Beyond Godel

Author: Craig Wright

Abstract

In this paper, we start by defining the basic predicate systems used by Gödel in his logical constructions for the creation of a system of computable mathematics. We demonstrate how each of these predicates and the primitive recursive functions can be mapped directly into bitcoin script operations. This is then extended to explore the dual stack 2PDA construction within bitcoin. In this paper we use this extension to demonstrate how the integration of these functions across a dual stack push down automata (2PDA) allows us to create a system that is equivalent to a Turing machine and which can hence handle all grammatical constructs that may be processed within a Turing machine. The function and operation of the bitcoin operational codes and the construction of the stacks leads to different operational conditions than a standard Turing machine, however, it is also noted how this differs from a standard modern registered machine in operation. Ignoring stack limitations we can then see that any computable function may be integrated into operation and solution within bitcoin scripts.

Keywords: Bitcoin, Turing Machines, Unrolled recursion,

Introduction

Many misconceptions as to the notion of what constitutes a Turing machine and a Turing compatible language have evolved within the computer science community (Cockshott & Michaelson, 2012). Over time, the notion of a finite but unbounded machine has devolved into a perceived requirement for an infinite tape. This concept belies the notion of what is required for effective computability. Any system that is effectively computable or that can be computed within a Turing machine must halt. All such systems unnecessarily finite. The difficulty of course lies within the realm of determination. The halting problem leaves us in a state where we are unable to determine whether any particular problem will halt or not for any particular input.

In the development of the systems we have documented within this paper, we do not solve the halting problem, rather we hand it off to an associated compiler. In this, the determination of whether a script will halt or not is removed from the end script and determined within the build process. In any system where a complete, or total script is able to be created, we know that the script will halt within a bounded timeframe. The difficulty remains however that we cannot determine in advance whether any script will be able to be compiled for the particular input.

J. B. Rosser (1939) addresses the notion of "*effective computability*" as follows: "*Clearly the existence of CC and RC (Church's and Rosser's proofs) presupposes a precise definition of 'effective'. 'Effective method' is here used in the rather special sense of a method each step of which is precisely predetermined and which is certain to produce the answer in a finite number of steps*" (Rosser, 1939). Hence effective" is used terms of "1a: producing a decided, decisive, or desired effect", and "capable of producing a result" (Webster).

In the following paper, "effectively calculable" is defined to represent the state "produced by any intuitively 'effective' means whatsoever" and "effectively computable" will be used to represent "produced by a Turing-machine or equivalent mechanical device". Turing's "definitions" given in a footnote in his 1939 Ph.D. thesis Systems of Logic Based on Ordinals, supervised by Church, are essentially the equivalent:

> "*We shall use the expression 'computable function' to mean a function calculable by a machine, and let 'effectively calculable' refer to the intuitive idea without particular identification with any one of these definitions.*" (Turing, 1939).

The thesis can be stated as follows:

- Every effectively calculable function is a computable function.[1]

Turing stated it this way:

> "*It was stated ... that 'a function is effectively calculable if its values can be found by some purely mechanical process.' We may take this literally, understanding that by a purely mechanical process one which could be carried out by a machine. The development ... leads to ... an identification of computability with effective calculability.*"

Consequently, we now know that any effectively calculable system can be created within a bitcoin script. This is demonstrated within this paper. An effectively calculable system does not mean that as an economically calculable system and nor does this mean that it will be mapped within any bitcoin script that would be run within the existing system. For instance, even where the current limitations on script size to be removed, it would be possible to create a bitcoin script so large that no miner would

[1] Gandy (Gandy 1980 in Barwise 1980:123) states it this way: What is effectively calculable is computable. He calls this "Church's Thesis".

accept it. Such a script would be known to hold and could be provably demonstrated to do so while simultaneously being so large as to be economically infeasible to run.

Turing machines revisited.

First, we define any Turing complete program to be a program that halts. Whilst it is true that we cannot determine in advance if any particular program will halt for any set of input data, we do know that a program must halt on a system that is Turing complete if it is indeed decidable and that only decidable programs are known to run to completion on a Turing complete system. As such, we know that there is necessarily a point where the decidable program halts.

The result is that all decidable programs must halt within some time bounded finite period. We can also say that no Decidable program will run infinitely on a Turing complete system or that when loaded on a Turing complete system that all decidable programs are finite.

From this, we can deduce that all Turing Complete programs and functions form a subset of the set of all possible programs. The set of all possible programs includes both those that halt on a Turing machine (that are decidable) as well as those that would run infinitely looping without end. We can show that the set of infinite and undecidable programs is not an empty set. It is a simple exercise to construct a program that will run indefinitely on an infinite tape. Such a program, given an infinite time to run will never halt. It is left to the reader to imagine a simple program that will infinitely recurse (i.e. never halts). From this, we now know that the set of all Programs, P(N) must be larger than and contain the set of decidable programs (N).

Any program that is decidable must halt. Consequently, it is bounded in time. We can hence state that for any program P(T) that is decidable and is run on a Turing Complete system that it is required to halt at some point in time. As such, the set of all programs of the form P(T) in the Set (N) that halt for some input must halt in finite time.

It is noted that although we cannot determine if every program is decidable with certainty, we can develop a Total Turing Machine that requires that all programs are “unrolled” and hence are determinable.

Although all programs need not be finite, all programs that are decidable will halt on a Turing machine. More, we can say that a Turing machine will run all decidable programs and only the set of all decidable programs. The set of programs that run on a Turing machine and halt that are not decidable is a NULL set.

If we take Wolfram’s conjecture of a (2,3) Universal Turning Machine (Smith, 2007), and use the logic in (wright) we can show that the predicate logic system deployed in Bitcoin as a scripting language, we see that Bitcoin is functionally a system that is known as a Total Turing Machine.

As is demonstrated by (Wright), Bitcoin’s Scripting language can recurse.

The language in (Wright) is more extensible than in (1, Smith). We can thus map directly the conjectures of Smith (2007) onto a system designed using the recursion system (wright) within Bitcoin.

The problem is not whether a computer can be built that can run any conceivable decidable program (i.e. that halts), it is now the program of determining the most effective means to minimise the size of the decidable program for when a recursive loop structure is “unrolled” it can grow exponentially in size.

For the set of all programs, it is not possible to determine if a program is decidable and will halt or even if a system can run it to completion within the time/space constraints available when run on a standard Turing Machine.

A Total Turing Machine however only accepts input that is defined within a bounded space/time parameter that is constructed and set in advance.

The foundation of recursion

Following Godel's Axiom, all mathematics that is decidable (from this we can say anything of any of use in science, engineering and finance) can be solved using on the following six (6) constructs.

1. $C_A(x)$, Where this represents the Characteristic function of A.
2. $S(x) = x+1$ (Successor Function)
3. $U_i^n(x_1,...,x_n) = x_i, \quad 1 \le i \le n.$ (Identity Functions)
4. $x + y$
5. $x \dot{-} y$ (Monus function)
6. $x.y$.

(1) and (3) can be created and unrolled using commands (Such as OP_PICK). In this paper, we shall define the basic components of the $\phi - \text{Re}\,cursive$ system that exists within Bitcoin script.

Characteristic function of A.

The characteristic function is a powerful tool for the analysis of algebraic functions of sets. This function is $C_A(x)$, Where this represents the Characteristic function of A.

Definition: We let A be a subset of E where x is any member of E and f_A is the function defined as:

$$f_A(x) = \begin{cases} 1 & if \ \ x \in A \\ 0 & f \ \ x \notin A \end{cases}$$

The characteristic function $C_A(x)$ is hence a predicate logic function. OP_Code in script for this function will need to be created for each individual system being evaluated. The simple example above does not account for stack values that are selected as a virtual registry and there are many enhancements that can be made on this script. The Ackerman function was a particular implementation (nChain paper). Most systems that will be implemented are not going to require this depth and will require only primitive recursive functions.

Successor Function

(2) is simply OP_1ADD and is defined.

<A> OP_1ADD

Identity Function (3)

More generally, $\forall\ (m, n > 0)$, $\exists$ a primitive recursive function $\exists\, s_n^m\ of\ (m+1)$ arguments that behaves as follows:

$\forall$ Gödel number **p** of a partial computable function with $(m+n)$ arguments, and all values of $x_1,\ldots,x_m$:

$$\varphi_{s_n^m(p,x_1,\ldots,x_m)} \square\ \lambda y_1,\ldots,y_n.\varphi_p(x_1,\ldots,x_m,y_1,\ldots,y_n)$$

The function **s** described above can be taken to be s_1^1. Kleene (1952) uses U_i^n to indicate the identity function over the variables x_i whereas Boolos, Burgess, & Jeffrey (2007) use the identity function id_i^n over the variables x_1 to x_n. For example, the function would select from a list as follows:

1. $U_1^1(a) = a$
2. $U_2^3(\ b, c, a\ ) = c$
3. $U_2^7(\ r, s, t, u, v, w, x\ ) = s$

In effect, what we are doing in U_i^n is selecting item 'n' from a list of 'm' in length.

Using an additional variable, we can also select from positions lower in the stack. The simplest implementation (without safety checks etc.) would be to use the following script on an existing stack variable:

<i> OP_PICK // Copy Xi to the top of the stack

In this example, the 'n' stack items $x_1, \ldots, x_n$ would already need to exist on the stack.

What is of particular interest is that <i> can be defined in a function and used subsequently in the selection from a list.

Basic Functions

(4) is OP_ADD. This is defined in script as: <A> <B> OP_ADD

(6) is OP_MUL. This is defined in script as: <A> <B> OP_MUL

OP_MUL (6) is a disabled code right now, but we have created an alternative means to do this and it is in the patent list (Ref.).

The Monus Function

(5) is the Monus function. This is:

$$a \dot{-} b = \begin{cases} 0 & \text{if } a < b \\ a - b & \text{if } a \geq b \end{cases}$$

Using a conditional, OP_IF, we can construct this. I have sent this to Allan and Stef to code into a high-level language that creates the primitive in script for users.

Primitive Recursive Functions

With a basis that is founded by using the most elemental primitive recursive functions, it is possible to construct more complex primitive recursive functions. This is achieved using functional composition coupled with primitive recursion. This section lists and derives a few basic operations using functional composition.

The main Primitive Recursive Functions

Starting from the simplest primitive recursive functions, we can build more complicated primitive recursive functions by functional composition and primitive recursion. In this entry, we have listed some basic examples using functional composition alone. In this entry, we list more basic examples, allowing the use of primitive recursion:

1) $Add(x,y) = x + y$

$Add(x,\phi) = id(x)$

$Add(x,n+1) = s(Add(x,n))$

2) $Mult(x,y) = xy$

$Mult(x,\phi) = z(x)$

$Mult(x,n+1) = Add(x, Mult(x,n))$

3) $p_2(x) = x^2$

A.k.a $Mult(x,x)$

$p_{m+1}(x) = Mult(x, p_m(x))$

- This is primitive recursive by induction on m.

4) $\exp_m(x) = m^x : \exp_m(\phi) = s(\phi)$

$\exp_m(n+1) = mult(const_m(n), \exp_m(n))$

5) $\exp(x,y) = x^y : \exp(x,\phi) = const_1(x)$

$\exp(x,n+1) = mult(x, \exp(x,n))$

6) $fact(x) = x! : fact(\phi) = s(\phi)$

$fact(n+1) = mult(s(n), fact(n))$

$\rightarrow$ factorial

7) $Sub_1(x) = x \dot{-} 1 := \begin{cases} \phi & if \quad x = 0 \\ x-1 & otherwise \end{cases}$

Constructed using

z and $Sub_1(\phi) = z(\phi)$

$Sub_1(n+1) = s(Sub_1(n));$

→ Monus function

8) $Sub_m(x) = x \dot{-} m,$

$Sub_m = Sub_1^m$

9) $Sub(x,y) = x \dot{-} y : Sub(x,\phi) = id(x)$

$Sub(x,n+1) = Sub_1(Sub(x,n))$

10) $diff(x,y) = |x-y| : Sub(x,y) + Sub(y,x)$

11) $d_\phi(x) := \begin{cases} 1 & if \quad x = \phi \\ \phi & otherwise \end{cases}$

This is constructed using $const_1$ and

$z : d_\phi(\phi) = const_1(\phi),$ and
$d_\phi(n+1) = z(d_\phi(n))$

12) $dm(x) := \begin{cases} 1 & if \quad x = m \\ \phi & otherwise \end{cases}$

This function is primitive recursive as it is:

$d_\phi(diff(x, const_m(x)))$

13) $d_s(x) := \begin{cases} 1 & if \quad x \in s \\ \phi & otherwise \end{cases}$

Where $s = \{a_i, \ldots, a_m\}$ is primitive recursive for its:

$d_{a_i} + \ldots d_{a_m}$

14) $\operatorname{sgn}(x) := \begin{cases} \phi & if \quad x = \phi \\ 1 & otherwise \end{cases}$

Also expressed as:

$$Sub\left(const_1(x), d_\phi(x)\right)$$

15) $rem(x,y) := \begin{cases} \phi & if \quad x = \phi \\ x \bmod y & otherwise \end{cases}$

Note:

$x \bmod y$ is the remainder of x / y

If we suppose $y = \phi$, then $\phi \bmod y = \phi$.

Further,

$$(n+1) \bmod y = \begin{cases} \phi & * \\ s(n \bmod y) & otherwise \end{cases}$$

* $diff\left(s(n \bmod y), y\right) = \phi$

Thus: $rem(\phi, y) = z(y)$

And

$$rem(n+1, y) = \operatorname{sgn}(y)\left(rem(n,y)+1\right)$$

$$\operatorname{sgn}\left(|\, rem(n,y)+1-y\,|\right)'$$

$$= mult(\operatorname{sgn}(y), mult(s)(rem(n,y)), \operatorname{sgn}(diff\,(s(rem(n,y)), y))))$$

$$= g\left(y, rem(n,y)\right)$$

Where we have:

$$g(x,y) := mult(\operatorname{sgn}(y)), mult(s(x)), \operatorname{sgn}(diff\,(s(x), y))))$$

We include $\operatorname{sgn}(y)$ to account for the instance of $y = \phi$.

16) $g(x,y) = \begin{cases} quotient\ of\ x/y & if \quad y = \phi \\ \phi & otherwise \end{cases}$

We can test that q is primitive recursive using the equation:

$$x = yq(x,y) + rem(x,y)$$

This is obtained using the division algorithm for integers.

From this:

$$yq(x,y) + rem(x,y) + 1 = x + 1$$

$$= yq(x+1,y) + rem(x+1,y)$$

Simplifying we obtain:

$$y(q(x+1,y) - q(x,y)) = rem(x+1) + 1 - rem(x+1,y)$$

Thus, we obtain:

$$q(n+1,y) = \begin{cases} q(n,y)+1 & if \quad rem(n,y)+1 = y \\ q(n,y) & otherwise \end{cases}$$

Hence:

$$q(\phi,y) = z(y)$$

And further:

$$q(n+1,y) = \text{sgn}(y)(q(n,y) + \text{sgn}(diff\,(s(rem(n,y)),y)))$$

In this, $\text{sgn}(y)$ takes the case of $y = \phi$ into consideration.

Note:

If we can recall that $S \subseteq N_n$ is referred to as being primitive recursive if its characteristic function Φ is primitive recursive.

Taking $s = \{m\}$, then $\Phi s = d_m$

Next, the predicate $\Phi(x)$ over N_k is primitive recursive where its associated set $S(\Phi) := \{x \in N_k \mid \Phi(x)\}$ is also primitive recursive.

The functions presenting in this section are examples of elementary recursive functions that can be used to create far more complex systems. We can use the notion of Bounded maximization to prove the privative recursive nature of the quotient and the remainder functions. These are native scripts in Bitcoin that are currently disabled:

OP_MOD	151	0x97	a b	out	Returns the remainder after dividing a by b.
OP_MUL	149	0x95	a b	out	a is multiplied by b.
OP_DIV	150	0x96	a b	out	a is divided by b.

We have now demonstrated that any primitive recursive function can be created within an unrolled bitcoin script. Using these massive recursive functions we will now extend the functionality of the system to incorporate the dual stack nature of the system. With an analysis of the minimal stack

machine, we can start to demonstrate how bitcoin can be incorporated into a series of novel script constructs that each increase the power of the system.

A Minimal Stack Machine

We will start in defining a simple stack. This will allow us to create a Total system that is Turing Complete. This is defined by Sipser (1996) using the terminology decider or alternatively as a Total Turing machine (Kozen, 1997). In this, we defined the stack discipline that is needed to bound the system. An n-ary function f computed on a stack S will require the arguments it uses to be on the stack and on RETURN it will replace these to create a new stack construct t :

$$S = a_{1,\ldots}a_{n,}\ u \Rightarrow t = f\left(a_1,\ldots,a_n\right),\ u \qquad \textit{Equation 1}$$

The stack S is the primary stack in the bitcoin system. If used, we will call the Return (or ALT) Stack By 'r'). This is a Turing complete system. The machine consists of the compiler, the script and the system that runs it. Only Total code is created by the compiler, but we cannot decide if the code will halt in all cases before it is compiled for the given input.

On the stack $1 \le \mathrm{i} \le n$ such that $a_i = \left(s\right)_{i-1}$ for any 2-position list indexing function of the form.

$$\left(a,s\right)_0 = a \qquad \left(a,s\right)_{i+1} = \left(s\right)_i \qquad \textit{Equation 2}$$

The element $\left(s\right)_0 = H\left(s\right)$ is the Top Stack item.

In this we can say:

$$\left(s\right)_n = \overbrace{HT\ldots.T}^{n}\left(s\right) \qquad \textit{Equation 3}$$

Our Stack machine expects a script that acts as a program which is defined to be an ordered set of instructions that operate on and alter a Stack of natural numbers (the Stack Set). This machine is Turing Complete IFF [2]a decidable program can be run on the Stack machine when that program is also computable on a Turing Machine.

We can only compute a function on the stack machine that is decidable, that is it satisfies the Church (1937) Thesis.

Our most basic commands are:

PUSH:

OP_PUSH DATA (1,2,4) $\qquad s \Rightarrow 0, s \qquad$ *Equation 4*

POP:

The top item is removed;

OP_Drop $\qquad a, S \Rightarrow S \qquad$ *Equation 5*

Bitcoin also extends the power of the system with the implementation of OP_2DROP and the related stack operations.

[2] IF and only IF

$Incr_{(i)}$:

Increase the i^{th} stack element by a value of 1.

$$s = \overset{i}{\ldots}, a,\ s_1 \Rightarrow t = \overset{i}{\ldots}, a+1, s_1 \qquad \textit{Equation 6}$$

$Decr_i(p)$:

$if\ (s)_i > 0$ decreases the i^{th} stack element by a value of 1 and preforms p ; $Decr_i(p)$.

Otherwise do nothing (Else). This is defined in detail in the Appendix.

Boot Strapping the Stack Machine

We can clear the i^{th} element of the Stack using:

$$Decr_i(Push; Pop)$$

First, clearing the i^{th} stack item involves an $(s)_i = 0$ operation.

This is:

$$Decr_i(Push; Pop)$$

And is completed within bitcoin Op_Codes. See the Appendix for more information.

OP_0 // (Push 0 to the stack)

OP_Roll (i) $//\ m := n = i+1$

OP_Roll (i) $//\ m := n-1 = i$

OP_Roll (i) $//\ m := n-2 = i-1$

OP_Roll (i) $//\ m := n-i = 1$

This is not optimised and for small code, it can be implemented immediately using code such as OP_2ROT etc.

$Decr_i$

These functions can be implemented as a complete machine.

To replace the j^{th} element of S to the i^{th} one, we use:

$$(s)_i := 0; Push; Decr_{j+1}(Incr_o; Incr_{i+1})$$

$$Decr_0(Incr_{j+1}); Pop$$

Also:

If $(s)_i > 0$ Then P Else Q by:

$$Push; Incr_0; Push;$$

$$(s)_0; = (s)_{i+2};$$

$$Decr_0\left((s)_0 := 0; p^{+2}; (s)_1 := 0\right);$$

$$Decr_i(q^{+2}); Pop; Pop$$

Here p^{+2} relates to the stack value p where each stack index is increased by 2.

Therefore: $p = Incr_{3i}(s)_6 := (s)_8 \Rightarrow p^{+2} = Incr; (s)_8 := (s)_{10}$;

Turing Complete

We now extend our minimal machine into the computation of functional terms. As above, these are the minimal set of expressions formed using $\vee$, n (an integer) by

- $Incr(a)$,
- $Decr(a)$,
- $Head(a)$,
- $Tail(a)$,
- $Tail(a)$,
- $Pair(a,b)$,
- $IF(a,b,c)$,
- $Apply(a,b)$, and
- $R(a)$

In this operation set, a, b and c are previous constructed functional terms.

A Turing computable (or decidable) function f can be computed in an evaluation of a functional term of f.

Using the unary contractions:

$\langle f \rangle(x)$ we have

$$\langle f \rangle(x) = f((x)_o, ..., (x)_{n-1})$$

Hence:

$$f(x_1, ..., x_n) = \langle f \rangle(x_1, ..., x_n, 0)$$

Functional Terms are created with a pair of variables V such that the single argument:

v of $\langle f \rangle(v) = r$

Here:

R represents the BODY.

This is the functional term:

r of $\langle f \rangle$

Thus:

$R(a)$ a occurring within a represents the recursive cell $\langle f \rangle(\alpha)$

We can thus represent the functional term Add_t here:

$$\langle Add \rangle(x) = (x)_o + (x)_1$$

And

$$\begin{aligned} if \quad & (Var(0), \\ & Incr\ R \quad Pair(DecrVar(0), \\ & \qquad\qquad Pair(Var(i), 0)), \\ & \qquad\qquad Var(1)) \end{aligned}$$

In this expression, $Var(i)$ represents the extended term:

$$\underset{\square\square\square\square i \square\square\square\square}{Head\ Tail.....Tail}\ (v)$$

Accessing the $(i+1)^{st}$ variable of f.

A functional term a denotes a number in an assignment of a number v to the variable V and a functional term r to the variable R.

Denotes relates to a value and evaluates to.

From this we can define the 3-place denotation function

$[a]_r^v$ which yields the value for a.

Here

$$[Add - t]_{Add\_t}^{6,2,0} = 8$$

The expression $Apply(a,b)$ represents the application of the function with functional term a to the argument denoted by the term b:

Such that

$$\left[Apply(a,b) \right]_r^v = [a]_a^{[b]_r^v}$$

The denotation function is a partial function as the bitcoin predicate logic system returns invalid (that is, has no value) when a recursion does not terminate. This means the unrolled total function only returns a valid response when the function is known to terminate.

The Dual Stack

A 2PDA is not only a viable alternative to a standard Turing machine (TM), you can do everything that a TM can achieve. A 2PDA is a push down automata with two stacks. Bitcoin implements this using a primary stack with a standard LIFO (last in first out) operation that is extended with commands (including OP_PICK) and unable stack operations to be conducted in an alternate order. It is widely known (*) that a two PDA is functionally equivalent to a TM and can simulate a single or multi-tape TM just as a TM can simulate a 2PDA.

Other researchers (*), have demonstrated that a deterministic 2PDA with only the use of a single state can simulate a TM. Bitcoin is generally deployed as a simple predicate module that acts at best as a nondeterministic PDA. That is, a 2PD it uses only the primary stack (and ignores the ultimate stack). In this we can model that is a simple machine without having to resort to the added power of the script primitives. We shall start by defining the most basic of 2PDA formats that can be deployed in bitcoin script. In this model we will use only the LIFO nature of the stack.

Definition

We will start with an arbitrary stack alphabet ζ .

Next, we find the primary stack as α in the alternative (ALT) stack using ζ in ζ^* . These will be defined to be juxtaposed to each other according to their relative stack height. The primary stack α is defined as our lower or less stack and the β stack or ALT stack is the upper or right stack. We shall ensure that the position in the machine is well defined by defining α and β to be members of a set of disjoint copies of ζ^* . In this, $\alpha \in \zeta^*$ for the primary stack and $\beta \in \zeta^*$ for the ALT stack. The concatenation of the primary and the ALT stacks can be referred to as the system stack.

We can imagine the equivalence in considering the first stack as the contents of the tape to the left of the current position, and the second as the contents to the right on an arbitrary Turing machine. In this system, we start by pushing the normal bottom of stack markers on both stacks allowing us to simulate the TM eye-popping from the right stack and pushing to the left in order to move right. We engage in the opposite in order to move left. Where we reach the bottom of the less stack we need to behave accordingly either halting or rejecting wearers we hit the bottom of the right stack we just push a blank symbol onto the left.

Using the assumption that the Turing machine has an infinite tape in one direction that extends infinitely to the right cover we can use one stack to represent the tape content on the finite portion of the tape to the left of the head with another representing the content on the finite nonblack portion of tape to the right of the head. The two stack PDA simulates a movement of the Turing machine by appropriately pushing and popping the two stacks.

Unlike a Turing tape, we do not use a storage cell but incorporate a stack pointer.

Definition

The 2PDA, $\langle Q, \mathrm{I}, F, \zeta, \Omega, \delta \rangle$ over $\sum$ is defined and constructed by:

- the set of Q states having subsets $\mathrm{I}, F \subseteq Q$ of initial and final states,
- the stack alphabet ζ constructed using the finite set $\Omega \subseteq \zeta^* \times \zeta^*$ of initial stack values,
- the joint finite transition relations:

$$Q \times (\zeta + \varepsilon) \times (\zeta + \varepsilon) \xrightarrow{a\delta} Q \times^* \zeta^* \times \zeta^*$$

$$\forall a \in \sum + \{\varepsilon\}$$

Here, all $\varepsilon - transitions$ of the type

$$Ux \xrightarrow{\varepsilon} Ux\varepsilon$$

or

$$xV \longrightarrow \varepsilon xV$$
$$for \qquad U,V \in \zeta + \{\varepsilon\}$$

and

$x \in \zeta$ as right, respectively left moves.

Each element of $Q \times \zeta^* \times \zeta^* \times \sum^*$ is designated as the configurations, the initial ones being part of $I \times \Omega \times \sum^*$ in the final configurations being a part of:

$$F \times \zeta \times \zeta \times \{\varepsilon\}$$

or

$$Q \times \{\langle \varepsilon, \varepsilon \rangle\} \times \{\varepsilon\}$$

or

$$F \times \{\langle \varepsilon, \varepsilon \rangle\} \times \{\varepsilon\}$$

as determined from the required mode of acceptance.

This form of word acceptance is directly attributable to the single stack PDA the addition of accepting by final state, empty stack or a combination thereof.

Theory: the basic pure 2PDA in bitcoin is mapped to an equivalent iTM

Definition: a pure 2 PDA

A 2PDA is pure if and only if:

- it has only one state, the initial;
- accepts only by empty stack;
- only single stack operations are conducted.

For this, we are ignoring the additional steps and operations available (such as OP_PICK) within bitcoin script and are working only with the LIFO stack. The complete bitcoin script language is indeed more powerful, but is not necessary to demonstrate that any iTM can be created in our pure 2PDA.

Proof

A TM can access only one cell tape at any time. Our 2PDA may access and view up to 2 storage locations. We hence seek to track the state of the iTM in one stack element of our 2PDA. We simulate the actions of the iTM using these stacks.

First, we begin by defining our iTM:

$$M = \langle Q, \Im, \delta \rangle$$

The nature of the 2PDA this set as:

- the stack alphabet $\zeta = \varepsilon = \Im \times (Q \times \Im)$
- a single (the primary) stack to begin
  - $\#\langle q_r, \# \rangle$
- Transitions
  - $\#\langle q_r, \# \rangle \xrightarrow{b} b\langle q_r, \# \rangle$

  And

  - $a\langle q_r, \# \rangle \xrightarrow{b} ab\langle q_r, \# \rangle \qquad \forall a, b \in \Sigma$

  that read the input into the primary stack.
- Transitions
  - $u\langle q_r, \# \rangle \xrightarrow{b} \varepsilon\langle q_0, u \rangle \qquad \forall u \in \Sigma + \{\#\}$

  that nondeterministic lease switch into computation mode.

An M-transition $\langle p, x \rangle \longrightarrow \langle q, y, @ \rangle$ using $p, q \in Q$, $x, y \in \Im$ and $@ \in \{L, N, R\}$ can be simulated with the use of:

(L) $u\langle p, x \rangle \longrightarrow \langle q, u \rangle y, \quad u \in \Im$

Resp. $\varepsilon\langle p, x \rangle \longrightarrow \varepsilon\langle q, \# \rangle y$

(N) $u\langle p, x \rangle \longrightarrow u\langle q, y \rangle, \quad u \in \Im + \{\varepsilon\}$

(R) $u\langle p, x \rangle \longrightarrow u\langle q, v \rangle \varepsilon, \quad u \in \Im + \{\varepsilon\}$

$\langle p, x \rangle \longrightarrow y\langle q, v \rangle, \quad v \in \Im$

Resp. $\langle p, x \rangle \varepsilon \longrightarrow y\langle q, \# \rangle$

The simulation of transitions on our iTM with the right move requires two steps, all others but one. The initial step moves the state information to the top of the primary stack. The second step only relates to the transitions that are expected to be found in this location. At this point, the 2PDA is not capable of realising transitions that are not directly corresponded to the M-transitions.

When q_f it is found in the ALT stack, we raise the primary stack first then the ALT.

$$\begin{aligned} u\langle q_f, x \rangle &\longrightarrow \varepsilon\langle q_f, x \rangle \\ \varepsilon\langle q_f, x \rangle &\longrightarrow \varepsilon\varepsilon \\ \varepsilon x &\longrightarrow \varepsilon\varepsilon \end{aligned}$$

$\forall u, x \in \Im$

At this point, we have acceptance by the empty stack. The predicate system can be extended to not only incorporate a series of logic acceptances but also to output different values when incorporated into OP_RETURN statements. The use of several OP_RETURN statements is currently considered non-standard within bitcoin, however this does not preclude its use within script edges lowers the rate of acceptance in any individual block.

Conclusion

Using these alone, we can define any partial-recursive ($\phi - \mathrm{Re}\,cursive$) function, this is any function that is decidable. Turing's original paper (*) did not note that the tape needed to be infinite. Rather, it was unbounded. Turing said "*The machine is supplied with a 'tape', (the analogue of paper) running through it, and divided into sections (called 'squares') each capable of bearing a 'symbol'. At any moment there is just one square, say the r-th being the symbol s(r) which is 'in the machine"*".

The length of the tape is not specified. The process does have to halt if the system is Turing complete. Hence, with the requirement that such a machine is "*calculable by finite means*", we express a finite but unbounded tape. In bitcoin, the limits imposed on the system limit the functional length of any script. It is conceivable that without these limits, and unbounded script could be created. It was Emil Post's eponymous machine (*) but first implemented the concept of an infinite tape, not the Turing machine. This error has provided much of recent computer science. All digital computers are finite, just as all calculable problems must halt. The difficulty is in deciding if a problem is calculable within finite time. This problem has not of course mean solved or removed when applied in bitcoin, rather the problem has been transformed. If a compiler can create a script within the bitcoin language, we know that it halts. However, we do not know if the creation of the script is either possible or feasible until the result is known.

In this paper, we have demonstrated that bitcoin's 2PDA is capable of computing any value that is computable in a system compatible with that of Godel's logic system. Consequently, we have demonstrated that bitcoin script language is Turing complete. The relative power and functional implementation of a problem using bitcoin script is a separate issue. In a paper to follow this one, we shall extend Godel's predicate system using bitcoin script operations. In being able to compile a script in bitcoin language, we have a system that always works within a defined bound. A deterministic and matched nondeterministic optimisation problem can be created within such bounds.

The richness and power of bitcoin scripting language has been overlooked due to the complexity of the system. In this paper we have demonstrated the true power of the system and how a complete implementation can create a script of a determined length and known maximum processing difficulty.

References

1. Boolos, G., Burgess, J. & Jeffrey, R., (2007), "Computability and Logic" Fifth Edition, Cambridge University Press, Cambridge, UK. Cf pp. 70–71.
2. Church, A. (1935) Abstract No. 204. Bull. Amer. Math. Soc. 41, 332-333.
3. Church, A. (1936) "An Unsolvable Problem of Elementary Number Theory." Amer. J. Math. 58, 345-363.
4. Cockshott, P. & Michaelson, G., (2012) `Tangled Tapes: Infinity, Interaction and Turing Machines', Turing Centenary Conference: CIE 2012 How the World Computes, Cambridge, June 2012
5. Kleene, S., (1952) "Introduction to Metamathematics". Walters-Noordhoff & North-Holland
6. Kozen, D.C. (1997), Automata and Computability, Springer.
7. Meyer, A.R., Ritchie, D.M. (1967), The complexity of loop programs, Proc. of the ACM National Meetings, 465.

8. Minsky, M. L. (1967), “Computation: Finite and Infinite Machines”, Prentice-Hall, Inc. Englewood Cliffs, N.J.
9. Meyer, A.R., Ritchie, D.M. (1967), The complexity of loop programs, Proc. of the ACM National Meetings, 465.
10. Penrose, R. The Emperor's New Mind: Concerning Computers, Minds, and the Laws of Physics. Oxford, England: Oxford University Press, pp. 47-49, 1989.
11. Rosser, J. B. (1939). "An Informal Exposition of Proofs of Godel's Theorem and Church's Theorem". The Journal of Symbolic Logic. The Journal of Symbolic Logic, Vol. 4, No. 2. 4 (2): 53–60.
12. Sipser, M. (1996), Introduction to the Theory of Computation, PWS Publishing Co.
13. Smith, A (2007) “Universality of Wolfram’s 2, 3 Turing Machine)” Wolfram, https://www.wolframscience.com/prizes/tm23/TM23Proof.pdf
14. Turing, A (1937) "On Computable Numbers with an Application to the Entscheidungsproblem," Proceedings of the London Mathematical Society Series 2, 42: 230-265.
15. Turing, A. M., (1939), Systems of Logic Based on Ordinals (Ph.D. thesis). Princeton University. p. 8.
16. Merriam Webster's Ninth New Collegiate Dictionary

Appendix 1

Characteristic function of A.

// The following is the characteristic function for X in set (A1 to An)

// (in 1-byte format in this example)

```
OP_0    OP_0                        // Counter set
OP_PUSHDATA1        <X>             // Value to check <X>
OP_OVER                             // Stack {X}, {0}

OP_1ADD                             // Stack {X}, {1} (lower 0 ignored)
OP_PUSHDATA1        <A1>            // Stack {X}, {1}, {A1}
        OP_DUP                      // Stack {X}, {1}, {A1}, {A1}
        OP_TOALTSTACK               // Stack {X}, {1}, {A1}
        3 OP_PIC                    // Stack {X}, {1}, {A1}, {X}
        {X}, {A1} OP_EQUAL// Stack {X}, {1}, {0/1}
        OP_IF
                // Copy to save position Stack-4
        OP_ENDIF
        …
OP_PUSHDATA1        <A2>
        OP_DUP
        {A2} X OP_EQUAL
        OP_IF
                //
        OP_ENDIF
OP_TOALTSTACK
        …
OP_PUSHDATA1        <Ai>
        OP_DUP
        {Ai} X OP_EQUAL
        OP_IF
                //
        OP_ENDIF
OP_TOALTSTACK
        …
OP_PUSHDATA1        <An>
        OP_DUP
        {An} X OP_EQUAL
        OP_IF
                //
        OP_ENDIF
OP_TOALTSTACK

// Assuming we need to have the data pushed
// and it is not on the stack already
OP_PUSHDATA1        <i>             // Save i to the stack
OP_PUSHDATA1        <X>             // Save n to the stack
// Safety check – not defined here

                                    // Return 0 if A<B
        {i} OP_FROMALTSTACK                 // Copy Xi to the top of the stack
OP_ELSE                             // Otherwise Return A-B
        OP_RETURN                   // Optional – error in script….
```

Successor Function

(2) is simply OP_1ADD and is defined.

<A> OP_1ADD

Identity Function (3)

More generally, $\forall\ (m,n>0)$, $\exists$ a primitive recursive function $\exists\, s_n^m$ *of* $(m+1)$ arguments that behaves as follows:

$\forall$ Gödel number **p** of a partial computable function with $(m+n)$ arguments, and all values of $x_1,\ldots,x_m$:

$$\varphi_{s_n^m(p,x_1,\ldots,x_m)} \simeq \lambda y_1,\ldots,y_n.\varphi_p(x_1,\ldots,x_m,y_1,\ldots,y_n)$$

The function **s** described above can be taken to be s_1^1. Kleene (1952) uses $U_i^{\ n}$ to indicate the identity function over the variables x_i whereas Boolos, Burgess, & Jeffrey (2007) use the identity function $id_i^{\ n}$ over the variables x_1 to x_n. For example, the function would select from a list as follows:

1. $U_1^{\ 1}(a) = a$
2. $U_2^{\ 3}(\ b,\ c,\ a\ ) = c$
3. $U_2^{\ 7}(\ r,\ s,\ t,\ u,\ v,\ w,\ x\ ) =$ s

In effect, what we are doing in $U_i^{\ n}$ is selecting item 'n' from a list of 'm' in length. We can define this in Bitcoin Script as:

// The following is the Identity function for X1 to Xn (in 1-byte format in this example)

```
OP_PUSHDATA1    <X1>
OP_PUSHDATA1    <X2>
        …
OP_PUSHDATA1    <Xi>
        …
OP_PUSHDATA1    <Xn>

// Assuming we need to have the data pushed
// and it is not on the stack already
OP_PUSHDATA1    <i>             // Save i to the stack
OP_PUSHDATA1    <n>             // Save n to the stack
// Safety check
OP_DEPTH                        // Check Stack depth <d>
2 OP_ADD                        // Add '2' to <d> - depth for i,n
OP_LESSTHAN                     // Is 'n' < <d>
// The actual function – past safety checks
OP_IF                           // Return 0 if A<B
        {i} OP_PICK         // Copy Xi to the top of the stack
OP_ELSE                         // Otherwise Return A-B
        OP_RETURN               // Optional – error in script….
```

OP_ENDIF

Using an additional variable, we can also select from positions lower in the stack. The simplest implementation (without safety checks etc.) would be to use the following script on an existing stack variable:

<i> OP_PICK // Copy Xi to the top of the stack

In this example, the 'n' stack items $x_1,...,x_n$ would already need to exist on the stack.

What is of particular interest is that <i> can be defined in a function and used subsequently in the selection from a list.

Basic Functions

(4) is OP_ADD. This is defined in script as: <A> <B> OP_ADD

(6) is OP_MUL. This is defined in script as: <A> <B> OP_MUL

OP_MUL (6) is a disabled code right now, but we have created an alternative means to do this and it is in the patent list (Ref.).

The Monus Function

(5) is the Monus function. This is:

$$a \dot{-} b = \begin{cases} 0 & \text{if } a < b \\ a - b & \text{if } a \geq b \end{cases}$$

Using a conditional, OP_IF, we can construct this. I have sent this to Allan and Stef to code into a high-level language that creates the primitive in script for users.

This is defined in Script as follows:

// The following is a Monus function for A and B (in 1-byte format in this example)

```
OP_PUSHDATA1      <A>
OP_PUSHDATA1      <B>
// Assuming we need to have the data pushed
// and it is not on the stack already
OP_2DUP
{A} {B} OP_LESSTHAN                    // A & B are on the Stack
OP_IF                                  // Return 0 if A<B
        OP_2DROP
        OP_0
OP_ELSE                                // Otherwise Return A-B
        {A} {B} OP_SUB
OP_ENDIF
```

There are far more efficient methods to implement a Decrement() and Increment function that these here. The Bitcoin scripting language is rich and a well-constructed compiler would use commands including OP_PICK {n}, OP_ROLL {n}, and OP_2ROT in order to increase efficiency. This exercise is not about efficient construction, just a proof of concept.

$Incr_{(i)}$:

Increase the i^{th} stack element by a value of 1.

$$s = \ldots, a, \underset{i}{s_1} \Rightarrow t = \ldots, a+1, \underset{i}{s_1}$$

This can be implemented in Bitcoin script from a compiled function as simply as:

```
// Incr_i()- expand Increment.
OP_Depth ()
OP_IF  {
        Depth <= i
        End loop;
OP_Else {                       // Run the expanded and tested function
        OP_TOALTSTACK           // j=0    //move stack for operation
        OP_TOALTSTACK           // j=1
        OP_TOALTSTACK           // j=2
        OP_TOALTSTACK           // j=3
        …
        OP_TOALTSTACK           // j=i-1
        OP_TOALTSTACK           // j=i
        OP_ADD                  // Add one to value on stack
        OP_FROMALTSTACK         // j=i
        OP_ FROMALTSTACK        // j=i-1
        …
        OP_ FROMALTSTACK        // j=2
        OP_ FROMALTSTACK        // j=1
        OP_ FROMALTSTACK        // j=0    // return stack order
}
```

The thing to remember is that Bitcoin script is a Total Turing system, the compiler decides if a script can be created that will end. The secret is not in solving the halting problem, but in moving it to a separate system.

$Decr_i(p)$:

$if\ (s)_i > 0$ decreases the i^{th} stack element by a value of 1 and preforms $p\,; Decr_i(p)$.

Otherwise do nothing (Else).

This can be implemented in Bitcoin script from a compiled function as simply as:

```
// Decr_i(OP_X. OP_Y)   - expand Increment.
OP_Depth ()
OP_IF  {
        Depth <= i
        End loop;
OP_Else {                       // Run the expanded and tested function
        OP_TOALTSTACK           // j=0    //move stack for operation
        OP_TOALTSTACK           // j=1
        OP_TOALTSTACK           // j=2
        OP_TOALTSTACK           // j=3
        …
        OP_TOALTSTACK           // j=i-1
        OP_TOALTSTACK           // j=i
        OP_SUB                  // Subtract one to value on stack
        OP_FROMALTSTACK         // j=i
        OP_ FROMALTSTACK        // j=i-1
        …
        OP_X  // This is the operation to run. Insert here
        OP_Y  // This is the operation to run. Insert here
        …
        OP_ FROMALTSTACK        // j=2
        OP_ FROMALTSTACK        // j=1
        OP_ FROMALTSTACK        // j=0    // return stack order
}
```