International Journal of Computer Trends and Technology
ISSN: 2231 – 2803 / https://doi.org/10.14445/22312803/IJCTT-V69I10P107

Volume 69 Issue 10, 43-46, Oct, 2021

Original Article

Bitcoin and the Theory of Computation

Martin V. Sewell

Cambridge, United Kingdom

Received Date: 13 September 2021
Revised Date: 15 October 2021
Accepted Date: 28 October 2021

***Abstract* —** *Bitcoin is assessed from the perspective of the theory of computation. Specifically, the computational power of Bitcoin is determined in the context of both computability theory and automata theory. In computability theory, there exists a hierarchy of functions, from the most powerful to the least powerful: partial recursive, total recursive, primitive recursive, elementary recursive, and lower elementary recursive. Whilst in automata theory, there exists a hierarchy of automata, from the most to least powerful: Turing machine, linear bounded automaton, non-deterministic pushdown automaton, deterministic pushdown automaton, and finite automaton. In both instances, Bitcoin lies within or below the least powerful category. Bitcoin is essentially a finite automaton that employs a scripting language for data manipulation that is even less powerful than a lower elementary recursive programming language. Bitcoin is not Turing-complete, either in whole or in part. For security reasons, Bitcoin was designed to be only as powerful as necessary.*

Keywords — *Automata theory, Bitcoin, Computability theory, Theory of computation, Turing-complete.*

I. INTRODUCTION

The goal of this paper is to assess Bitcoin[1] in the context of the theory of computation. The power of Bitcoin shall be assessed in terms of both computability theory and automata theory. Computability theory allows us to classify programming languages in terms of the set of functions that they are able to calculate. Whilst automata theory can be used to classify automata by the class of formal languages that they are able to recognize.

The remainder of the paper proceeds as follows. In Section II, we utilize computability theory and, in the context of Bitcoin, consider five classes of functions in turn, from the least to the most powerful: lower elementary recursive, elementary recursive, primitive recursive, total recursive, and partial recursive. The data-manipulation rules employed by Bitcoin are even less powerful than the lower elementary recursive functions. Whilst in Section III, we utilize automata theory and, in the context of Bitcoin, consider five classes of automata, from the least to the most powerful: finite automaton, deterministic pushdown automaton, non-deterministic pushdown automaton, linear bounded automaton, and Turing machine. Bitcoin may be considered a finite automaton. In Section IV, we conclude.

This article was written in part whilst working for nChain Limited.

II. COMPUTABILITY THEORY

Computability theory allows us to classify data-manipulation rules (e.g., programming languages) in terms of the set of functions that they are able to calculate. When analyzing the power of Bitcoin in the context of computability theory, we are concerned with the data-manipulation rules within the Bitcoin client that govern Bitcoin transactions. Bitcoin uses a scripting system for processing transactions known as Script. Script is similar to Forth, and is a simple, stack-based language processed from left to right. In programming languages, a loop provides a way of repeating instructions. The two most common loops are the while loop and the for loop, but a for loop is merely syntactic sugar for a while loop, supporting a subset of the use cases that while supports. To avoid the possibility of infinite loops, which would consume all available processor time and can cause a computer to hang, Script does not include any statements that enable loops. Recursion is equivalent to a loop plus a stack, so Script also has no recursion. Therefore with Script, each instruction is executed at most once in a linear manner, and any program will always halt.

Of course, with Script, if one wishes to emulate looping through a piece of code n times, one can simply repeat the code n times. One can even repeat the code m times and use if-then statements such that the code will only be repeated n ($\leq m$) times, where the stack initially contains n.However, this is not emulating loops in the proper sense because the code should produce the correct result for all n, not just $n \leq m$.

We can classify Script in terms of the set of functions that the language is able to calculate. The computability hierarchy of functions includes partial recursive, total recursive, primitive recursive, elementary recursive, and lower elementary recursive functions and can be described by the nested subsets shown in Fig. ***1***. Bitcoin Script is included and is a proper subset of the lower elementary recursive functions. All functions work over the natural numbers.

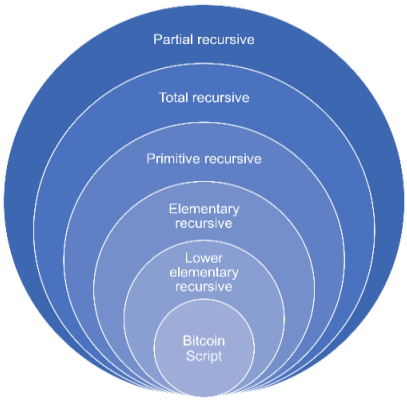

Fig. 1 Computability hierarchy of functions

A. Lower Elementary Recursive Functions

The lower elementary recursive functions are constructed using the following basic functions[2]:

- The *zero function* returns zero, $f(x) = 0$.
- The *successor function*, $f(x) = x + 1$. Via the repeated application of a successor function, one can achieve addition.
- *Projection functions* are used for ignoring arguments, for example $f(a, b) = a$.
- The *subtraction function*, $f(x, y) = x \dot{-} y$ if $y<x$, or 0 if $y \geq x$, is used to define conditionals and iteration.
- *Composition* involves applying values from some elementary recursive function as an argument to another elementary recursive function. If h is elementary recursive and each g_i is elementary recursive, $f(x_1, \ldots, x_n) = h(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n))$ is elementary recursive.
- *Bounded summation*: if g is elementary recursive, $f(m, x_1, \ldots, x_n) = \sum_{i=0}^{m} g(i, x_1, \ldots, x_n)$ is elementary recursive.

The lower elementary recursive functions have polynomial growth. Because none of the functions in Script has time complexity greater than polynomial, in principle, the lower elementary recursive functions can perform everything that Script can perform. In order to perform bounded summation, one needs finite loops of the form 'for *i* = 1 to *n* do…' where *n* (the number of times the loop executes) is fixed in advance (before the loop starts), and you cannot change *i* or *n* inside the loop. Lacking loops, it is clear that Script is unable to perform bounded summation. Therefore, the functions that Script can perform are a proper subset of the functions that may be performed by the lower elementary recursive functions. Script is thus less powerful than any of the five sets of functions defined.

B. Elementary Recursive Functions

The elementary recursive functions are functions that can be obtained from addition, multiplication, subtraction, and division, using basic operations such as substitutions and finite summation and product. The definitions of elementary recursive functions are the same as for the lower elementary recursive functions, with the addition of the bounded product.

- *Bounded product*: if g is elementary recursive, $f(m, x_1, \ldots, x_n) = \prod_{i=0}^{m} g(i, x_1, \ldots, x_n)$ is elementary recursive.

The *LOOP* programming language is a core imperative language in which programs consist only of assignments, sequences, and bounded loops.[3]The elementary functions are characterized by programs written in LOOP in which the nesting of for loops is restricted to a depth of at most 2.

C. Primitive Recursive Functions

The primitive recursive functions are the functions that can be computed by Turing machines that always halt and contain no infinite loops. The definition of primitive recursive functions is the same as for elementary recursive functions, except that bounded summation and bounded product are replaced by primitive recursion.

- Primitive recursion is such that, given f, a k-ary primitive recursive function, and g, a $(k+2)$-ary primitive recursive function, the $(k+1)$-ary function h is defined as the primitive recursion of f and g, i.e. the function h is primitive recursive when$h(0, x_1, \ldots, x_k) = f(x_1, \ldots, x_k)$ and$h(S(y), x_1, \ldots, x_k) = g(y, h(y, x_1, \ldots, x_k), x_1, \ldots, x_k)$.

Primitive recursive programming languages include LOOP[3] and BlooP[4].

D. Total Recursive Functions

The total recursive functions are the set of functions that can be computed by Turing machines that always halt. An example of a total computable function that is not primitive recursive is the Ackermann function.[5] One common version, with nonnegative integers n and m, is defined as follows:

$A(0,n) = n+1$,
$A(m+1,0) = A(m,1)$ and
$A(m+1,n+1) = A(m,A(m+1,n))$.

The halting problem is a decision problem that can be stated as follows: given the description of an arbitrary program and a finite input, decide whether the program will halt or run forever. Alan Turing proved in 1936 that a general algorithm (running on a Turing machine) to solve the halting problem for all possible program-input pairs could not exist[6, 7]. The halting problem is undecidable. Assume that we have a programming language that captures exactly the total recursive functions. It must be well defined, so we can construct an algorithm that takes an arbitrary function and a finite input and decides whether or not the function can be implemented using the programming language. This contradicts the fact that the halting problem is undecidable.

Therefore a programming language that captures exactly the total recursive functions cannot exist.

E. Partial Recursive Functions

The partial recursive functions are the set of functions that can be computed by Turing machines. A Turing machine is a mathematical model of a hypothetical computing machine that manipulates symbols on a strip of tape according to a predefined set of rules. We already know that Script cannot calculate all of the partial recursive functions, so it is not Turing-complete. However, it is straightforward to prove it directly. Consider a Turing machine defined by a two-symbol alphabet {0, 1} and one state {A}, with the rules: when reading 0, write 0 and move the tape one cell right and stay in state A; when reading 1, write 1 and move the tape one cell right and stay in state A. It should be clear that such a program never halts. A programming language is said to be Turing-complete if it can be used to simulate any single-taped Turing machine. As any program written in Script always halts (even with unbounded memory), it is clear that Script cannot simulate the described Turing machine. Therefore, Script is not Turing-complete.

F. Discussion

To conclude this section, Script cannot implement loops, so cannot calculate all of the lower elementary recursive functions, and is not Turing-complete. More generally, because Bitcoin's data-manipulation rules ensure that the system always halts, Bitcoin is not Turing-complete.

The script is unusual in not being able to implement loops. Microsoft's language Bosque has no loops, but it does have recursion. Whilst the vast majority of programming languages are Turing-complete. As in practice, most computable functions are primitive recursive; one could argue that the benefits gained by using a Turing-complete language are not worth the costs associated with infinite loops. However, in general, the size or complexity of a function written in a Turing-complete language is smaller than the equivalent function written in a primitive recursive programming language.[8]

In contrast to Bitcoin, Ethereum[9], a blockchain with smart contract functionality, provides a virtual machine that can execute Turing-complete scripts. In practice, infinite loops are avoided by requiring that each transaction sets a limit to how many computational steps of code execution it can use, and users must pay for this limit in advance using a scarce resource. Because the limit is fixed in advance of the code being executed, unbounded loops are not possible, and thus Ethereum is not Turing-complete either.

III. AUTOMATA THEORY

Automata theory is the study of abstract computing devices that follow a predetermined sequence of operations automatically.[10] An automaiton consumes a string of input symbols, called a word. For each input symbol, it transitions to a new state and continues until all input symbols have been consumed. It then accepts or rejects the given string of symbols. The set of all the words accepted by an automaton is called the language recognized by the automaton. The languages recognized by automata may be classified as nested subsets. Thusautomata may be ranked in terms of how powerful they are. The larger the subset of languages recognized, the more power the automata has. Fig. ***2*** shows the hierarchy of automata, along with Bitcoin as a finite automaton.

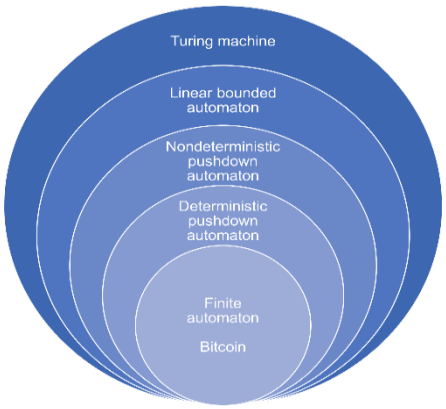

Fig. 2 Hierarchy of automata

We now consider each type of automata in turn, from the least powerful to the most powerful, and how they relate to Bitcoin.

A. Finite Automaton

A finite automaton is the least powerful type of automata that we consider, and can be represented formally by a 5-tuple, $M=\langle Q, \Sigma, \delta, s, F\rangle$, where:

- Q is a finite set of states,
- Σ is a finite set of input symbols (the alphabet),
- δ is the transition function, $\delta: Q \times \Sigma \rightarrow Q$, mapping state-input pairs to successor states,
- s is the start state ($s \in Q$), and
- F is the set of accepting states ($F \subseteq Q$).

A finite automaton accepts regular languages. A finite automaton requires a transition function such that the next state is a function of the current state and the current input symbol. Is Bitcoin a finite automaton? Bitcoin may be considered a finite automaton if we let transactions be the input symbols, the blockchain represents states, and the Bitcoin Core code the transition function.

B. Deterministic Pushdown Automaton

A deterministic pushdown automaton is essentially a finite automaton plus a stack and accepts the deterministic context-free languages. So a pushdown automaton requires a stack, plus a transition function such that the next state is a function of the current state, the current input symbol, and the symbol at the top of the stack. Is Bitcoin a deterministic pushdown automaton? The stacks within Script are contained within VerifyScript, which does not enable the transfer of any data

from one instance to another, so the stacks have no memory of past transactions. Alternatively, if the blockchain represents the stack, the transition function, VerifyScript, must have access to some external state within the Bitcoin Core code, but VerifyScript is not a function of any external state. Therefore Bitcoin is not a deterministic pushdown automaton.

C. Nondeterministic Pushdown Automaton

A non-deterministic pushdown automaton is similar to a deterministic pushdown automaton, except that the transition function is a multivalued function. That is, the transition function maps the current state, the current input symbol, and the symbol at the top of the stack to a *set* of states (zero, one, or more). We may think of a non-deterministic pushdown automaton as branching at every step, and if at least one of the permutations accepts the input string, the string is said to be recognized by the automaton. A non-deterministic pushdown automaton can recognize all context-free languages. Because Bitcoin is not a deterministic pushdown automaton, it cannot be a non-deterministic pushdown automaton (which is more powerful).

D. Linear Bounded Automaton

A linear bounded automaton is a non-deterministic Turing machine that satisfies the following three conditions[11]:

- Its input alphabet includes two special symbols, serving as left and right end markers.
- Its transitions may not print other symbols over the end markers.
- Its transitions may neither move to the left of the left end marker nor to the right of the right end marker.

Because Bitcoin is not a pushdown automaton, it is not a linear bounded automaton (which is more powerful).

E. Turing Machine

We defined a Turing machine above as a mathematical model of a hypothetical computing machine that manipulates symbols on a strip of tape according to a predefined set of rules. Perhaps more useful in the current context is to note that a Turing machine is equivalent to a pushdown automaton with two stacks. That is, every language that is accepted by a Turing machine can also be accepted by a deterministic pushdown automaton with two stacks. Whilst a non-deterministic two-stack pushdown automaton is equivalent to a deterministic two-stack pushdown automaton. With a Turing machine, the next state is a function of the current input symbol, the current state, the symbol at the top of stack 1, and the symbol at the top of stack 2. Turing machines accept all recursively enumerable languages. Because Bitcoin is not a linear bounded automaton (or a pushdown automaton), it cannot be a Turing machine (which is more powerful).

F. Discussion

To conclude this section, Bitcoin may be considered a finite automaton, but not a deterministic pushdown automaton, a non-deterministic pushdown automaton, a linear bounded automaton, or a Turing machine. The only memory Bitcoin has the blockchain, which represents its state.

IV. CONCLUSION

In computability theory, there exists a hierarchy of functions, from the most to least powerful: partial recursive, total recursive, primitive recursive, elementary recursive, and lower elementary recursive. Whilst in automata theory, there exists a hierarchy of automata, from the most powerful to the least powerful: Turing machine, linear bounded automaton, non-deterministic pushdown automaton, deterministic pushdown automaton, and finite automaton. In both instances, Bitcoin lies within or below the least powerful category. Bitcoin is essentially a finite automaton that employs a scripting language for data manipulation that is even less powerful than a lower elementary recursive programming language. Bitcoin was designed to be as powerful as it needed to be, and no more. No part of Bitcoin is Turing-complete. The entire Bitcoin system only becomes Turing-complete if we use a Turing-complete programming language to broadcast transactions.

ACKNOWLEDGMENTS

The author wishes to thank an anonymous internal reviewer from Ledger for useful feedback.

REFERENCES

[1] S. Nakamoto, Bitcoin: A peer-to-peer electronic cash system, https://bitcoin.org/bitcoin.pdf (bitcoin.org) (2008).
[2] Wikipedia, ELEMENTARY, https://en.wikipedia.org/wiki/ELEMENTARY, (2021).
[3] A. R.Meyer, and D. M.Ritchie., The complexity of loop programs, in S. Rosenthal, Ed. ACM '67: Proceedings of the 1967 22nd national conference. New York: Association for Computing Machinery, (1967) 465–469.
[4] D. R.Hofstadter, Gödel, Escher, Bach: An eternal golden braid. New York: Basic Books, (1979).
[5] W. Ackermann, Zum Hilbertschen Aufbau derreellen Zahlen, Mathematische Annalen, 99(1) (1928)118–133.
[6] A. M.Turing, On computable numbers, with an application to the Entscheidungs problem, Proceedings of the London Mathematical Society, s2-42 (1) (1937) 230–265.
[7] A. M.Turing, On computable numbers, with an application to the Entscheidungs problem. A correction, Proceedings of the London Mathematical Society, s2-43 (1) (1938) 544–546.
[8] M. Blum, On the size of machines, Information and Control, 11(3) (1967) 257–265.
[9] G.Wood, Ethereum: A secure decentralized generalized transaction ledger, https://ethereum.github.io/yellowpaper/paper.pdf., (2020).
[10] J. E.Hopcroft, R. Motwani, &J. D.Ullman., Introduction to automata theory, languages, and computation,3rd ed. Harlow: Pearson, (2014).
[11] Wikipedia, Linear bounded automaton, https://en.wikipedia.org/wiki/Linear_bounded_automaton, (2020).