackrel{\text{def}}{=} (\mathit{map}\\ &\qquad((o, i) \Rightarrow \mathsf{Input}(t^{\#}, i))\\ &\qquad t.outputs.zipWithIndex\\ &\quad).toSet\end{aligned}$$

where: $zipWithIndex$ augments the outputs with their respective indexes, the anonymous function maps an output to a spendable input consisting of the transaction's hash and the output's index, and $toSet$ converts the list to a set.

1. The types Address and Value are regarded here as aliases for unbounded unsigned non-negative integers.

The *outputs spent* by a transaction are simply the transaction's inputs, and can be computed by applying the following function:

$$\begin{aligned}&\mathrm{spentOutputs}\colon \mathsf{UtxoTx} \to \mathsf{Set}[\mathsf{Input}]\\ &\mathrm{spentOutputs}(t) \stackrel{\text{def}}{=} t.\mathit{inputs}\end{aligned}$$

Definition 5. The *set of unspent outputs* of a ledger can be computed by applying the following function:

$$\begin{aligned}&\mathrm{unspentOutputs}\colon \mathsf{Ledger} \to \mathsf{Set}[\mathsf{Input}]\\ &\mathrm{unspentOutputs}([]) \stackrel{\text{def}}{=} \emptyset\\ &\mathrm{unspentOutputs}(t::\lambda) \stackrel{\text{def}}{=} \mathrm{unspentOutputs}(\lambda)\\ &\qquad\qquad - \mathrm{spentOutputs}(t)\\ &\qquad\qquad + \mathrm{unspentOutputs}(t)\end{aligned}$$

Definition 6. A UTxO-based transaction t is *valid* for a ledger λ iff the following two conditions hold:

all inputs refer to unspent outputs:

$$\forall i \in t.\mathit{inputs}, i \in \mathrm{unspentOutputs}(\lambda)$$

value is preserved:

$$t.\mathit{forge} + \sum_{i \in t.\mathit{inputs}} \mathit{value}(i, \lambda).\mathit{get} = t.\mathit{fee} + \sum_{o \in t.\mathit{outputs}} o.\mathit{value}$$

Definition 7. The UTxO-*balance* of an address a in a valid transaction t w.r.t. a ledger λ is:

$$\mathcal{B}_{\text{UTxO}}\colon \mathsf{Address} \to \mathsf{UtxoTx} \to \mathsf{Ledger} \to \mathsf{Value}$$

$$\mathcal{B}_{\text{UTxO}}(a, t, \lambda) \stackrel{\text{def}}{=} \sum_{\substack{o \in t.\mathit{outputs}\\ o.\mathit{address}=a}} o.\mathit{value} - \sum_{\substack{i \in t.\mathit{inputs}\\ o'=\mathrm{out}(i,\lambda).\mathit{get}\\ o'.\mathit{address}=a}} o'.\mathit{value}$$

Definition 8. The UTxO-*balance* of an address a in a ledger λ is:

$$\begin{aligned}&\mathcal{B}_{\text{UTxO}}\colon \mathsf{Address} \to \mathsf{Ledger} \to \mathsf{Value}\\ &\mathcal{B}_{\text{UTxO}}(a, []) \stackrel{\text{def}}{=} 0\\ &\mathcal{B}_{\text{UTxO}}(a, t::\lambda) \stackrel{\text{def}}{=} \mathcal{B}_{\text{UTxO}}(a, \lambda) + \mathcal{B}_{\text{UTxO}}(a, t, \lambda)\end{aligned}$$

4. Bitcoin Transactions

In contrast to the model presented in the previous section, Bitcoin does not have a built-in notion of address. An output declares not an address to which it belongs, but rather a validator script that checks whether an input trying to spend the output's value

is authorized to do so. To prove that it indeed is authorized, an input specifies a redeemer script. To be faithful to Bitcoin, the definition of transaction could be modified as shown below:

Definition 9. The datatype for *Bitcoin-style* UTxO-*based transactions* is defined (as before) as:

$$\begin{aligned}\mathsf{BUtxoTx} \stackrel{\mathsf{def}}{=} (&\mathit{inputs}\colon \mathsf{Set}[\mathsf{Input}],\\ &\mathit{outputs}\colon \mathsf{List}[\mathsf{Output}],\\ &\mathit{forge}\colon \mathsf{Value}, \mathit{fee}\colon \mathsf{Value})\end{aligned}$$

The datatype for *outputs* is:

$$\mathsf{Output} \stackrel{\mathsf{def}}{=} (\mathit{validator}\colon \mathsf{Script}, \mathit{value}\colon \mathsf{Value})$$

where value is the value of the output and $\mathit{validator}$ is the script that checks that a redeemer is authorized to spend it. The datatype for *inputs* is:

$$\mathsf{Input} \stackrel{\mathsf{def}}{=} (\mathit{id}\colon \mathsf{Id}, \mathit{index}\colon \mathsf{Int}, \mathit{redeemer}\colon \mathsf{Script})$$

where id is the id of a previous transaction to which this input refers, index indicates which of the referred transaction's outputs should be spent and $\mathit{redeemer}$ is the script that provides evidence of authorization to spend the output.

Because the datatype for inputs is not just a reference to an output of a transaction anymore, the definitions of unspent outputs (definitions 4 and 5) would need to be changed as well. This issue is ignored in this section, but addressed in the next section.

In Bitcoin, the scripts are sequences of operations that manipulate a stack, which is initially empty. Some operations execute built-in cryptographic functions such as collision-resistant hashing and signature checking. Some operations may fetch state information from the ledger (e.g. block number) or from the transaction (e.g. a modified representation of the transaction for the purpose of checking signatures). The authorization succeeds if the execution of the redeemer script followed by the execution of the validator script leaves the boolean value true on the stack.

In an abstract model, the particularities of Bitcoin's scripting language can be left aside and the $\mathit{redeemer}$ and $\mathit{validator}$ scripts can be assumed to denote pure functions $[\![\mathit{redeemer}]\!] : \mathsf{State} \to \mathsf{R}$ and $[\![\mathit{validator}]\!] : \mathsf{State} \to \mathsf{R} \to \mathbb{B}$ where R is an arbitrary type, $\mathbb{B}$ is the type of booleans and State is the type for relevant state information about the ledger and the current transaction[2]. It is then possible to define validity as follows:

Definition 10. A bitcoin-style UTxO-based transaction t is *valid* for a ledger λ iff the two conditions of Definition 6 hold and additionally the following conditions hold:

no output is double spent:

$$|t.\mathit{inputs}| = |t.\mathit{ins}.\mathit{map}(i \Rightarrow (i.\mathit{id}, i.)|$$

all inputs validate:

$$\begin{aligned}&\forall i \in t.\mathit{inputs},\\ &[\![\mathrm{out}(i, \lambda).\mathit{get}.\mathit{validator}]\!](s, [\![i.\mathit{redeemer}]\!](s)) = \mathit{true}\end{aligned}$$

where s is the current state, which may depend on, and contain information about, λ and t.

In Definition 6, no explicit condition preventing double spending was needed because two inputs that try to spend the same output would be syntactically equal and, therefore, would occur only once in $t.\mathit{inputs}$, since $t.\mathit{inputs}$ is a $\mathsf{Set}[\mathsf{Input}]$. However, in Definition 10, it is needed, because $t.\mathit{inputs}$ may contain two inputs that try to spend the same output and differ from each other by having different[3] redeemer scripts.

Bitcoin's script language contains built-in functions for checking signatures, and the most common kind of validator-redeemer scripts, known as *pay-to-pubkey-hash*, simply check signatures. The redeemer script provides a public key and a signature of (a modified copy[4]) the transaction using the corresponding private key. The validator script contains a hash of the public key and it checks that this hash is equal to the hash of the public key provided by the redeemer script and verifies the signature.

It is the hash of the public key contained in the validator script that is commonly referred to as an address. However, this is just a convention and, in general, it is possible to define validator and redeemer scripts for which there is no natural notion of address.

2. For example, the state may contain the length of the current ledger. With such information it is possible to write a validator script that will allow an output to be spent before (or after) a certain "time" (measured in ledger length).

3. Two scripts are considered equal iff they are syntactically equal.

4. Signing modified copies is necessary, otherwise the creation of transactions would require solving a difficult fixpoint equation, since the unmodified transaction depends on the signature of itself. Modified copies are also useful to allow schemes such as *anyone-can-pay* [2].

For instance, the validator of an input could simply allow anyone to spend it, or allow no one to spend it.

The generality of Definition 9 makes it unclear to whom an output belongs. Consequently, it is unclear how to adapt the notion of balance defined in the previous section and ensure that wallets have a uniform way of accounting outputs. Perhaps the most intuitive solution to this problem would be to associate ownership of the output to the redeemer who is authorized to redeem and spend it. However, as Example 1 illustrates, this is problematic, because an output may be redeemable by several different redeemer scripts and hence the final owner of an output can only be known after it is spent.

Example 1. Consider the following transaction:

$$t \stackrel{\text{def}}{=} \mathsf{BUtxoTx}(\emptyset, [\mathsf{Output}(v, \$1000)], \$1000, \$0)$$

where the function denoted by the validator script v is such that:

$$[\![v]\!](s, r) \stackrel{\text{def}}{=} true$$

The output of this transaction can be spent by anyone with any redeemer script.

The problem of outputs without addresses is avoided in Bitcoin by considering outputs without conventional addresses non-standard. By convention, miners running a standard bitcoin client will not process transactions that contain non-standard outputs.

Besides pay-to-pubkey-hash, a second type of validator-redeemer script pair known as *pay-to-script-hash* is recognized as standard. Here, one of the components of the redeemer script is a serialized script; and the validator contains a hash value. To validate the expenditure, the validator script checks that the hash of the serialized script is equal to the hash value contained in the validator script, and then the serialized script is de-serialized and executed using the other components of the redeemer script as arguments. Interestingly, de-serialization is not a standard operation in the Bitcoin script language. There is no op-code that tells the bitcoin script interpreter to de-serialize the serialized script. It is an extra step that is done by convention since the acceptance of BIP-16 [3] and had to be hard-coded in the interpreter [4]. When the interpreter recognizes a validator script of a particular shape, it does the extra de-serialization step. As the BIP-16 itself admits, "recognizing one 'special' form of [validator script] and performing extra validation when it is detected is ugly".

Despite the ugliness, one positive aspect of pay-to-script-hash outputs is that again there is a natural notion of address: the hash of the serialized script, which is included in the validator script. Another interesting aspect is that the validator script does not reveal the validation conditions; instead, it only presents the hash of the serialized script, and it is the serialized script that contains the validation conditions. Since the serialized script becomes publicly known only when the output is spent, the pay-to-script-hash approach provides, at least temporarily, more privacy.

5. A Model with Scripts and Addresses

Recognizing the importance of addresses, this section presents an abstract UTxO-model that, unlike Bitcoin, has addresses as a built-in feature. Furthermore, inspired by the advantages of pay-to-script-hash, but wishing to avoid its "ugly" aspects, the model has both the validator and redeemer scripts in the input. A new datatype for output references is created in order to properly define unspent outputs, now that inputs are not just output references anymore.

Definition 11. The datatype for *script-address* UTxO-*based transactions* is defined (as before) as:

$$\begin{aligned}\mathsf{SUtxoTx} \stackrel{\text{def}}{=} (&\mathit{inputs}\colon \mathsf{Set}[\mathsf{Input}],\\ &\mathit{outputs}\colon \mathsf{List}[\mathsf{Output}],\\ &\mathit{forge}\colon \mathsf{Value}, \mathit{fee}\colon \mathsf{Value})\end{aligned}$$

The datatype for *outputs* is:

$$\mathsf{Output} \stackrel{\text{def}}{=} (\mathit{address}\colon \mathsf{Address}, \mathit{value}\colon \mathsf{Value})$$

where $value$ is the value of the output and $address$ is the address that owns it. The datatype for *output references* is:

$$\mathsf{OutputRef} \stackrel{\text{def}}{=} (\mathit{id}\colon \mathsf{Id}, \mathit{index}\colon \mathsf{Int})$$

where id is the id of a previous transaction to which this input refers, $index$ indicates which of the referred transaction's outputs should be spent. The datatype for *inputs* is:

$$\begin{aligned}\mathsf{Input} \stackrel{\text{def}}{=} (&\mathit{outputRef}\colon \mathsf{OutputRef}, \mathit{validator}\colon \mathsf{Script},\\ &\mathit{redeemer}\colon \mathsf{Script})\end{aligned}$$

where $validator$ is the script that checks that a redeemer is authorized to spend the output and $redeemer$ is the script that provides evidence of authorization to spend the output.

The definitions of unspent outputs are adapted to use output references instead of inputs.

Definition 12. The *unspent outputs* of a transaction can be computed by applying the following function:

$$\begin{aligned}&\mathrm{unspentOutputs}\colon \mathsf{SUtxoTx} \to \mathsf{Set}[\mathsf{OutputRef}]\\ &\mathrm{unspentOutputs}(t) \stackrel{\mathsf{def}}{=} (map\\ &\qquad((o,i) \Rightarrow \mathsf{OutputRef}(t^{\#},i))\\ &\qquad t.outputs.zipWithIndex\\ &\quad).toSet\end{aligned}$$

The *outputs spent* by a transaction are the output references included in the transaction's inputs, and can be computed by applying the following function:

$$\begin{aligned}&\mathrm{spentOutputs}\colon \mathsf{SUtxoTx} \to \mathsf{Set}[\mathsf{OutputRef}]\\ &\mathrm{spentOutputs}(t) \stackrel{\mathsf{def}}{=} t.inputs.map(i \Rightarrow i.outputRef)\end{aligned}$$

And, as before, the unspent outputs of a ledger can be defined inductively.

Definition 13. The *set of unspent outputs* of a ledger can be computed by applying the following function:

$$\begin{aligned}&\mathrm{unspentOutputs}\colon \mathsf{Ledger} \to \mathsf{Set}[\mathsf{OutputRef}]\\ &\mathrm{unspentOutputs}([]) \stackrel{\mathsf{def}}{=} \emptyset\\ &\mathrm{unspentOutputs}(t::\lambda) \stackrel{\mathsf{def}}{=} \mathrm{unspentOutputs}(\lambda)\\ &\qquad - \mathrm{spentOutputs}(t)\\ &\qquad + \mathrm{unspentOutputs}(t)\end{aligned}$$

To check the validity of a transaction, it is now necessary to additionally check that the address is equal to the hash of the validator script.

Definition 14. A script-address UTxO-based transaction t is *valid* for a ledger λ iff the conditions of Definition 6 hold and additionally the following conditions hold:

no output is double spent:

$$|t.inputs| = |t.ins.map(i \Rightarrow i.outputRef)|$$

all inputs validate:

$$\forall i \in t.inputs, [\![i.validator]\!](s, [\![i.redeemer]\!](s)) = true$$

where s is the current state, which may depend on, and contain information about, λ and t.

validator scripts hash to their output addresses:

$$\forall i \in t.inputs, i.validator^{\#} = \mathrm{out}(i,\lambda).get.address$$

Note that, in contrast to Bitcoin's pay-to-script-hash approach, here the validator script is not serialized.

With a sufficiently expressive scripting language, it is possible to implement pay-to-pubkey-hash as a special case of the pay-to-script-hash approach supported by the model defined in this section. Therefore, it is not necessary to define special input and output types for pay-to-pubkey-hash. A concise abstract model is desirable in theory. In practice, nevertheless, it may be advantageous to handle pay-to-pubkey-hash outputs and inputs as a special case, to save ledger space and transaction processing time.

Acknowledgments: Manuel Chakravarty, Duncan Coutts, Erik de Castro Lopo, Pablo Lamela Seijas, Darryl McAdams and Simon Thompson provided feedback and discussions that were useful in the production of this addendum.

References

[1] J. Zahnentferner, "Chimeric ledgers: Translating and unifying utxo-based and account-based cryptocurrencies," Cryptology ePrint Archive, Report 2018/262, 2018, https://eprint.iacr.org/2018/262.

[2] "OP_CHECKSIG," Bitcoin Wiki, January 2018. [Online]. Available: https://en.bitcoin.it/wiki/OP_CHECKSIG

[3] G. Andresen, "Pay to script hash," *Bitcoin Improvement Proposals*, 2012. [Online]. Available: https://github.com/bitcoin/bips/blob/master/bip-0016.mediawiki

[4] "Reference implementation of the bitcoin interpreter." [Online]. Available: https://github.com/bitcoin/bitcoin/blob/595a7bab23bc21049526229054ea1fff1a29c0bf/src/script/interpreter.cpp#L1375

-----BEGIN PGP MESSAGE-----

hQEMA3mYtjIcCbbOAQf/ci71Krwdlkd3ZzsoAkZdMKQYseQxI1YAxqEeshvnqcDJ
aCbKf5YXMdxEjumXW9EvTzjG8PnkLfviN9tpPWxmTujX0JcoiBkZ/CGSeO2msuhd
tC+W9xn5z0+Z7p4HZPRFfiZ4ZmlBow77JFRMcqqf/0G/eZ+7kkigECZzs9bBE4b+
kG3EF1268D69JVu6qleiyybF6U9D8OmDK4kRA3LuPyzqrE8sfdUDW8U0AuSUIsfY
YW8Jy6TdgikEDq6NzSS6X/jcjBq03XM3HJvA78l2XIooNSFKyoXWcJHTe9J+zXHe
krtnsI8Q/5US/C3FK/fX77k2gIKNgO8lYZQrQztBLNLA7AGYfZvDVc7WGliw8UJn
wPlJR6DUuOLvo1laej70hYHvLqFQFoLjS0KzM77uaftXWpEIc+ETY4eD6mEthBnz
kLraN9ZRhu3u8yunuIpBrSIpX50EXKwOA/aUhTvuR8ZH9GEMVyL3tpqxNbUFbSTp
lXElcCeE7TkcORLpV8E/NzPFk2TlazRl8iHGepLR7vw1DT66H3xN4+qcI2FwtEs8
5rDKq6NDkhPKNC8IsYcdtmNzdylCK2KbD3lBrEB3io1DvU2sD20a+NEsl8MeMknY
YW5GZ75y8W3YfL5/S0pVB6RqanpyZIHcq2P6wMYojpPhnPc500chOYGdLHeKwlJL
uEC3dE0Mg3LBtkBWQTiNXF/Nk02WS4YmTZJROI7oZnlGuoIZYDkj3r/WaJHTZd7K
7A7BhxJSiDbj5uC2F/cpBduHV6mtdpByv87wXWFlJMNFWcVow0x99gg1j/aCCToE
fjOl/sqpsxoo4jNqr9kREqpVYghoayxNy0fQYfew/aaP0p382gfKEbDeEhNfMV+f
Gq3MDJfGuUmfFyDnGLvr/fZu8o83elb446mwtBj+
=cj7k
-----END PGP MESSAGE-----