Self-Reproducing Coins as Universal Turing Machine

Alexander Chepurnoy[1,2], Vasily Kharin[3], Dmitry Meshkov[1]

[1] Ergo Platform
catena@protonmail.com
[2] IOHK Research
alex.chepurnoy@iohk.io
[3] Research Institute
v.kharin@protonmail.com

Abstract. Turing-completeness of smart contract languages in blockchain systems is often associated with a variety of language features (such as loops). In opposite, we show that Turing-completeness of a blockchain system can be achieved through unwinding the recursive calls between multiple transactions and blocks instead of using a single one. We prove it by constructing a simple universal Turing machine using a small set of language features in the unspent transaction output (UTXO) model, with explicitly given relations between input and output transaction states. Neither unbounded loops nor possibly infinite validation time are needed in this approach.

Keywords: smart contracts, Turing-completeness, blockchain, cellular automata

1 Introduction

Blockchain technology has become widely adopted after the introduction of Bitcoin by S. Nakamoto [10]. This peer-to-peer electronic cash ledger drew the enormous attention from the public, which resulted in rapid development of the technology and appearance of hundreds of alternative cryptocurrency projects. It also turned out that the blockchain applications expand quite far beyond the simple ledger niche. The rules of transaction validation can incorporate complicated logic, which is the essence of so-called smart contracts. In the case of Bitcoin the logic is implemented in the special-purpose Script language, which is believed not to be Turing complete. This belief stimulated the development of other smart contract platforms with the emphasis on the language universality. Particularly, in Ethereum [5] the *jump* opcode was introduced in a virtual machine assembly language in order to incorporate unlimited loops. In practice this resulted in various vulnerabilities and DoS attacks [3] since transaction computation cost (so-called *gas*) can only be calculated in runtime. Moreover, Turing-completeness of Ethereum is still a subject of debates mostly due to the undecidability of the halting problem in combination with a bounded block

validation time. The gas limit is often viewed as a fundamental component preventing Turing-completeness [9].

A Turing-complete programming language is a language which allows description of a universal Turing machine. A universal Turing machine is the Turing machine which can simulate any other Turing machine; its existence is one of the main results of the Turing theory [12]. The study of Turing machines is strongly motivated by the Church—Turing thesis, which states that Turing machines are capable of universal computation. The thesis is often viewed as a definition of computation and computability [13]. The set of known computation devices and models was rapidly growing during the twentieth century, and the methods of their analysis were improved as well. The usual way of proving the Turing-completeness of a system, a device or a language is about using it to emulate a system that is already proven to be Turing complete. A system which we are using in this work is one-dimensional cellular automaton Rule 110. It was conjectured to be Turing complete by S. Wolfram [16]. The conjecture was proven by M. Cook [7] based on previous works by E. L. Post [11].

The utter simplicity of Rule 110 makes it an appealing basis for proving Turing-completeness. In the present work we construct Rule 110 automaton algorithm for UTXO blockchain and implement it in Σ-State smart contract language[6]. We require neither loops, nor jump operator, nor recursive calls inside a transaction. Instead, we treat the computation as if it is occurring between the transactions (or maybe blocks). In this context transaction chaining and replication furnishes us with potentially infinite loops and recursion, while a combination of outputs for multiple transactions yields analog of a potentially infinite tape. The underlying idea of complexity growth is similar to the one expressed in [14,15].

This paper is structured as follows: in Section 2, we first describe a naive implementation of Rule 110 using a simple Bitcoin-like scripting language. Then we discuss the pitfalls arising from compliance with the blockchain properties, and show the way to overcome them. Section 3 describes the implementation for the real-world blockchains, and also sketches the discussion on the nature of computation in the framework of blockchain scripting and validation rules. In Appendix we describe the structure of a general-purpose guarding script for an output which can be transformed into an actual algorithm, along with the transformation procedure.

2 Rule 110 implementation

In this section we describe an implementation of Rule 110 cellular automaton. The automaton is transforming one-dimensional string of zeros and ones by applying evolution rules. One step of evolution for one bit is defined by its value c together with the values of the two neighboring bits — the left one ℓ and the right one r, along with a transition rule defined in Algorithm 1

Algorithm 1 Transition function of the Rule 110 automaton

```
1: function calcBit(ℓ, c, r)
2:     return (ℓ ∧ c ∧ r) ⊕ (c ∧ r) ⊕ c ⊕ r
3: end function
```

For the automaton implementation in a blockchain we use Bitcoin-like transactions consisting of inputs and outputs. Every output consists of a guarding *script* and a *payload*, while an input is a reference to an output from a previous transaction. We assume that the current state of the automaton is stored in the transaction output's *payload*. The general idea is to use the next transaction as a single step of the system evolution. In order to achieve this, two main conditions must be satisfied. First, the *payload* of at least one newly generated output should contain the updated state of the automaton. Second, this output must contain exactly the same script. These conditions require the transaction input to have access to the output's *script*s and *payload*s. It is implicitly present in the vast amount of existing blockchains, since in most cases scripts verify the signature of the spending transaction, which is constructed over the byte array containing the new outputs. However, this way of accessing output's data may be hardly exploitable. In the paper we assume that the guarding script of an input has direct access to the spending transaction outputs.

Keeping all these in mind, we come to the following validation script:

Algorithm 2 Script, that ensures that the transaction performs correct rule 110 transformation keeping the same rules for further iterations

```
 1: function validate(in, out)
 2:     function isRule110(inLayer, outLayer)
 3:         function procCell(i)
 4:             ℓ ← inLayer[i − 1 mod inLayer.size]
 5:             c ← inLayer[i]
 6:             r ← inLayer[i + 1 mod inLayer.size]
 7:             return calcBit(ℓ, c, r)
 8:         end function
 9:         return outLayer = inLayer.indices.map(procCell)
10:     end function
11:     return isRule110(self.payload, out[0].payload) ∧ (self.script = out[0].script)
12: end function
```

The script performs two checks. First, it takes the payload of a current input and ensures, that the result of Rule 110 application equals to the payload of the first output. Second, it checks that the guarding script of the first output is the same as a script of the input. The full implementation of this script in the smart contract language of an existing UTXO blockchain Ergo is provided at [2].

With this script, the cellular automaton evolution may be started by chaining transactions in a blockchain. Fig. 1 shows three transactions (on the left), each one representing the iteration of the automaton (on the right).

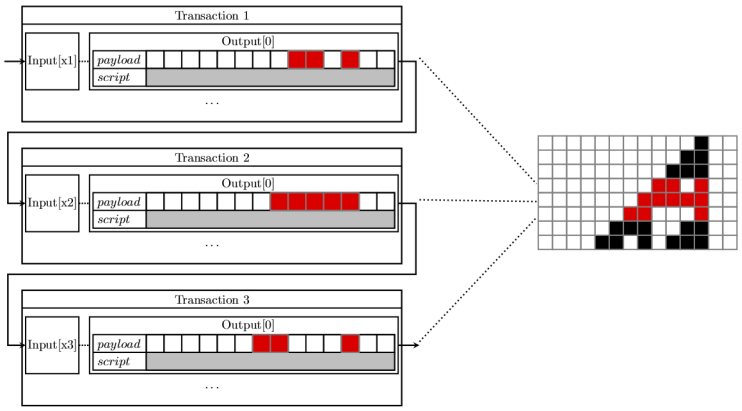

Fig. 1. Transaction chain following Rule 110. See Alg. 2 for the *script* field description.

Potentially infinite evolution of a cellular automaton, which is required for Turing-completeness, can be modeled by chaining potentially infinite number of transactions in the blockchain. However, there is a pitfall left. The size of the data stored in output must have an upper-bound, and validation time for a transaction must be bounded as well, otherwise blockchain is losing its security properties [4].

The natural workaround is to split the automaton state between transactions once it becomes too large. As an extreme case one can make a transaction output play a role of a single bit of the automaton. While being inefficient, this implementation keeps the logic simple and complies with the requirements of the blockchain and of potentially infinite evolution in space and time. The pseudocode of the corresponding script is provided in the Algorithm 3 and its implementation in Σ-State contract language is provided at [1]. Fig. 2 schematically shows the sequence of transactions (on the left), that corresponds to some area evaluation (on the right) of the automaton run.

[4] For example, in the Bitcoin backbone protocol model from [8], block validation should happen within finite and a-priori known round duration.

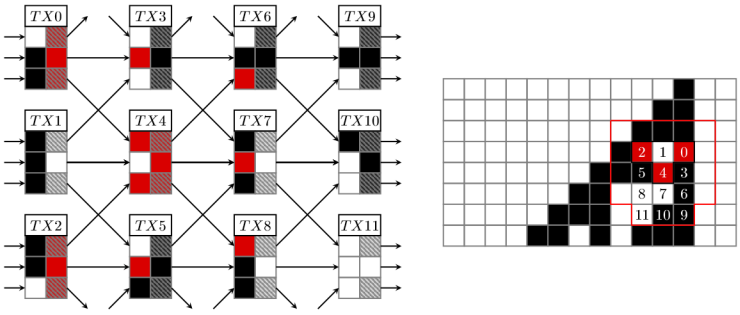

Fig. 2. Evolution of the cellular automaton described in Alg. 3. Every non-boundary transaction spends three outputs, and generates three new ones with identical bit values. Hatching indicates "mid" flag being unset. Numbers in the cells on the right pane correspond to the transaction numbers on the left.

Algorithm 3 Validation script for the output representing the single bit, and the unbound grid

```
1: function VERIFY(in, out)                    ▷ "in" and "out" are lists of inputs and outputs
2:     function OUTCORRECT(out, script)                                  ▷ output structure check
3:         scriptCorrect ← out[0].script = script
4:         isCopy1 ← out[1] = out[0].copy(mid← true)
5:         isCopy2 ← out[2] = out[0].copy(mid← false)
6:         return (¬out[0].mid) ∧ scriptCorrect ∧ isCopy1 ∧ isCopy2
7:     end function
8:     function CORRECTPAYLOAD(in, out)                                  ▷ output payload check
9:         ▷ mid flag is only set for the middle input
10:        inMidCorrect ← in[1].mid ∧ ¬(in[0].mid ∨ in[2].mid)
11:        ▷ input positions are correct; n is the index of leftmost column
12:        inYCorrect ← (in[1].n = in[0].n) ∧ (in[2].n = in[0].n)
13:        inXCorrect ← (in[1].x = in[0].x+1) ∧ (in[2].x = in[1].x+1)
14:        ▷ bits satisfy Rule 110
15:        inValCorrect ← out[0].val=CALCBIT(in[0].val, in[1].val, in[2].val)
16:        ▷ output position matches the input one
17:        outPosCorrect ← out[0].x = in[1].x ∧ (out[0].n = in[0].n−1)
18:        return inValCorrect ∧ inXCorrect ∧ inYCorrect ∧
               inMidCorrect ∧ outPosCorrect ∧ in.size=out.size=3
19:    end function
20:    if in[0].x=in[0].n ∧ in.size=1 then          ▷ leftmost — add 2 zeros to the left
21:        middle ← in[0].copy(x←in[0].n−1, val←0, mid← true)
22:        left ← in[0].copy(x←in[0].n−2, val←0, mid← false)
23:        realIn ← left ++ middle ++ in
24:    else if in[0].x=in[0].n ∧ in.size=2 then ▷ next to leftmost — add 0 to the left
25:        left ← in[0].copy(x←in[0].n−1, val←0, mid← false)
26:        realIn ← left ++ in
27:    else if in[0].x=−1 ∧ in.size=2 then          ▷ rightmost — add 0 to the right
28:        right ← in[0].copy(x← 1, val←0, mid← false)
29:        realIn ← in ++ right
30:    else                                                              ▷ normal cell
31:        realIn ← in
32:    end if
33:    return CORRECTPAYLOAD(realIn, out) ∧ OUTCORRECT(out, in[0].script)
34: end function
```

The script works as follows. Every output's payload contains its bit value val, the column index x, and the minimal x index at the current step n. As the grid expands by one at every step, $-n$ also serves as the row number. By default, the transaction spends three inputs (corresponding to the three neighboring bits from the previous row), and creates three outputs with the same bit value by the automaton rule. One output flagged by mid is supposed to be spent for new value with the column number x, and another two — for the columns $x \pm 1$ (see Fig. 2). In case the transaction creates the boundary cells, either one or two inputs are emulated to have zero bit values (lines 20–32). The overall validation script checks the correctness of the positions of inputs (lines 12 and 13) and outputs (line 17), correspondence of of the bit values (line 15), the correctness of the mid flag assignment for inputs (line 10) and the fact that all outputs are identical except the mid flag, which is set only once (lines 2–7).

Since the Turing-completeness of Rule 110 was proven in [7], we conclude that even though the scripting language itself does not allow loops, Turing-completeness of the system can be achieved by combining multiple transactions together. Note that our language requirements are not very demanding, just about bit operations, comparisons, assignments, and by-index access.

3 Discussion

The crucial move in our work is unwinding recursive calls by means of transaction chaining, although the language we use contains neither cycles nor recursion. By doing this we let a program to be executed over a sequence of transactions and blocks. This approach allows us to run programs in potentially infinite time on top of the blockchain while there is a strict upper-bound for block validation time.

A single transaction in the blockchain approximately corresponds to a single step of a computing machine. The step may be as complex as language built-ins allow; however, for security reasons it should be possible to estimate its running time before the actual evaluation.

One can wonder how evolving data structures (a blockchain and a corresponding UTXO set) along with programmable validation rules constitute a Turing machine. Obviously, we do need to include clients, forming transactions, and honest majority of miners, including transactions into blocks, as a component of the machine as well — their efforts are making the input tape of the machine. The same is true for Ethereum and other blockchains with smart contracts: the blockchain as a data structure does not endorse any computations — they should be initialized by a client.

Our approach can be used for Turing-completeness proofs of various smart contract languages in general. For example, it might be possible to prove that smart contracts of Waves platform [4] are actually Turing complete, although the authors claimed the opposite. Rule 110 implementation is not required for practical cases, it just guarantees that any algorithm can be potentially implemented. Despite existence of this guarantee, efficient usage of self-reproducing coins in

practice may require new machinery, including development environments and high-level smart contract languages for the multiple-transactions computations.

Acknowledgments

Authors thank Manuel Chakravarty, Oksana Klimenko, and Georgy Meshkov for the discussions and helpful comments on early drafts of this paper.

References

1. One bit per output rule 110 implementation in σ-state smart contract language, `https://git.io/vj6rX`
2. One layer per output rule 110 implementation in σ-state smart contract language, `https://git.io/vj6sw`
3. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on ethereum smart contracts (sok). In: International Conference on Principles of Security and Trust. pp. 164–186. Springer (2017)
4. Begicheva, A., Smagin, I.: Ride: a smart contract language for waves. Apograf.io (2018), `https://apograf.io/articles/14027`, v1.1
5. Buterin, V., et al.: A next-generation smart contract and decentralized application platform. white paper (2014), `https://git.io/vj6X9`
6. Chepurnoy, A.: σ-state authentication language, an alternative to bitcoin script. In: International Conference on Financial Cryptography and Data Security. pp. 644–645. Springer (2017)
7. Cook, M.: Universality in elementary cellular automata. Complex systems **15**(1), 1–40 (2004)
8. Garay, J., Kiayias, A., Leonardos, N.: The bitcoin backbone protocol: Analysis and applications. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 281–310. Springer (2015)
9. Miller, A.: Ethereum isn't turing complete and it doesn't matter anyway. IC3 NYC Blockchain Meetup, talk (2016), `https://www.youtube.com/watch?v=cGFOKTm_8zk`
10. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008)
11. Post, E.L.: Formal reductions of the general combinatorial decision problem. American journal of mathematics **65**(2), 197–215 (1943)
12. Turing, A.M.: On computable numbers, with an application to the entscheidungsproblem. Proceedings of the London mathematical society **2**(1), 230–265 (1937)
13. Turing, A.M.: Systems of logic based on ordinals. Proceedings of the London mathematical society **2**(1), 161–228 (1939)
14. Von Neumann, J.: The general and logical theory of automata. Cerebral mechanisms in behavior **1**(41), 1–2 (1951)
15. Von Neumann, J., Burks, A.W., et al.: Theory of self-reproducing automata. IEEE Transactions on Neural Networks **5**(1), 3–14 (1966)
16. Wolfram, S.: Theory and applications of cellular automata: including selected papers 1983-1986. World scientific (1986)

Appendix

This section addresses a question of guarding script conversion into the procedure being executed by a client or a miner. Note that the guarding script itself does not explicitly prescribe the course of computational actions needed to produce a valid transaction. It rather describes the algorithm of telling whether the result of the actions is correct or not. As an example, one could set a guarding script in the form $5^{out[0].x} \bmod 23 = 13$. This script structure is admissible, but it is hard to say that it describes an actual program of discrete logarithm calculation. In our particular case the solution is simple. If the guarding script is of the form $(out[0].x = f(in)) \wedge (something)$ with f being some function, then in order to satisfy the condition one can replace the equality check with a variable assignment. Hence if we require the script to be conjunction of equality checks containing the fields of the outputs solely on the left hand sides, and functions of the inputs on the right hand sides, then it actually defines the program (assuming that the inputs are fixed). It is fully present in the Alg. 2. Another problem is collecting the right set of inputs for the transaction. Suppose one wants to spend $in[0]$. If the condition for $in[1]$ is conjunction of the expressions of type $in[1].x = f(in[0])$, then finding the suitable $in[1]$ is the lookup over the possible inputs with field x being the key. Therefore, if the guarding script can be represented in the form

$$\Big(\bigwedge_i \bigwedge_j (out[i].x_j = f_{ij}(in))\Big) \wedge$$
$$(\bigwedge_i in[1].x_i = g_{1i}(in[0])) \wedge (\bigwedge_i in[2].x_i = g_{2i}(in[0], in[1])) \wedge \ldots, \qquad (1)$$

it can be efficiently converted to the transaction generation algorithm:

Algorithm 4 Transaction creation algorithm

```
1: for in[0] ← UTXO do
2:     i ← 0
3:     while scripts of in[0]...in[i] have rule g() for in[i + 1] do
4:         in[i+1] ← UTXO(g(in[0]...in[i]))
5:         i ← i+1
6:     end while
7:     j ← 0
8:     while scripts of in[0]...in[i] have rule f() for out[j] do
9:         out[j] ← f(in[0]...in[i])
10:        j ← j+1
11:    end while
12:    if tx(in,out).isValid then return tx
13:    end if
14: end for
```

Here the last if-statement is the consistency check. Note that both Alg. 2 and 3 can be represented as the desired form (1) with the length checks.