Finite Machines

C. Merck

October 27, 2009

Abstract

An otherwise universal computer model running within a finite memory space is presented. The behavior of such a resource-limited machine is shown to be in principle distinct from that of the traditional Turing-machine with respect to the halting problem, Chaitin’s number, maximum output length, space complexity, and completeness.

1 Motivation

The notion of a universal computer, famously developed by Alan Turing and independently by Emil Post in the 1930s [1], has formed the basis for the canonical theory of computation. However, the universal computer is not physically realizable, due to the requirement of infinite space. Accordingly, present day physical computers, both those built by humans and those inherent in nature, are not universal but rather finite and therefore would be more accurately described by a finite theory than by the canonical theory of computation. Here a simple finite machine model is developed and then some of its properties are studied.

2 The Machine Model

We define a machine called $Q$ which is programmed in an imperative structured programming language - a variant of the $P'$ language developed by Böhm [2].

2.1 The Programming Language

The basic operation of the machine is this: it begins reading the given program at the beginning, with all memory initialized to zero. The machine runs, reading instructions, manipulating memory, and writing to the output as directed by the program until reaching the program’s end, resulting in a halt condition.

A $Q$ program is a finite-length string of symbols from an alphabet of seven instructions. Each machine cycle the instruction indicated by the program pointer is read and action is taken depending on the instruction and the value of the memory cell indicated by the memory pointer. The effects of the seven instructions are:

1The terms space, tape, and memory are used interchangeably.
The only restriction on input programs is that the loop instructions (l and u) be matched. The set of syntactically correct Q programs is entirely described by the following grammar:

1. **Base Axiom**: The following are Q programs: f, r, i, δ, w, and the empty string.

2. **Concatenation Rule**: Given any two Q programs ⃗p and ⃗q, ⃗p & ⃗q is a Q program.

3. **Loop Rule**: Given any Q program ⃗p, l & ⃗p & u is a Q program.

This grammar suggests a method of constructing syntactically correct programs, a challenge taken up in Section 2.4.

### 2.2 The Memory

The memory available to the machine is fixed during its operation, but may be chosen to be in various configurations. A particular machine, denoted by Q"n,m", has a fixed number, n, of memory cells (also known as *order*) each of which may be in any of m states (also known as *base*). More compactly, we say the memory is a vector in Z^n_m. Consequently, when n and m are finite (which is hereafter assumed unless written otherwise), increment and decrement operations on the memory pointer and on any memory cell are taken (mod n) and (mod m) respectively. For example, attempting to increment a memory cell already at the maximum value of m − 1 results in the cell's value becoming zero.

### 2.3 Q as a Recursive Function

Formally, Q is a function which maps program strings to output strings. When Q(⃗p) converges given a particular program string ⃗p, denoted Q(⃗p) ↓, the program is said to halt with output Q(⃗p). Otherwise the program does not halt, that is, it diverges, denoted Q(⃗p) ↑.

We can write down the exact behavior of Q in a single recurrence relation, but first the initial conditions for that recurrence must be established. We let

\[
Q^n_m(\vec{p}) = Q^n_m[0, \vec{r}_0, 0, \vec{s}_0](\vec{p}),
\]

where \(\vec{r}_0 = \vec{0} \in Z^n_m\) and \(\vec{s}_0\) is the empty vector².
Now we can define the recurrence relation describing the machine's operation:

\[ Q^n_m[i, r, j, s](p) = \begin{cases} 
  s', & \text{if } i = |p| \\
  Q^n_m[i + 1, r, j + 1 \pmod{n}, s'(p)], & \text{if } p^i = f \\
  Q^n_m[i + 1, r, j - 1 \pmod{n}, s'(p)], & \text{if } p^i = r \\
  \bigcup_{i=0}^{n-1} \begin{cases} 
    r_i + 1 \pmod{m}, & \text{if } i = j \\
    r_i - 1 \pmod{m}, & \text{otherwise} \\
  \end{cases} \bigcup_{i=0}^{n-1} \begin{cases} 
    j, s'(p), & \text{if } p^i = i \\
    j, s'(p), & \text{if } p^i = \delta \\
  \end{cases}, & \text{if } p^i = \text{ other} \\
  Q^n_m[i + 1, r, j, s, s'(p)], & \text{if } p^i = \omega \\
  Q^n_m[i + 1, r, j, s'(p)], & \text{if } p^i = I \\
  Q^n_m[i + 1, r, j, s'(p)], & \text{if } p^i = \text{ other} \\
  \bigcup_{d=0}^{d=0} \begin{cases} 
    i + 1, & \text{if } d = 0 \\
    \delta_{p}(i - 1, d - 1), & \text{if } p^i = I \\
    \delta_{p}(i - 1, d + 1), & \text{if } p^i = U \\
    \delta_{p}(i - 1, d), & \text{otherwise} \\
  \end{cases}, & \text{if } p^i = U \land r^j \neq 0 \\
\end{cases} \]

where \( \delta_{p}(i, d) \) is defined as

Note that if the input programs are ungrammatical, \( Q(p) \) may diverge because of the failure of a single evaluation of \( \delta_{p} \) to converge. For example, if a program \( p \) contains unmatched \( u \) instructions, then \( \delta_{p}(i, 1) \uparrow \) for some \( i \). This strongly suggests that \( Q(p) \uparrow \). However, in an effort to make \( Q \) independent of the behavior of \( \delta \) for programs with bad-syntax, we will from now on only discuss programs \( p \) which are in the set of \( Q \)-programs defined by the grammar in 2.1. The unfortunate side-effect of so restricting the domain of \( Q \) is a complication of the task of enumerating programs, discussed presently.

### 2.4 Enumerating Grammatical Programs

In order to tackle the problem of program-size complexity we must obtain a Gödel numbering for \( Q \)-programs. That is, we must find a one-to-one function \# : \( Q \)-programs \( \rightarrow \mathbb{N} \). Further, it is desired that this function always assigns larger numbers to longer programs. Two such definitions of \# are proposed here.

The obvious Gödel numbering method is to define a syntax checking function and use it to filter all possible program strings. Let the syntax checker function be defined as

\[ S(p) = \begin{cases} 
  1 & \text{if } p \in \text{Q-programs} \\
  0 & \text{otherwise} \\
\end{cases} \]

We cannot say with certainty that

\[ \forall \bar{p}, (\exists i \in \{0..|p| - 1\} \text{ with } p^i = u \text{ such that } \delta_{p}(i - 1, 1) \uparrow \Rightarrow (Q(p) \uparrow) \]

because of trivial counter-examples such as \( p = u \) or \( \text{fill } \text{null} \) where the last clause of the piecewise recurrence relation for \( Q \) is never evaluated.
Then let the Gödel numbering of all possible program strings grammatical and ungrammatical alike, the naïve numbering, be defined as

$$#_{all}(\vec{p}) = \sum_{i=0}^{\lfloor |\vec{p}| - 1 \rfloor} 7^i p^i.$$  

Now using the inverse of the above naïve Gödel numbering to give an ordered enumeration which is then filtered by the syntax checking function, we can define an acceptable numbering for grammatical programs:

$$\#(\vec{p}) = \begin{cases} \sum_{i=0}^{#_{all}(\vec{p})} S(#_{all}^{-1}(i)) & \text{if } S(\vec{p}) \text{ undefined otherwise} \end{cases}.$$  

A Gödel numbering could also be devised based on the grammar previously given. A set of known-grammatical programs can be iteratively augmented, and in that way enumerating all finite-length grammatical programs.

3 Halting

The busy beaver function $\Sigma_U(|\vec{p}|)$ denotes the largest output which a program of length $|\vec{p}|$ or less can produce on a computer $U$. On Turing-complete machines $\Sigma$ grows faster than any recursive function [3]. On finite machines however there exists a recursive upper-bound on the busy beaver function. We derive this bound via the following argument.

A finite machine with a finite program and finite memory (a finite machine-tuple) has a finite number of unique states which it can occupy. Furthermore, a machine is deterministic in that its next state can always be determined from its previous state. Lastly suppose there exists a special state, call it halted, marking the end of execution. We can now reason that if a machine were ever to reach the halted state, it must have passed through every other state either exactly once or not at all - for if a state were to be revisited, the program would loop indefinitely and never halt. So at the very most, the machine may pass through every one of its states once before halting. Since at most one symbol may be written to the output during each state transition (instruction), the output may contain no more symbols than there are states in the machine.

A machine $Q^n_m$ given program $\vec{p}$ has a memory pointer with $n$ possible states, a program pointer with $|\vec{p}|$ possible states, and $n$ memory cells which may each be in any one of $m$ states. In total there are $nm^n |\vec{p}|$ states. Thus

$$\Sigma_{Q^n_m}(|\vec{p}|) \leq nm^n |\vec{p}|.$$  

Note that it is not remarkable that individual values of $\Sigma$ are finite, only that we can write down an upper-bound which is recursive in program-size. What is remarkable is that the upper-bound given above is not only recursive but linear in program-size.

As a result of a computable upper-bound on program runtime, we can write an effective procedure $\text{HALT}(\vec{p})$ returning 1 if a program $\vec{p}$ halts and 0 otherwise, thus solving the halting problem for finite machines. Thus equipped, we may construct Chaitin’s number, the halting probability, as the convergent sum

$$\Omega = \sum_{n=0}^{\infty} 2^{-n} \text{HALT}(#^{-1}(n)).$$

\footnote{Compare with Nietzsche’s eternal recurrence argument.}
4 Functions

Thus far we have only discussed programs which take no input, being concerned with the complexity of the program’s output strings. However, the complexity of functions, mappings of inputs into outputs, is of greater interest. In this section, functions of the form \( f : \mathbb{Z}_m \rightarrow \mathbb{Z}_m \) and a manner of implementing them as programs on \( \mathbb{Q} \) is developed.

4.1 Implementing Functions on \( \mathbb{Q} \)

It is desired to implement functions of the form \( f : \mathbb{Z}_m \rightarrow \mathbb{Z}_m \) (hereafter called functions of base \( m \)) as \( \mathbb{Q} \)-programs. This is accomplished by appropriately padding programs with increment and output instructions so that the function’s argument is placed in the first memory cell and the memory cell pointed to when the program halts is taken to be the function’s output.

In terms of the definition given earlier for \( \mathbb{Q} \), we can define a higher-order function \( R \) such that 

\[
R^n_m[\vec{p}](a) = \begin{cases} 
Q^n_m(\vec{p} \& \vec{w})^0 & \text{if } a = 0 \\
R^n_m[i \& \vec{p}](a - 1) & \text{otherwise}
\end{cases}
\]

To ensure clarity one elementary example is given. Consider the base-3 successor function, call it \( f \). Naturally we have \( f(0) = 1 \), \( f(1) = 2 \), and \( f(2) = 0 \). The function \( f \) can be implemented by the program which is a single increment instruction, requiring but one cell of memory. This is written \( R^1_3[i] = f \), and is derived as follows:

\[
R^1_3[i](0) = Q^1_3(iw)^0 = (1)^0 = 1 = f(0), \\
R^1_3[i](1) = R^1_3[\vec{i}](0) = Q^1_3(iiw)^0 = (2)^0 = 2 = f(1), \\
R^1_3[i](2) = R^1_3[\vec{i}](1) = R^1_3[\vec{i}i](0) = Q^1_3(iiiw)^0 = (0)^0 = 0 = f(2),
\]

\[
\therefore R^1_3[i] = f.
\]

4.2 Sets of Implementable Functions

It is useful to consider the set of functions which are implementable as \( \mathbb{Q} \) programs in the above-mentioned manner. To denote the functions implementable on \( Q^n_m \), we write \( \Phi^n_m \). Or, in terms of the above-defined \( R \) notation, we let

\[
\Phi^n_m = \{ \text{base-}m \text{ function } f \mid \exists \vec{p} : R^n_m[\vec{p}] = f \}.
\]

Furthermore we adopt the following short-hand for unions of these sets:

\[
\Phi_m = \bigcup_{n=1}^{\infty} \Phi^n_m, \quad \Phi^n = \bigcup_{m=1}^{\infty} \Phi^n_m, \quad \Phi = \bigcup_{m=1}^{\infty} \bigcup_{n=1}^{\infty} \Phi^n_m.
\]

4.3 Space Complexity

The space complexity, or neatness, of a function is the number of memory cells required to compute that function. We can leverage the newly defined \( R \) notation in defining the neatness\(^5\) of a base-\( m \) function \( f \) as

\[
N(f) = \min \{ n \mid R^n_m[\vec{p}] = f \}.
\]

\(^5\)We say a function \( f \) is neater than a function \( g \) if \( N(f) < N(g) \).
function $f$ as

$$N(f) = \min\{n \mid f \in \Phi^n_m\}$$

This notion of neatness is simple but is not obviously computable. Suppose you claim that you have computed $N(f)$ and found it to be equal to 3. Most likely you claim this because you have found a program $\vec{p}$ such that $\mathcal{R}^3_m[\vec{p}] = f$. But then I ask: on what grounds do you say that there exists no program $\vec{q}$ such that $\mathcal{R}^2_m[\vec{q}] = f$? If you are right, then you cannot answer my question in finite time by exhaustively searching through programs, since you will never find such a program $\vec{q}$. Therefore, without some theoretical knowledge of an upper-bound on program-size, one cannot compute $N$ for cases where $N(f) > 1$. In an attempt to eliminate this problem, we define a more qualified version of neatness, called $H$-neatness. Let

$$N_H(f) = \min\{n \mid f \in \Phi^n_m \land |\vec{p}| = \min(|\vec{q}| \mid \exists n' : \mathcal{R}^{n'}_m[\vec{q}] = f)\}.$$  

4.4 Several Hypotheses involving $\mathcal{R}$

There is much to be learned about finite machines, especially with regard to space-complexity. Here follows several conjectures and associated proofs or counterexamples where they have been discovered.

4.4.1 Composition Lemma

$$\forall n : \forall f, g \in \Phi^n_m : f \circ g \in \Phi^n_m.$$  

In words, the set of functions implementable on $Q^n_m$ is closed under composition.

Outline of proof: After running an arbitrary program on $Q$ the memory and memory pointer are in an arbitrary state. However, if the program is appended with a so-called housekeeping routine \(^6\), the memory cells can be all initialized to zero with the exception of the cell at the memory pointer. If another arbitrary program is then appended, it will behave exactly as if it were run alone (without the first program and housekeeping routine). Thus, by placing the housekeeping routine between the implementation of two functions, the second function runs with the first program’s return value as its argument, which is the definition of function composition.

4.4.2 Finite Completeness Theorem

$$\forall f \text{ of base } m : f \in \Phi_m.$$  

In words, any (finite) function from $Z_m$ to $Z_m$ can be implemented as a $Q$-program in $\mathcal{R}$ notation.

Proof is by construction. Implementations of three elementary functions are proven. Then it is demonstrated that any function may be constructed through composition of the elementary functions.

\(^6\)The housekeeping routine of order $n$ is defined as

$$\vec{h} = (\&_{n-1}\text{fliu}) \& f.$$  

6
Assume an order, \( n \), of 3 and a base, \( m \), of 2 or greater\(^7\). Define the three elementary functions

\[
R(x) = x + 1,
\]

\[
S(x) = \begin{cases} 
0 & \text{if } x = 0 \\
1 & \text{if } x = 1 \\
x & \text{otherwise}
\end{cases}
\]

\[
G(x) = \begin{cases} 
1 & \text{if } x = 0 \\
1 & \text{if } x = 1 \\
x & \text{otherwise}
\end{cases}
\]

called rotate, swap, and group respectively. Furthermore, these functions are implementable as follows:

\[
R(x) = R^3_m[i],
\]

\[
S(x) = R^3_m[\text{swap}]
\]

\[
G(x) = R^3_m[\text{group}]
\]

\[\therefore R, S, G \in \Phi^3_m\]

Now, an algorithm for the construction of any base-\( m \) function \( f \) is presented, during which the reader should keep foremost in mind the list of values \( f(0), f(1), \ldots, f(m-1) \). Let \( \text{FREE}(x) \) denote the \( x^{th} \) number in the set \( \mathbb{Z}_m - f(\mathbb{Z}_m) \). Let \( \text{DUP}(x) \) indicate whether \( f(x) \) has already appeared in the list of values, so that \( \sum_{i=0}^{x-1} \text{DUP}(i) \) is the number of duplicate values in the range of \( f \) for arguments less than \( x \).\(^8\) Let \( \alpha \) be \( f \) but with duplicate values replaced with the next free value

\[
\alpha(x) = \begin{cases} 
\text{FREE}(\sum_{i=0}^{x-1} \text{DUP}(i)) & \text{if } \text{DUP}(x) = 1 \\
f(x) & \text{otherwise}
\end{cases}
\]

This function \( \alpha \) is a permutation of the base-\( m \) identity function and so may be obtained through repeated composition of the rotate and swap functions. Likewise, \( f \) may be obtained by repeatedly composing \( \alpha \) with \( R \), \( S \), and \( G \). Since \( R \), \( S \), and \( G \) are in \( \Phi^3_m \) and \( f \) is a composition of \( R \), \( S \), and \( G \), by the Composition Lemma we have that \( f \) is in \( \Phi^3_m \), Q.E.D.

4.4.3 A Bound on Neatness

\[
\exists \text{ a recursive upper-bound } B_N \text{ s.t. } \\
\forall f \text{ of base } m : N(f) < B_N(m).
\]

The bound on neatness is established using the Finite Completeness Theorem: Since all functions may be constructed from the elementary functions GS, ZS, and ZR, and the elementary functions have a neatness of at most 3, all functions have a neatness of at most 3.

\(^7\)The assumption of \( n = 3 \) can be weakened to \( n \geq 3 \) if a better swap function can be found which behaves correctly when \( n > 3 \).

\(^8\)A suitable, purely symbolic definition would be

\[
\text{DUP}(x) = \begin{cases} 
1 & \text{if } \exists y : y < x \land f(x) = f(y) \\
0 & \text{otherwise}
\end{cases}
\]

Note that this does not treat the first occurrence of a duplicated value as a duplicate.
4.4.4 Nonequivalence of Neatnesses

\[ \neg \forall f : N(f) = N_H(f). \]

**Proof by Example:** The shortest implementation of the function \( f(x) = 0 \) for any order is \( f \), valid when \( n \geq 2 \). So \( N_H(f) = 2 \). However, an implementation also exists when \( n = 1 \), specifically \( \text{liu} \). So \( N(f) = 1 \neq N_H(f) \), Q.E.D.

5 Other Machine Models

The above analysis was made with respect to a particular finite machine model. Other models of finite machines exist which may be subjected to similar analysis. Some other finite machines of particular interest are:

- A finite cellular automaton of Wolfram Class III, i.e. one which is computationally universal in the infinite case.

- A universal Turing-machine with its infinite tape replaced with a finite tape, perhaps "shaped" like a Möbius strip so that seeking beyond the last cell returns the tape-head to the first cell. The behavior of this setup is expected to be similar but more general than that of \( Q \) due to the isomorphism between \( Q \) given some program and a particular Turing-machine with a blank finite tape.

- A self-rewriting variant of \( Q \) where the program is loaded into and read from memory. In this case the problem of improper grammar becomes dynamic, as an initially grammatical program may modify itself to become ungrammatical.

I suspect that a study of these and other finite machine models would lead one to believe in a sort of finite-computational universality, i.e., that all reasonable finite machine models share the same overall properties with respect to the halting problem, completeness, and the various complexities.

6 Conclusions

The well-known principle of the recursive unsolvability of the halting problem, which holds true for the universal machines of canonical computational theory, is invalid when applied to finite machines. Consequences of the solvability of the finite machine halting problem include the computability of the busy beaver function and the halting probability for finite machines. Furthermore, for the machine and function-program isomorphism assumed here, all finite functions of a given base can be computed on a machine of that base within at most three cells of memory. This *finite completeness* stands in contrast to the famous incompleteness of infinite machines, i.e. the existence of uncomputable functions.
Symbol List

$\mathbb{Q}$ the machine model used here
$\mathbb{Z}$ the set of integers
$\mathbb{Z}_m$ the set of integers (mod $m$)
$\Sigma$ the busy beaver function
$\wedge$ boolean AND
$\uparrow$ diverges
$\downarrow$ converges
$\vec{x}$ a vector (sometimes called a string)
$x^i$ the $i^{th}$ component of $\vec{x}$, if defined
$\&$ vector append operator
$\in$ is a member of
$\circ$ function composition
$f^n$ function composed with itself $n$ times
$\mathcal{R}$ interpretation of a program as a finite function

References


---

$^9$Two vectors or a vector and a scalar may be appended by this operator. Also, the following notations are defined:

$$\&_{i=0}^n \vec{f}(i) = \vec{f}(0) \& \vec{f}(1) \& ... \& \vec{f}(n),$$

$$\&_{n} \vec{x} = \vec{x} \& \vec{x} \& ... \& \vec{x} \ (n \ times).$$