# A Proof of Turing completeness in Bitcoin Script

Author: Craig Wright

Abstract

*The concept of a Turing machine has been well defined {[17], [20]}. It would be sufficient to show that Bitcoin uses a dual stack architecture that acts as a dual counter machine. Such systems have already been demonstrated as being Turing complete [17].*

*We demonstrate that Bitcoin script is a minimal family of which* $\lambda$ and *R are members. Further using the compositional product rule and the iteration rule we demonstrate that Bitcoin scripting is Turing complete with the limitations imposed on any real-world computer. This limitation is that there cannot be an infinite tape. Iterations can be simulated using an "unrolled" loop function with allocation to the "Alt" stack.*

*As the product rule states that if A, B are machines, then A.B is also a machine [17]. The iteration rule shows that if A is a machine then (A) is also a machine. Further the minimum power of A under which the observed square of the final configuration is blank. The consequence of these rules is that for every partial recursive function of in variables we can show that it can be evaluated by machine of the proposed family {[7], [21]}.*

**Keywords:** Bitcoin, Quantum Computing, Encryption,

# 1 Contents

# 2 Introduction

The concept of a Turing machine has been well defined {[17], [20]}. It would be sufficient to show that Bitcoin uses a dual stack architecture that acts as a dual counter machine. Such systems have already been demonstrated as being Turing complete [17].

For our purposes, we will revert to the early basic Turing machines $\lambda$and R. These shall be used in the explanations presented in this paper {[17], [21]}. The machines are defined using a one-way infinite tape that implements an infinite alphabet $\{c_0, c_1, \cdots, c_n\}$ in which $n \geq 1$. In this alphabet, $c_0$ is defined as the blank symbol of the tape. We will demonstrate that Bitcoin script is a minimal family of which $\lambda$ and R are members. Further using the compositional product rule and the iteration rule we demonstrate that Bitcoin scripting is Turing complete with the limitations imposed on any real-world computer. This limitation is that there cannot be an infinite tape. Iterations can be simulated using an "unrolled" loop function with allocation to the "Alt" stack.

As the product rule states that if A, B are machines, then A.B is also a machine [17]. The iteration rule shows that if A is a machine then (A) is also a machine. Further the minimum power of A under which the observed square of the final configuration is blank. The consequence of these rules is that for every partial recursive function of in variables we can show that it can be evaluated by machine of the proposed family {[7], [21]}.

## 2.1 A programming language in Bitcoin script that is equivalent to the Turing machine family

We shall start by giving our system an alphabet, $C = \{c_0, c_1, \cdots, c_n\}$ with $n \geq 1$.

We will introduce a Turing machine that uses an infinite tape to the left divided into squares of which only a finite number of symbols of our alphabet C are included with the remaining squares all being blank. For this purpose, each of these remaining squares will be marked with the symbol $c_0$.

In our definition, we use ahead which can read or write one square at a time and is hence movable on the tape. This process is simulated using the primary Bitcoin script stack coupled with the use of the Alt stack is a dual counter machine. It would be sufficient to demonstrate that the use of a dual stack system, which is in effect, a multi-tape 2-symbol Post-Turing machine with its behavior restricted so its tapes act like simple "counters"[1]. The Church-Turing thesis hypothesizes that anything that can be "computed" can be computed by some Turing machine [20]. Hence, any system that can act as a Turing machine is a complete computational system. In this manner we will demonstrate the Bitcoin script is Turing complete other than the limitation imposed on real-world conditions of non-infinite bounds.

The system deployed inside Bitcoin replicates well-known machines that have been shown to be equivalent to a program with a fixed number of atomic instructions {[1], [8], [13], [15], [16], [21]}.

Cockshott & Michaelson [22] explored the idea of the Turing machine in depth with regards to the need for Turing machines to have infinite tapes and whether their "*expressive power is limited by an inability to interact with the wider environment during computations"*. They noted that Turing did not state a machine to be infinite, but rather used the term, unbounded.

[1] Marvin Minsky, Computation: Finite and Infinite Machines, Prentice–Hall, Inc., N.J., 1967. See Chapter 8, Section 8.2 "Unsolvability of the Halting Problem."

It is also known that at least in the case of n=1 that a general set can be formed through the three following instruction [12]:

- R. This command shifts the head of one square to the right.
- $\lambda$. This instruction directs the machine to replace the scanned symbol $c_i$ with the symbol $c_{i+1} \bmod(n+1)$ and then to shift the head one square to the left.
- $\uparrow$. This instruction directs the machine to jump to the instruction marked with an arrow if the scanned symbol is not blank, i.e. $c_0$.Otherwise, the machine will continue with the next instruction immediately to the right of the current position if one exists [16].

For instance:

$$S^+ \equiv \overline{\left|\underset{\rightarrow}{\overline{R}}\right|}\left[\lambda R\right]^n \lambda \overline{\left|\rightarrow \lambda R \overline{\left|\_ \lambda R\left[\lambda R\right]^n \lambda \_\right|} \overline{\lambda R}^{\rightarrow}\right|}\left[\lambda R\right]^n \lambda \left|\right.$$

In this, $\left[H\right]^l$ replaces $HH...H$, $l$ times with $H^0 = \Phi$ being used to represent an empty word.

And program, $\left[G\right]^l$ replaces $GG...G$, $l$ times with $G^0 = \Phi$ being used to represent an empty word representing functional boxes [3].

In this paper, we extend this to the general case of (n>1) as well. It is already proven [11] that programs can be normalised in a manner that allows for the following:

1. That a program can be constructed from our machine with a first instruction performed which is on the far left will always be $\lambda$ and the final instruction to formed by a machine on the far right will always be R.
2. That any conditional jump will always begin at a point immediately after and to the right of an instruction R. This can also be demonstrated [18] to always reach an instruction $\lambda$.
3. Will also demonstrate the conversed conditional jump $\mp$ can be used to replace the instruction $\uparrow$. This condition occurs where the scan symbol is a blank, $c_0$.

For definitional purposes, each $\lambda$ that is reached by our instructions will be appended with a subscript $i = 1, 2, \cdots$. In this way, there will not be two $\lambda$ instructions with subscripts of the same value. Next, for each R where there is an arrow on its right going to $\lambda_i$ we will append the subscript $i$ and suppress the arrow which can be shown to convey no additional information.

Using the script language in Bitcoin, we can copy from the primary stack to the Alt stack in the manner of a dual counter stack machine [17].

By repeating this procedure for every value of $i$, we obtain a defined word in the infinite alphabet [15]:

$$\Omega \equiv \left\{\lambda, R\right\} \cup \quad \Omega_\lambda \quad \cup \Omega_R \qquad\qquad where \qquad \begin{matrix} \Omega_\lambda \equiv \left\{\lambda_1, \lambda_2, \cdots\right\} \\ \Omega_R \equiv \left\{R_1, R_2, \cdots\right\} \end{matrix}$$

Next, let us further suppose that among the atomic instructions we have included the jump, $\mp$, in the same manner such that we can replace the arrows with subscripts. Hence, we will write $\overline{R}_i$ in the place of where we would otherwise write $R_{\mp}$. Hence, we extend $\Omega$ to the following format:

$$\Omega' \equiv \Omega \cup \Omega_{\overline{R}} = \Omega \cup \left\{\overline{R}_1, \overline{R}_2, \cdots\right\}$$

From this we can create a formal syntactic definition of a program with both $\Omega$ and $\Omega'$. This will follow in the next section.

## 2.2 Syntactic characterization of a program

*We can be we can define a program to be a word P in* $\Omega(\Omega')$ *such that for* $i = 1, 2, \cdots$:

4. For each and every occurrence of $R_i$ there exists in *P* exactly one occurrence of $\lambda_i$.
5. For each occurrence of $\lambda_i$ there is in *P* at least one occurrence of $R_i$.

### 2.2.1 Proof and denotations

We will start by interpreting $\lambda, R$ as functions that are partially defined on configurations having configurations as values [12]. In this sense, a configuration is defined as an ordered pair of words over $C \cup \{c_0\}$ where the first word in the set cannot be of the type $c_0, h > 0(3)$. This configuration further represents the finite contents of the tape with the inclusion of the scanned symbol being through convention the first symbol to the left of the second word [2]. In this condition, the i[th] jump is conditioned by a predicate *p* for which we can say:

- $p \Rightarrow (\text{Scanned}) = c_0$ (Where scanned refers to the scanned symbol read by the machine head),
- p is otherwise undefined.

We extend this definition to let the symbols x, y, be defined in such a way that they can express variable words in the set $\Gamma$ of all words over the alphabet $C \cup \{c_0\}$.We may further indicate using $c$, $c'$, $c''$ the variables defined in the set C. Similarly, this also defines the set of variables, $C \cup \{c_0\}$. We will further define $xy$ to be the word resulting from the concatenation of the individual words $x$ and $y$.

Following, we may use the following symbols $R, \lambda, p$, that can be partially defined recursively for each possible configuration:

6. $R(x, \Phi)\downarrow$ $\quad \lambda(x, \Phi)\downarrow$ $\quad p(x, \Phi)\downarrow$
7. $R(\Phi, c_0 y) = (\Phi, y)$ $\quad \lambda(\Phi, c_i y) = (\Phi, c_0 c_{i+1} y)$ $\quad p(x, c_0 y).True$
8. $R(\Phi, cy) = (c, y)$ $\quad \lambda(xc', c_i y) = (x, c' c_{i+1} y)$ $\quad p(x, cy).False$
9. $R(xc', c''y) = (xc'c'', y)$

Remembering that, $i = 1, 2, \cdots, n$ and the sum is always $\text{mod}(n+1)$.

If we define $Z = (Q_z, m_z, q_0, q_m)$ to represent a Turing machine over the alphabet $C \cup \{c_0\}$, where we have defined $Q_z = \{q_0, q_1, \cdots, q_m\}$ As representing the set of internal states available to our machine and where $m_z$ is the table of $Z$. Hence:

$$m_z(q_i, c_i) = (c_{ij}, t_{ij}, q_{ij})$$

$$q_i \in Q_z - \{q_m\}; c_i, c_{ij} \in C \cup \{c_0\}; t_{ij} \in \{R, I, L\}; q_{ij} \in Q_z$$

In the above equation,

- $I$ Represents the identity,
- $L$ Represents the left ships where: $L = [\lambda \quad R] \circ \lambda$,
- $q_0$ Represents the initial state of $Z$.
- $q_m$ Represents the final state of $Z$.

We will further identify $c_j$ with $j\,(j = 0,1,\cdots,n)$.

These definitions allow us to state that for each Turing machine $Z$ over $C \cup \{c_0\}$ there will exist a corresponding program $P_z$ over $\Omega$[2] in the manner that there will exist a starting configuration where $(a,b), Z$ and $P_z$ either does not stop and continue in the same manner failing to holt, or both stopped in the same configuration, $P_z(a,b) = (a',b')$.

2.2.2 Proof

Case $\Omega$ (jump on $p(x, c'y) = \perp$ or $c' \in C$)

In order to simplify this, we will write it as follows:

$i, j$ to be inserted in the place of $2(n+1)i + j$

And

$i, \underline{j}$ to be inserted in the place of $2(n+1)i + (n+1)j$

It is then easy to determine whether for $P_z$ if we can select the word,

$$\lambda R_{0,n} \lambda R_{0,n} H_0 G_{00} \cdots G_0 H_1 G_{10} \cdots G_{1n} \cdots H_{m-1} G_{m-1\ 0} \cdots G_{m-1\ n} H_m$$

In which we have the following values:

[2] Here $\Omega$ is equivalent to $\Omega' - \Omega_R$.

$$H_i = \lambda_{i,n}R_{i,n-i}\lambda R_{i,n-i}\lambda_{i,n-1}\cdots\lambda R_{i,1}\lambda R\lambda R \quad (i=0,1,\cdots,m-1)$$

$$H_m = \lambda_{m,n}R\lambda_{m,n}R\left[\lambda R\right]^{n-2}$$

$$G_{ij} = \lambda_{i,j}R\left[\lambda R\right]^{c_{ij}-2}U_{t_{ij}}\lambda R_{q_{ij},\underline{n}}\lambda R_{q_{ij},n} \quad (i=0,1,\cdots,m-1;\ j=0,1,\cdots,n)$$

Being defined using the difference $\mathrm{mod}(n+1)$ and in the Case:

$$\Omega'-\Omega_R \qquad \text{(We jump when the condition } p\left(x,c'y\right)=\mathrm{T};\ c'=c_0\text{)}$$

We can represent the program $P_z$ that we write through the use of $i, j$ substituted in the place of $\left(n+1\right)i+j$

We can use this to simply show that for $P_z$we can select a word:

$$\lambda R_{0,n}\lambda R_{0,n-1}\cdots\lambda R_{0,1}\lambda RH_{00}^{'}H_{01}^{'}\cdots H_{0n}^{'}H_{10}^{'}\cdots H_{1n}^{'}\cdots H_{m-1\ \ 0}^{'}\cdots H_{m-1\ \ n}^{'}H_m^{'}$$

Were it can also be seen that:

$$H_{ij}^{'} = \lambda_{i,j}R\left[\lambda R\right]^{c_{ij}-1}U_{t_{ij}}\lambda R_{q_{ij},n}\lambda R_{q_{ij},n-1}\cdots\lambda R_{q_{ij},0}$$

$$\begin{bmatrix} i=0,1,\cdots,m-1 \\ j=0,1,\cdots,n \end{bmatrix}$$

Further, we have the state:

$$H_m^{'} = \lambda_{m,n}R\lambda_{m,n-1}R\cdots\lambda_{m,1}R\lambda_{m,0}R\left[\lambda R\right]^n$$

Where the prior uses the same conventions as have been previously deployed.

## 2.3 The Selection of a Sub-Language $\Im''$ contained within $\Omega'$ and the accompanying properties

We start by formulating the set of all possible programs in $\Omega'$ as $\Im'$. This construct represents the set of all words in $\Omega'$ which satisfy the definitions listed above. Through the methods detailed by Ianov [11] we can extend $\Im'$as a more generalized system that can describe programs for any existing computer. This can be mapped directly onto the stack [13].

Through the use of our previously defined functions, we can combine several functional elements together as objects that are far more powerful. In using this method, we shall generally substitute selected subscripts to avoid collisions and escapes from $\Im'$. In a general Turing system, the base alphabet is infinite. In this way, we can say that our program formation rules are context depending.

Any potentially infinite language is to set the least problematic. As such it is necessary to extract a sub language $\Im''$from the complete set $\Im'$ where we can say that $\Im''\subset\Im'$and that is context free more maintaining the capability to compute all partially recursive functions.

### 2.3.1 A definition of $\Im''$

For any word, *P*, that is defined within $\Omega'$ may be considered a functional program within $\Im''$ when it fulfils the conditions listed above and the following properties [17]. The conditions above defined $P \in \Im'$.

#### *2.3.1.1 Condition 1*

For each occurrence of $\lambda_i$ there will exist at most a single occurrence of $R_i$ or its alternative $\overline{R_i}$.

#### *2.3.1.2 Condition 2*

The number of jumps within our language is even and may be defined as $2.N$ jumps that is part of a set that is partition of all within pairs $(i_j, k_j), (j = 1, 2, \cdots, N)$. These pairs are defined in a manner such that the following pairs of words can be considered to occur within P:

$$\left[\left(\overline{R}_{i_j}\lambda_{k_j}\right), \left(R_{i_j}\lambda_{i_j}\right)\right].$$

#### *2.3.1.3 Condition 3*

For any two values of j that are distinct such that $j_1 \neq j_2$ which we will define as q and r, we can consider P in the following geometrical intervals:

$$I_q = \overline{R}_{i_q} \cdots \lambda_{iq}, I_r = \overline{R}_{i_r} \cdots \lambda_{ir}$$

All such intervals will lie within the following relation:

$$\left(I_q \subset I_r\right) \vee \left(I_q \supset I_r\right) \vee \left(I_q \equiv \varnothing\right)$$

A restriction on bitcoin scripting languages[3] is that they are only able to enter a cycle from its first instruction. This limitation also exists within the Forth and FORTRAN programming languages[4] such as with the "DO" statement. A further restriction is that it is only possible to leave a loop was in bitcoin code from its final instruction.

Following this definition, we can see that the $\Im''$ language and constructs created from it such as the bitcoin scripting language are context free. For instance, if we are to use the following values $R(\lambda$ in place of $\left(\overline{R}_{i_j}\lambda_{k_j}\right)$ and $R)\lambda$ in place of $\left(R_{i_j}\lambda_{i_j}\right)$ for each $\left(j = 1, 2, \cdots, N\right)$ we find that the information conveyed remains the same.

As a consequence, we argue that the finite alphabet $\Omega'' = \left|\lambda, R, (,)\right|$ is sufficient to define the language $\Im''$. This can be defined using the following:

- $\lambda, R$ a words contained within $\Im''$ (using the axiom of atoms);
- Where $\alpha, \beta$ a words contained within $\Im''$, then it follows using the composition rule that $\alpha\beta$ are also words within $\Im''$

---

[3] See https://en.bitcoin.it/wiki/Script and http://davidederosa.com/basic-blockchain-programming/bitcoin-script-language-part-one/

[4] See http://fortranwiki.org/fortran/show/HomePage and https://gcc.gnu.org/fortran/

- in the instance where $\alpha$ is defined as a word within $\mathfrak{I}$", then $(\alpha)$ is also a word within $\mathfrak{I}$" following the iteration rule.

### 2.3.2 Semantics

Every $P \in \mathfrak{I}$" can be interpreted as a partially defined function on the set $\Theta$ consisting of the configurations onto itself. $\lambda$, $R$ are defined within this known interpretation. Where we now have $\alpha$, $\beta$ as functions and $(a,b)$ can be any configuration, we can say that the application of $\alpha\beta$ to $(a,b)$ refers to the process of applying the function $\gamma$, which is defined in the following manner:

$$\gamma(a,b) = \beta(\alpha(a,b))$$

For instance, we can say that $\alpha\beta$ is the composition of $\alpha$ and $\beta$.

We can also state that where $\alpha$ is a function and where we designate $\alpha^n$ to refer to the composition of $\alpha$ upon itself n-times, we can easily prove that when $(\alpha)$ is applied to the configuration $(a,b)$ it can also be interpreted as $\alpha^n(a,b)$ and we can further [2] show that:

$$n = \mu\nu\left[p\left(\alpha^{\nu}(a,b)\right)\right]$$

And of course that $\alpha^0 = I$ defines the identity function [17].

For simplicity it is easier to define and identify the language $\mathfrak{I}'$ to be the family of altering machines which also corresponds to the family of all corresponding functions from $\Theta$ onto $\Theta$. In this we will consider the definition set $\Theta_z$ of the initial configurations with the property that all other configurations are defined for each machine Z. Another way of stating this is that programs defined within $\mathfrak{I}'$ exceed the right and of the tape during computation by avoiding the head.

#### *2.3.2.1 Condition 4 – Right Monotony*

Where $Z \in \mathfrak{I}', (\gamma_1,\gamma_2) \in \Theta_z, Z(\gamma_1,\gamma_2) = (\delta_1,\delta_2)$ then we can also say that for every word

$$x : Z(\gamma_1,\gamma_2 x) = (\delta_1,\delta_2 x)$$

#### *2.3.2.2 Condition 5 - Invertibility*

We can extend right monotony if we add the following:

$$\text{Z is a word over } [\lambda, R]$$

then we can also say that a further word $Z^{-1}$ exists over $[\lambda,R]$ such that it is necessary for:

$$Z^{-1}(\delta_1,\delta_2) = (\gamma_1,\gamma_2)$$

And of course, using the identity function we can also say that $ZZ^{-1} = 1$

Other forms of machines having right inverse structures [3] of the type mentioned can be created {[10], [17]} but from this we know that the above statements hold for $\mathfrak{I}$" and have been applied within bitcoin script. We will further consider the machine H which is defined to do nothing [12] if the head scans

initially a symbol that differs from $c_0$ and otherwise proceeds to the left until it first reads the symbol $c_0$ .

This machine can be represented as follows:

$$H \equiv r'\lambda R_1 \lambda_2 R r''\lambda r' R_2 \lambda_1 R r'$$

Where:

$$r' = [\lambda R]^n \qquad r'' = [\lambda R]^{n-1}$$

In this representation, it is evident that $H^{-1}$ exists and we can represent this as follows:

$$H^{-1} \equiv r'\lambda R_1 \lambda_2 R r' R_2 \lambda_1 R r'$$

At present, neither $\Im'$ or $\Im''$ have been adequately characterised. Functionally, if we take this as the perspective, the following section allows us to did use that is not representable within $\Im''$.

#### 2.3.2.3 Condition 6

If $Z_2$ has brackets, $Z_1$ does not even exist in $\Im'$, such that $Z_1 Z_2 = I$.

Through the use of *reductio ad absurdum*, we will start with the postulation:

$$Z_1 Z_2' (Z_2'') Z_2''' = I$$

In the above formulation, $Z_2'$ and $Z_2''$ can be empty and define words over $\{\lambda, R\}$.

From the above postulation using the associative property of the composition and the Invertibility of $Z_2'''$ we can deduce the absurd result that:

$$Z_2''' Z_1 Z_2' (Z_2'') = I$$

The closing configuration for I coincides with the initial one in this implementation. This result is arbitrary not of the type $(x, c_0, y)$ as occurs for every program that ends with brackets. In the instance where $Z_1 = H$ we can determine that the conclusion is false and hence the postulation itself is also false. Here, for any $Z_2 = H^{-1}$ that exists within $\Im''$, the value must not have brackets. This is a contradiction and cannot be true as the number of instructions executed within such a program is fixed whilst in the particular instance of $H^{-1}$ this number depends upon the initial configuration. Such we can say that $H^{-1}$ is not representable within $\Im''$.

To generalise this, we can state that all programs that are invertible in $\Im'$ which are not words contained within $\{\lambda, R\}$ are excluded from $\Im''$.

## 2.4 Computing a partial recursive function within $\Im''$

Putting all of these results together, we can state that for all possible partial recursive functions $f$ of $m \geq 0$ variables, a program $P_j \in \Im''$ exists which can compute within our framework. As we can

represent $\mathfrak{I}''$ as a series of expanded recursive functions within bitcoin script, we can align this with the results stated by Robinson [14] concerning the recursive functions of one variable. This theorem is syntactically aligned to the results we have presented here.

These results can be achieved through the process of attempting to simplify and normalise the flow diagrams associated with Turing machines or standardised computers. In this, the proving formulas are for all intents and purposes independent of the number n of symbols in C. As such we can see that only the sub-programs $r, r'$ remain dependent upon n. In the particular case of n=2, this methodology aligns to that of Smulluan [18, p10] and can be achieved by adopting a uniform method of representing the integers through digits of the base $\{1,2,\cdots,n\}(n \geq 1)$ according to which every integer $\nu > 0$ has a uniquely determined representation of the form or type:

$$\nu_h \nu_{h-1} \cdots \nu_0 \qquad 1 \leq \nu_i \leq n$$

Where $$\nu = \sum_0^h \nu_i n^i$$

in such a representation, we can display the hexadecimal (base 16) set as:

$$\{1,2,\cdots,9,a,\cdots,f,t\} \text{ and } 10 \leftrightarrow t, 100 \leftrightarrow ft, 1010 \leftrightarrow ftt$$

As a result, in a system the number of zero will be represented using the empty word $\Phi$.

In order to prove our propositions, we need to start by defining the meaning of computing and computation. We will begin by documenting a formal definition of the concept of a partial recursive function.

Definition 1:

A program $P_j$ computes the function $f$ of $n$ variables in the event that by denoting through the use of the words $a_1, a_2, \cdots, a_m$ over C that correspond respectively to the values of $x_1, x_2, \cdots, x_m$ and through the use of $f(a_1, a_2, \cdots, a_m)$ the word corresponding to the value of the function, we obtain:

$$P_j(\Phi, \square a_1 \square a_2 \square \cdots \square a_m \square) = P_j(\Phi, f(a_1, a_2, \cdots, a_m) \square)$$

in this equation the character $\square$ is used to define the blank symbol $c_0$.

Definition 2:

We can obtain the set of partial recursive functions through an application against the basic functions a finite number of times against the schemas below [10].

*Basic Functions:*

$O^{(m)}$ Zero function of m variables $O^m(x_1, x_2, \cdots, x_m) = 0 \qquad m \geq 0$

$S^+$ successor $S^+(x) = x + 1$

$S^-$ predecessor $S^-(x) = x - 1$ undefined for $x = 0$

$U_i^{(m)}$ selection function $U_i^{(m)}\left(x_1, x_2, \cdots, x_m\right) = x_i \qquad i = \left(1, 2, \cdots, m\right)$

*Schemas:*

1. *Composition.*

Where $f^{(i)}, g_i^{(m)}$ are partial recursive $\left(m \geq 0, l > 0, i = 1, 2, \cdots, l\right)$ and $h^{(m)}$ with defined as $h\left(x_1, x_2, \cdots, x_m\right) = f\left(g_1\left(x_1, x_2, \cdots, x_m\right), \cdots, g_l\left(x_1, x_2, \cdots, x_m\right)\right)$ which is partial recursive.

2. *Recursion.*

Where $f^{(m+1)}, g^{(m+2)}, q^{(m+1)}, p$ (consisting of a single variable) are partial recursive then $h^{(m+1)}$ is also partial recursive. We define $h^{(m+1)}$ as follows:

$$h\left(y, x_1, x_2, \cdots, x_m\right) = \begin{cases} f\left(y, x_1, x_2, \cdots, x_m\right) & \text{if} \quad q\left(y, x_1, x_2, \cdots, x_m\right) = 0 \\ g\left(h\left(p\left(y\right), x_1, \cdots, x_m\right)\right), p\left(y\right), x_1, \cdots, x_m & \text{otherwise} \end{cases}$$

We note that the above-mentioned recursion schema does not follow the standard process of more widely known schemas of primitive recursion and minimization [14]. Hence we can say that it differs from the more standard state of programming used today [12]. It is nevertheless simple to demonstrate that the minimisation schema can be obtained by writing [11]:

$$f^{(m+1)} = U_1^{(m+1)}, g^{(m+2)} = U_1^{(m+2)}, S^+$$

From which:

$$h'\left(x_1, x_2, \cdots, x_m\right) \equiv h\left(0, x_1, x_2, \cdots, x_m\right) = \mu y\left\{\left(x_1, x_2, \cdots, x_m\right) = 0\right\}$$

We can further obtain the primitive recursion schema by writing:

$$q = U_1^{(m+1)}, p = S^-.$$

## 2.5 Proof

One use of the sort of function is in the creation of encoding sequences. Various possibilities could be sent to recipient parties who would then be able to extract the valid transaction with the addition of minimal work. Implemented in conjunction with a set theoretic framework, the inclusion of sub and supersets of keys would enable the secure creation of threshold-based key hierarchies. Each of these hierarchies could act in partial signing allowing for the secure amalgamation of signature parts. Additionally, further privacy and security could be obtained through the integrated salting and randomisation of key values and signatures. The aim of this research would lead to fully integrated homomorphism encryption schemes built into simple distributed scripts.

We are now ready to prove the predicate system is complete. We will start by defining some subprograms which make the application of the proof simpler:

$$r = \lambda R, r' = \left[\lambda R\right]"$$

And we have already demonstrated above that:

$$rr' = r'r = I,\ r'\lambda = L$$

Such that:

$$r(\Phi, c_i) = (\Phi, c_{i+1}), r'(\Phi, c_i) = (\Phi, c_{i-1})$$

We will now introduce the following routines, $\mathrm{T}^+$ and $\mathrm{T}^-$.

$$\mathrm{T}^+ \equiv (r) R^2 ((r'LrR) R) L$$

$$\mathrm{T}^- \equiv (r) L^2 ((r'RrL) L) R$$

From this we can show that for any word x over C where $c \in C \cup \{\square\}$ that:

$$\mathrm{T}^+ (x\square, c) = (\Phi, \square x\square)$$

$$\mathrm{T}^- (\Phi, c\square x\square) = (x, \square\square)$$

Utilising the functions, we have just defined, $\mathrm{T}^+$ and $\mathrm{T}^-$ we will now define a doubling routine that provides recursion in $\mathfrak{I}''$ and hence bitcoin script:

$$W \equiv LT^- L_\square LT^- L\left(L_\square^2 rR_\square^2 RrL^2 r'\left(L_\square^2 RrR_\square^2 RrL^2 r'\right) L\right) RT^+ R_\square RT^+$$

Where:

$$L_\square \equiv L(L), \qquad R_\square \equiv R(R)$$

With the effect of:

$$W(\Phi, \square x\square) = (\Phi, \square x\square x\square)$$

In more general terms, we can build, $W_m$ (where $W_1 \equiv W$)

$$W_m \equiv LT^{-m} L_\square^m LT^{-m} L\left(L_\square^{m+1} rR_\square^{m+1} RrL^2 r'\left(\ L_\square^{m+1} RrR_\square^{m+1} RrL^2 r'\right) L\right) RT^{+m} R_\square^m RT^{+m}$$

With the effect of:

$$W_m (\Phi, \square x_1 \square x_2 \square \cdots \square x_m \square) = (0, \Phi, \square x_1 \square x_2 \square \cdots \square x_m \square)$$

Finally, we integrate two cancellation routines:

$$K_L \equiv R((r) R)$$

$$K_R \equiv R_\square R\left(T^+ R_\square R\right) T^+$$

These have the following effect:

$$K_L (\Phi, \square x\square y\square) = (\Phi, \square y\square)$$

$$K_R (\Phi, \square x\square y\square) = (\Phi, \square x\square)$$

Now we can create the programs for the recursive functions. It is relatively simple to verify that the following programs satisfy the requirements:

$$O^{(m)} \equiv K_L^m L$$

$$U_i^{(m)} \equiv K_L^{i-1} K_R^{m-i}$$

$$S^+ \equiv R_\square L\left(r\left(L_\square L^2 r R^2\right) rL\right) rL\left((r) R(r) R(r)\right)$$

$$S^- \equiv R_\square L\left(r'\left(L_\square\right) r' L\right) Rr$$

1. Composition.

We will let $F, G_1, \cdots, G_l$ represent the programs for $f^{(l)}, g_1^{(m)}, \cdots, g_l^{(m)}$.

As a result, the program for $h^{(m)}$ is defined as:

$$H \equiv W_m^m G_l W_{m+1}^m G_{l-1} \cdots W_{m+1-1}^m G_1 F K_R^m$$

2. Recursion.

We set $F, G, Q, P$ to represent the functions, $f^{(m+1)}, g^{(m+z)}, q^{(m+1)}, p$. As such, we find the program for $h^{(m+1)}$ to be:

$$H \equiv L W_{m+2}^{m+1} QR\left(\left((r) R\right) W_{m+2}^{m+1} P W_{m+2} S^+ W_{m+2}^{m+1} QR\right) W_{m+2}^{m+1} F R_\square R \bullet$$
$$\bullet\left(L L_\square K G R_\square R\right) L L_\square K_R^{m+2}$$

This proof follows the most commonly entered methodology used by Turing [11], Davis [5] and Hermes [8]. In this proof, we have taken advantage of the right monotony property [6].

As with parallel computation, we have demonstrated that our system can act using predicate constructs to solve complex calculations. Each script takes the same input and can both run within script or output depending on the returned value. When the predicate value returns true prior to any other conditions being returned, the initial response becomes the probabilistic answer written into the Blockchain. All false returns are rejected and are not transmitted across the network. In this way, we can test multiple statements simultaneously acting only upon the correct results. Sending each of these on as an independent predicate function allows as to build a series of constructs that are guaranteed to be true.

# 3 Conclusion

It is not clear to see that Bitcoin is Turing complete, but once the use of the stack commands is introduced to the repertoire, we see that Script is a Stack machine that consists of a Dual stack architecture. In this, the Primary stack has a series of commands that are designed to manipulate the stack with a high degree of malleability and the Alt stack allows for a pointer and counter function.

Script commands such as OP_PICK and OP_ROLL provide a great deal of granularity and control of the stack at any depth. When these commands are combined with data Pop'd into and off of the Alt stack as a control, the ability to extend Bitcoin script (using the existing operations) becomes clear.

In this paper, we have provided a methodology to both extend bitcoin within script using unrolled functions as well as being able to link predicate systems and an extended multi-script system. The power of this system can be greatly extended and developed using compilers. These would simplify the

intimate process and allow for separate script functions. Small decompiled functions can be run within each script using the extensibility of the predicate logic system. Here, individual scripts can be included into a larger function running in parallel. The combination of unrolled scripts the creation of a class of languages defined within the scripting system will allow for the extension of parallelised computations. Each invalid and false script act as a false signal in an extended set of functions. These can be simultaneously fed into other transactions depending on the input. Only valid scripts can send to the next block or input into another transaction extending the capabilities of the system. Bitcoin can hence incorporate a computable function set that extends to the primitive recursive. In this paper, we have documented the foundations for a programming language based within bitcoin script that can be extended to operate endlessly. We have demonstrated that bitcoin is both a decider and a larger scale touring machine able to compute anything that is computable the decidable within any standard limitations within time-space constraints. This works within script to the point of allowed functions and externally to the script as long as the Blockchain runs.

In future papers, we will extend this research to include multi-sorted predicate calculus. Using bitcoin script and transactions, this can be coded into a single sorted predicate function. Coupled, each of the systems can form the input into a perception based neural network. The inability to handle scripts that fail is overrated. Firstly, the ability to write multiple script constructs hooked into OP_Return values coupled with parallel computational systems leads to the creation of a rich environment. These unrolled scripts coupled with interlocking transactions offers the capability to provide a richer structure than can be attained using first-order predicate logic alone. The ability to create parallel transactions enables many sorted first-order logic constructs to be reduced into a single sorted first-order logic function. In this way, the single sorted theory can use a unary predicate symbol for each sort applicable to values are many sorted theory. We can state the axiom that these unary predicates act to partition discourse domains within our system.

# 4 References


1. Autebert, J., Berstel, J., & Boasson, L. “Context-Free Languages and Push-Down Automata”, in: G. Rozenberg, A. Salomaa (eds.), Handbook of Formal Languages, Vol. 1, Springer-Verlag, 1997, 111-174.
2. Bailey, Chris (2000). "Inter-Boundary Scheduling of Stack Operands: A preliminary Study" (PDF). Proceedings of Euroforth 2000 Conference.
3. Bohm, C. – Jacopini, G., “Nuove tecniche di programmazione semplificanti la sintesi di machine universali di Turing”, Rend. Acc. Naz. Lincei, serie VIII, Vol.32 fasc.6, giugno 1962, pp. 913-022.
4. Shannon, Mark; Bailey C (2006). "Global Stack Allocation : Register Allocation for Stack Machines" (PDF). Proceedings of Euroforth Conference 2006.
5. Davis, M. “Computability and Unsolvability”, McGraw-Hill, New York 1958.
6. Ginsburg, S.; Greibach, S. & Harrison, M. (1967) “Stack Automata and Compiling”, JACM, 14 (1967), pp. 172–201
7. Ginsburg, S.; Greibach, S. & Harrison, M. (1967) "One-Way Stack Automata". J. ACM. 14 (2): 389–418
8. Hermes, H., “Aufzahlbarkeit, Entscheidbarket, Berechenbarkeit”, Springer Verlag, Berlin, 1961.
9. Hopcroft, J.& Ullman, J. (1967). "Nonerasing Stack Automata" (PDF). J. Computer and System Sciences. 1 (2): 166–186.
10. Hopcroft, J.; Motwani, R. & Ullman, J. (2003). Introduction to Automata Theory, Languages, and Computation. Addison Wesley. Here: Sect.6.4.3, p.249
11. Ianov, Yu. I., “On the equivalence and transformation of program schemes”, Doklady Akad. Nauk S.S.S.R., 113, pp. 39-42, (1957), (in Russian).

12. Koopman, Philip (1994). "A Preliminary Exploration of Optimized Stack Code Generation" (PDF). Journal of Forth applications and Research. 6 (3).
13. Lee, C. Y., “Automata and finite automata”, Bell Syst. Techn. Journal, 30 (1960), Sept., pp. 1267-1295.
14. Robinson, J., “General recursive functions”, Proc. Amer. Math. Soc., 1, pp. 703-718, (1950).
15. Shannon, Mark; Bailey C (2006). "Global Stack Allocation : Register Allocation for Stack Machines" (PDF). Proceedings of Euroforth Conference 2006.
16. Shepherdson, J.C.- Sturgis, H.E., “Computability of recursive functions”, J. of the Ass. Comp. Mach., Vol. 10 (1963), pp. 217-255.
17. Sipser, M. (1997). Introduction to the Theory of Computation. PWS Publishing. Section 2.2: Pushdown Automata, pp. 101–114.
18. Smullyan, R. M.: “Theory of Formal Systems”, Annals of Mathematics Studies, Number 47, Princeton, 1961.
19. Stearns, R., Hartmanis, J. & Lewis, P. “Heirarchies of memory limited computations” IEEE Conference Record on Switching Circuit Theory and Logical Design, Institute of Electrical and Electronics Engineers, New York (1965), pp. 179–202
20. Turing, A. M. “On computable numbers with an application to the Entscheidungs-problem”, Proc. Lond. Math. Soc. {2}, 42 (1936-7), pp. 230-265: addendum and corrigendum, 43 (1937), pp. 544-536.
21. Wang, H., “A variant to Turing’s theory of computing machines”, J. ACM 4 (1957), pp. 63-92.
22. Cockshott, P, Michaelson, G (2012) “Tangled Tapes: Infinity, Interaction and Turing Machines”, https://www.semanticscholar.org/paper/Tangled-Tapes-Infinity-Interaction-and-Turing-Mach-Cockshott-Michaelson/19a6bcb7edccff58b6a2d5ba6a451b9ad4956312

## 4.1 Websites

Bitcoin scripting:

https://en.bitcoin.it/wiki/Script
http://davidederosa.com/basic-blockchain-programming/bitcoin-script-language-part-one/

Fortran Programming

http://fortranwiki.org/fortran/show/HomePage
https://gcc.gnu.org/fortran/