AN ABSTRACT OF THE THESIS OF

Lisa Jo Wagner for the Master of Science

in Mathematics presented on May 9, 1989

Title: THE HALTING PROBLEM

Abstract Approved: [illegible]

The primary purpose of this thesis is to show the unsolvability of the Halting problem. To do this, an introduction of the Turing machine and some basic examples using the Turing machine are given. Primitive recursion, Godel numbering, mu-recursion and recursive enumerability are discussed in relation to Church's thesis and solvability using a Turing machine. Using this information, the solvability of the Halting problem is discussed. Finally, an example of an uncomputable function, Rado's function, (the Busy Beaver problem) is presented

THE HALTING PROBLEM

A Thesis

Presented to

The Division of Mathematical and Physical Sciences

EMPORIA STATE UNIVERSITY

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Lisa Jo Wagner

May, 1989

Thesis
1459
[illegible]

James W. Calvert
Approved for the Major Division

Charles Greenleaf
Committee Member

[illegible] Scott
Committee Member

[illegible]
Committee Member

James G. Anderson
Committee Chairman

James L. Wolfe
Approved for the Graduate Council

ACKNOWLEDGEMENTS

The completion of this thesis would not have been possible without the contributions of some very important people in my life.

I want to thank Dr. James Anderson and Dr. William Simpson for their patience and understanding throughout the writing of this paper. I would also like to thank the entire faculty of Emporia State University for the quality of education I feel that I have received in the past six years.

I also want to thank my friends for putting up with me throughout this last semester.

Finally, I want to express my gratitude to my famnily for their everlasting encouragement, and their continued love and support throughout. I have a great deal to be thankful for, and the accomplishments I have made reflect the ideals instilled in me by my parents.

CONTENTS

Chapter One

Background for Theory of Computing

Introduction

Characteristically, computer science is thought of as an applied field. Research continues for applications based on mathematical models of theories from a variety of fields; including biology, psychology, physics, and chemistry. Since the applications of computer science are so widespread, the limitations of the computer, and the knowledge of the type of questions one can ask while realistically expecting an answer, becomes of interest.

There are several questions one might ask in beginning to explore computer theory. How did the theory of computer science develop? How did the formalisms of computer theory evolve? Is there a "standard" computer that can solve any problems which might be presented? If not, what types of problems are known to be solvable by this "standard" computer? Is there a method for determining if a problem can be solved by a computer?

This thesis will contain some background information leading to the development of the theory of computing. The "standard computer" (Universal Turing Machine) will also be considered in an attempt to answer such questions as mentioned above. Various types of functions and algorithms (defined on p.21), and the ability to recognize these will be introduced as a method for determining the type of questions for which the machine can find an answer. These and other ideas to be presented will lead us to the problem of knowing when, or if, the machine will actually reach a conclusion. This problem is also known as the decision problem or halting problem.

Historical Background

Before delving into the theory of computation, the historical development of this theory will be outlined. The theory of mathematical logic plays an important role in computer theory. As the twentieth century began, Georg Cantor's (1845-1918) introduction of Set Theory (unions, intersections, inclusion, cardinality, etc) presented some unusual findings in mathematics that needed to be resolved. Cantor uncovered in his work concepts which appeared to be contradictions in what had previously been considered "rigorously proven mathematical theorems". [2,4] Some of his unusual findings were accepted (such as that infinity comes in different sizes), although others were not readily accepted by his colleagues (such as that some set is bigger than the universal set).

These and other questions aroused the natural inquisitiveness of David Hilbert (1862-1943) who believed that all of mathematics could be derived on the same solid conditions as Euclidean Geometry. Specifically, Hilbert believed all of mathematics could be characterized by precise definitions, definitive axioms, and rigorous proofs. The mathematics that had developed over the centuries since Euclid (300 B.C.) did not meet these standards of precision. Hilbert implicitely believed that if mathematics were put back into a form such as the Euclidean standard, the problems Cantor's work had presented would simply go away. His two major projects were to create this new system and establish that it was free of paradoxes; and to find a method which would guarantee the ability to construct proofs of all true statements in mathematics. Hilbert wanted an approach that was clear-cut and without any reliance on mathematical insight. He wanted a strictly defined set of rules which, if followed, would eventually yield the answer. Hilbert thought it possible to develop methods for solving mathematical problems; perhaps even <u>a</u> specific method which could solve <u>all</u> mathematical problems in some finite number of steps. Before looking for such a method, the exact notion of what is or what is not a mathematical statement had to be developed. There was also the problem of defining precisely what can and what cannot be a step in such a method.

Mathematical logicians, while trying to follow the suggestions of Hilbert and straighten out the predicaments left by Cantor, found they were able to mathematically prove some of the desired methods for solving a particular problem <u>cannot</u> exist. One of these logicians, Kurt Godel (1906-1978), showed that there was no single algorithm which would guarantee the ability to provide proofs for all the true statements in mathematics. He also proved that there exist true statements that could not be proven to be correct. In his **Incompleteness Theorem**(1931), Godel shows that for any consistent formal system (axioms plus rules of inference) whose axioms adequately define addition and multiplication of natural numbers, there are propositions which are true of the system, but which are not provable within the system, that is, which are not derivable from the axioms using rules of inference.[1, 5] Godel's proof shows that a specific mathematical system either contains some true statements without any possible proof, or else false statements which can be proven true.

This result caused mathematicians and logicians, who were still attempting to fulfill Hilbert's program, to wonder about the methods they were trying to find,what the fundamental composition of some method was, and if there was a way of finding proof-generating procedures for those true statements in mathematics which do have proofs. Using Church's definition of this method (termed an algorithm), Alonzo Church, Stephen Cole Kleene and also, independently, Emil Post were able to prove that there were problems which no algorithm could solve. While investigating algorithms, Church made a proposal,now known as **Church's Thesis**, about algorithms in general and the ability to determine if a problem is or is not solvable based on the type of algorithm. Church's Thesis may be stated in a variety of ways and it is useful to see the statements which will be referred to in this paper.

Church's Thesis may be stated as any of the following equivalently:

(1) a set is decidable if and only if the set is recursive;

(2) a function over the nonnegative integers is computable if and only if it is a mu-recursive function;

(3) if a set is not recursive, it is not decidable;

(4) there is an effective procedure to solve a decision problem if and only if there is a Turing machine accepting a recursive language[set] which solves the problem;[7, 257]

(5) a decision problem is partially solvable if and only if there is a Turing machine that accepts precisely the elements of the decision problem whose answer is yes.[7, 257]

While also solving this problem independently, Alan Mathison Turing (1912-1954) developed the concept of a theoretical "universal-algorithm machine". In his study of what was or was not possible for such a machine to do, Turing found some tasks were impossible to solve using this abstract machine. Turing's model for this universal algorithm machine played an important role in the construction of the first computer, which was based on his work in abstract logic. In order to explore the theories behind the operating computer, the main thrust of the rest of this paper will be concerned with Turing's "universal algorithm machine", the Turing machine, and this machine's ability to solve certain types of problems or answer certain types of questions.

Chapter Two

The Turing Machine

Description, Definition of a Turing Machine

A.M. Turing's formalization of his "universal algorithm machine" which will from now on be referred to as a Turing machine, remains a standard for computer theory today. In its simplest form, a Turing machine consists of a finite-state control unit, a tape, and a read/write head. The control unit operates in descrete steps; at each step it performs two tasks depending on its current state and the tape symbol currently scanned by the read/write head. In this machine, the tape has a left end but extends indefinitely to the right,and is divided into a sequence of 'cells'. Each cell contains either one character or a blank. The input word is presented to the machine one letter per cell beginning in the left-most cell. The rest of the tape is initially filled with blanks, which will be denoted as #. The read/write head may scan any cell along the tape, after moving there one cell at a time, and consequently change the symbols in the cells of the original input string as well as write symbols on the unlimited blank portion of the tape to the right. Since the machine can move its head only one square at a time, after any finite computation only finitely many tape squares will have been visited. The tape head can never move left off the end of the tape; if the machine attempts or is given orders to do so, it ceases to operate (also known as "hanging"). At this point, the Turing machine is a function of its states and some input string. A program, which is the set of rules that guide the Turing machine (or function) on the basis of the letter the tape head has just read, tells the Turing machine how to change states, what to print and where to move the tape head.

At the beginning of any computation, the tape is either initiated with blanks or has the input string written on it. If the input word is written on the tape, it is written beginning with a blank in the leftmost cell, followed by the required string, with another blank at the end of the string. The read/write head is initially scanning the blank just to the right of the input string.

Because a Turing machine can write on its tape, it can leave an answer on the tape at the end of a computation. There is a specific state, known as the halt state, which is used to signal the end of a computation. Since for all Turing machines the halt state will be the same, it shall be denoted by h and this symbol will not be used for any other purpose. To terminate execution of a program successfully on a Turing machine, the input and results must lead to a halt state, h. The input word is then said to be accepted by the Turing machine. L and R will denote movement of the tape head to the left or to the right. It is important to note in the following definition L and R are not contained in the input alphabet of the Turing machine.

The formal definition of a standard Turing machine can now be presented.

Definition 2.1:

A Turing Machine, TM, is a quintuple TM = $(\Sigma, Q, q_0, \delta, \Gamma)$, where

Q is a finite set of states, not containing the halt state, h;
Σ is an alphabet, containing the blank symbol #, but not containing L and R;
$q_0 \in Q$ is the initial state;
δ is a function from $K \times \Sigma$ to $(K \cup \{h\}) \times (\Sigma \cup \{L, R\})$;
Γ is an alphabet of characters that can be printed (which may include Σ).[6,170]

Notice that the Turing machine is defined as a function and will be used as a method of representing functions (see Example 2.24). This machine will be used as a check to determine which type of functions can be formalized in such a way that these functions can be calculated.

Examples of Basic Turing Machines

There are several types of basic Turing machines which may be shown at this point. These machines can be used within other Turing machines to simplify many aspects of what could be a difficult computation. These include Turing machines such as a shift right/shift left machine (which takes the given input and shifts it one space to the right/left); a copy machine (which copies a given input directly to the right of the original); and also addition, subtraction and multiplication machines. (See the figures which follow.)

The following symbols will be used in the examples of Turing machines which follow and have teh designated meaning.

$>$	this indicates where the machine begins reading the diagram for the 'program'.
$L_\#$	the read/write head moves left until it scans a blank symbol
$R_\#$	the read/write head moves right until it scans a blank symbol
L_a .... R_a	the read/write head moves left/right until it scans an a
L_i^2 .... R_i^2	the read/write head moves to the left/right until it reaches the second i
σ.......	the read/write head will write a σ
$\xrightarrow{\sigma}$...	the arrow designates the direction and rules the machine will follow according to the symbol scanned

EXAMPLE: Shift Right Machine $\mathfrak{S}_R$

$$\begin{tikzcd}
>L \arrow[r, "\sigma \neq \#"] \arrow[d, "\#"] & R\,\sigma\,L\,\# \arrow[l, bend right] \\
R_{\#}\ h &
\end{tikzcd}$$

Figure 2.11

EXAMPLE: Shift Left Machine $\mathfrak{S}_L$

$$\begin{tikzcd}
>L_{\#} \arrow[r] & R \arrow[r, "\sigma \neq \#"] \arrow[d, "\#"] & L\,\sigma\,R \\
& L\,\#\,h &
\end{tikzcd}$$

Figure 2.12

EXAMPLE: Copy Machine $\mathfrak{C}$

$$\begin{tikzcd}
>L_{\#} \arrow[r] & R \arrow[r, "\sigma \neq \#"] \arrow[d, "\#"] & \#\ R^2_{\#}\,\sigma\ L^2_{\#}\,\sigma \arrow[l, bend right] \\
& R\,\#\,h &
\end{tikzcd}$$

Figure 2.13

EXAMPLE: Addition Machine

Given two input values, say $x_1 = 2$ and $x_2 = 3$,
(i.e., #11# and #111#)
the following machine will find the sum of the two inputs and leave that sum on the tape as an answer.

$> S_L \; L_\# \; 1 \; R_\# \; h$

Figure 2.14

This machine begins in this position

```
# 1 1 # 1 1 1 # ...
              ↑
```

and immediately uses the shift left machine to transform the input to the following

```
# 1 # 1 1 1 # ...
            ↑
```

The read/write head then moves one square at a time to the left until it scans a #.

```
# 1 # 1 1 1 # ...
    ↑
```

The head then writes a 1 and moves right until it scans a #.

```
# 1 1 1 1 1 # ...
            ↑
```

The computation is now complete.

EXAMPLE: Integral Subtraction Machine

Define integral subtraction to be the following:

$$x \dot{-} y = \begin{cases} x - y & \text{if } x > y \\ 0 & \text{otherwise} \end{cases}$$

The Turing machine which performs this operation takes an input of two numbers, x and y , and 'erases' from the first input, the second. If the second is larger, the tape will be left with nothing but blanks when the computation is complete.

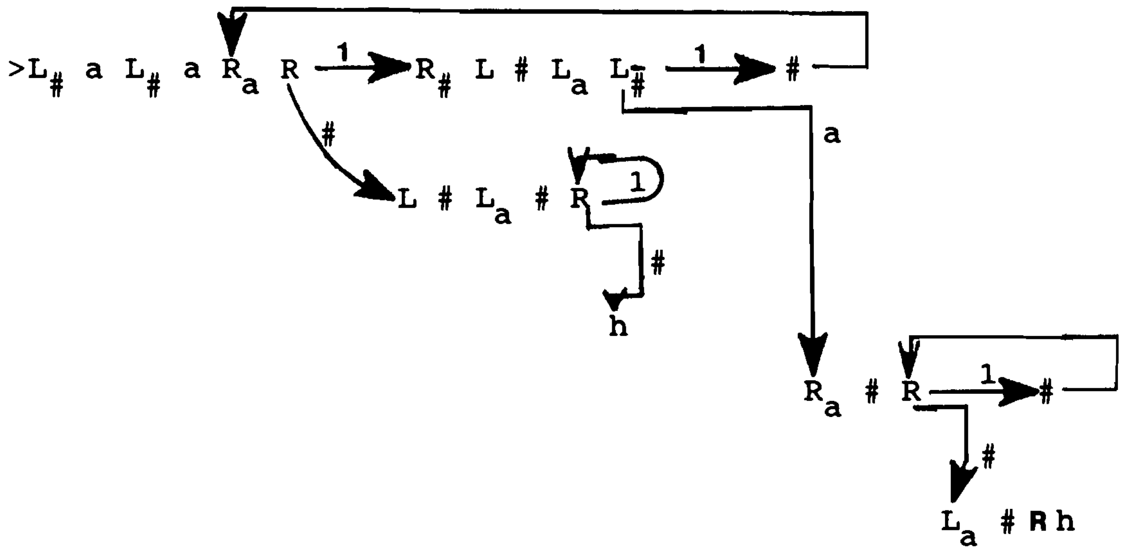


Figure 2.15

<u>EXAMPLE</u>: Multiplication Machine

$$\begin{tikzcd}
>a\ L_{\#}\ L \arrow[r, "1"] \arrow[d, "\#"] & \#\ L \arrow[r, "1"] \arrow[rrd, "\#"] & a\ R_a\ \#\ \mathfrak{C}\ a\ L_a\ \# \arrow[l, bend right] & & \\
R\ a \arrow[r] & R \arrow[r, "1"] \arrow[d, "a"] & \# \arrow[l, bend right] & & \\
& \#L_a\#\ h & & & \\
& & a\ R_a\ \# \arrow[r] & L_{\overline{1}} \arrow[r, "\#"] \arrow[lld, "\#"] \arrow[d, "a"] & 1\overset{\#}{R}_{\#}L\# \arrow[l, bend right] \\
& L_a\ \#\ R_{\#}\ h & & \#\ R_a\ h &
\end{tikzcd}$$

Figure 2.16

A Turing machine may be used as an **acceptor**. This machine will reach the halt state h, if and only if the input string is contained in the set of strings produced by the defined function. If the input string is not accepted by the given Turing machine, the machine never halts; it will loop infinitely. An example of this type of machine follows.

<u>EXAMPLE</u>: Accepting Machine

The following machine accepts the strings over the alphabet $\Sigma = \{a, b\}$ which start with a and loops infinitely on ~~all~~ strings which so not.

(i)

$$\begin{tikzcd}
>L_{\#} & R \arrow[r, "a"] \arrow[d, "\bar{a}"] & R_{\#} \; h \\
& \# &
\end{tikzcd}$$

(ii)

# a b a a \# ...

$\Uparrow$ (under the second \#)

Figure 2.17

The read/write head begins in the above position (ii). According to (i), the head moves left one square at a time until it reaches a blank and then moves one square to the right.

(iii)

# a b a a \# ...

$\uparrow$

The head either reads an 'a' or some other symbol in Σ. If an 'a' is in the square, (i) requires that the head move right one square at a time until it reaches a blank and then halts. The Turing machine has accepted the string.

(iv)

# b a \# ...

$\uparrow$

(v)

# \# a \# ...

$\uparrow$

Otherwise(iv), the read/write head writes a blank in the square and loops infinitely writing a blank in this square repeatedly. The Turing machine does not accept this input string.

A Turing machine may also **emulate a function.** Let Σ_1 and Σ_2 be alphabets not containing #. Let f be a function from Σ to Σ . A Turing machine TM is said to compute f if $\Sigma_1, \Sigma_2 \subset \Sigma$ and for $w \in \Sigma_1$ if f(w) = u then the machine TM begins in the start state q_0, reads w, and ends in the halt state, h, with u on the tape. An example of this follows.

<u>EXAMPLE</u>:
Consider the successor function, S(n), where

$$\begin{cases} S(0) = 1 \\ S(n) = n + 1 \qquad \text{for all } n \in N. \end{cases}$$

This function, S, can be emulated by the following TM.

Given input n, such as for n = 3 # 1 1 1 # ...

S(n) is computed by

$$\boxed{> 1\ R\ \#\ h}$$

Figure 2.18

Another function, $f(w) = ww^r$ where w is some arbitrary string and w^r is the palindrome for this string, is computed as follows.

$$\begin{tikzcd} >L \arrow[r, "\sigma \neq \#"] \arrow[d, "\#"] & \# R_\# \sigma L_\# \sigma \arrow[l, bend right] \\ R_\# h & \end{tikzcd}$$

(for example, f(#ab#) = #abba#)

Figure 2.19

A generating machine is a Turing machine which initially begins operation with a blank tape and after some operation prints a string from the set defined by the some function. The machine continues in this manner. The order in which the strings occur does not matter and any string may be repeated. This method of generating a set is also known as enumerating a set. An example of a generating machine follows.

<u>EXAMPLE</u>: Generating Machine

The following Turing machine generates string of the form $\{a^i b^i : i >= 0\}$ given the input i. (An example of i would be for i=3 i would be represented on the tape in the following manner:

# 1 1 1 # ...)

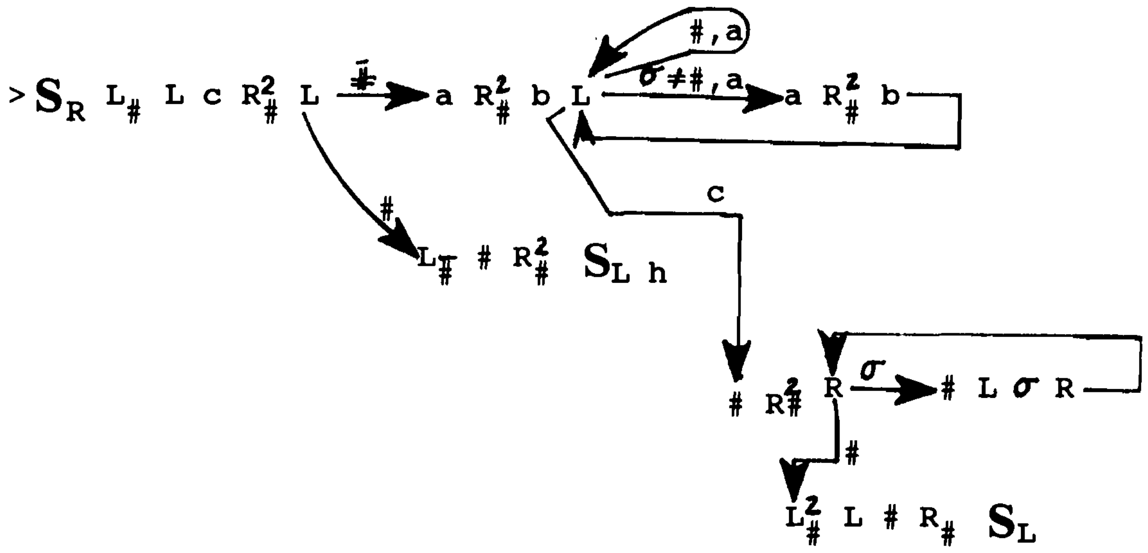


Figure 2.20

A **decision machine** is a Turing machine which given any input string, eventually erases the input string and writes "YES" or "NO" on the tape depending on whether or not the input would be accepted by a machine defined on the same function. In other words, the machine answers the question of whether or not the input string is in the set of strings produced by a given function. An example of a decision machine follows.

<u>EXAMPLE</u>: Decision Machine

The following Turing machine decides whether or not the given input is of even length. If the input is of even length, the TM writes a (Y) on the tape. If the input is odd, the TM writes a (N) on the tape.

$$\begin{tikzcd}
{>}\,L \arrow[r, "a"] \arrow[d, "\#"] & L \arrow[l, bend right, "a"'] \arrow[r, "\#"] & R\ (N)\ R\ \#\ h \\
R\ (Y)\ R\ \#\ h & &
\end{tikzcd}$$

Figure 2.21

This TM begins with the read/write head scanning the squares and checking for multiples of two a's. When it reaches a blank at an odd interval, the head is directed to answer (N). If the input is even, the head will write (Y) as an answer.

The similarity within these definitions yields a result which will be useful in chapter three but should be noted at this time. Given a set of strings which are accepted by some Turing machine, there exists another Turing machine which will generate the same set of strings. The converse is also true: given a Turing machine which will generate a set of strings, there is another Turing machine which will accept this same set of strings.(for constructive proof see [2,797-8]) This implies that any set of strings which is accepted by a Turing machine is a set which is enumerable, and any set of strings that is enumerable is a set which will be accepted by some Turing machine. Therefore, if it can be determined that a set is enumerable, then the set is also acceptable by some machine and the Turing machine will reach a halt state when computing on this set.

The previous examples illustrate that Turing machines can perform powerful computations. How powerful the Turing machine actually is will be treated later in the development of various types of functions which are acceptable or decidable by a Turing machine.

Extensions of the Turing Machine

This simple machine's ability to perform such powerful operations as those previously shown warrants consideration of the effects of extending the Turing machine in various ways. For example, the addition of extra tapes, the addition of extra heads on those tapes, allowing the tape to be infinite in both directions instead of to the right only, or considering a multi-dimensional tape are all extensions of a standard Turing machine. It can be shown that in any case, the operation of any extended machine does not add to the capabilities of the standard Turing machine. Any extended machine can be imitated by a standard Turing machine. [6, 192-204] [7, 221-227] This is an advantage as these results allow use of the additional features or not according to which would be more beneficial in finding a solution to the type of problem being solved.

The knowledge of how a Turing machine operates, and that any computing machine is equivalent to the standard Turing machine, leads to the question of which types of problems a Turing machine can be used to emulate and which types will lead to a solution on the machine. Exploration into this question, as well as whether or not a Turing machine is capable of emulating every formalized procedure, is the subject of the next chapter.

Chapter Three

Primitive Recursion, Godel Numbering, and Mu-Recursion

Algorithms

The following describes a formal procedure which a Turing machine can represent. This procedure, called an algorithm, will be used as a method of specifying functions to be emulated by the Turing machine.

Definition 3.1:

> "An effective procedure is a finite, unambiguous description of a finite set of operations. The operations must be effective in the sense that there is a strictly mechanical procedure for completing them.
> An effective procedure which specifies a sequence of operations which always halts is called an algorithm. [Any] program which always halts for any input is an algorithm. The point at which an algorithm will halt is not necessarily calculable in advance." [1, 2]

The following are implications of the conditions of this definition. 1)Only a limited amount of information is added as a result of each step in the sequence of operations. 2)The fact that the input is finite implies that after any finite number of steps there is only a finite amount of information (in particular, upon termination). An output of infinite size is therefore impossible. 3)Once the algorithm, input values, and mode of implementation have been chosen, there are no other choices relevant to execution available. (This is the condition of determinism).

One classic example of an algorithm is the Euclidean algorithm for finding the greatest common divisor of two positive integers. This algorithm has components which typify many algorithms.

EXAMPLE: Euclidean Algorithm; Remainder Version.

Compute integers $x_1, x_2, \ldots,$ as follows. x_1 and x_2 are, respectively, the first and second inputs. For each i >= 2, when the sequence has been computed as far as x_i:

i) If $x_i = 0$, halt and take x_{i-1} as output.

ii) Otherwise, if $x_{i-1} < x_i$,

let $x_{i+1} = x_{i-1}$.

iii) Let $x_{i+1} = \text{Remainder } (x_{i-1}/x_i)$

For example, consider

$x_1 = 352$ and $x_2 = 154$.

Then $x_3 = 352 - 308 = 44$

$x_4 = 154 - 132 = 22$

$x_5 = 0$

The output would be $x_4 = 22$.

Since the definition states that any execution of an algorithm terminates, this implies that the question asked of the algorithm is always answered. The difficulty in verifying that an specific algorithm satisfies this condition is also known as the **Halting Problem.**

Definition 3.2:

> The halting problem for Turing machines is the problem of determining, given an arbitrary Turing machine with an arbitrary input string, whether or not the Turing machine will eventually reach a halt state.[6, 277]

However, in order to determine if a specific algorithm <u>will</u> terminate, the exploration into the classes of problems for which an algorithm exists, must continue.

<u>Recursion</u>

One of the methods which readily conforms to the concept of an algorithm is that of defining sets or functions using a recursive definition. A recursive definition must require:

1) specification of the beginning cases;
2) rules for construction of more objects in the set from the beginning cases, and;
3) the self-referential sense that no objects except those constructed in this way are allowed in the set.

Definition 3.3:

Given two n-argument functions, g and h, another n-argument function f may be obtained recursively in the following manner:

$$f(x_1, \ldots, x_{n-1}, 0) = g(x_1, \ldots, x_{n-1})$$

$$f(x_1, \ldots, x_{n-1}, S(x_n)) = h(x_1,\ldots,x_n,f(x_1,\ldots,x_n)).$$

Recall the successor function, S(x), defined in the example on page 16. The following functions are defined using the successor function and a recursive definition:

addition ========> $\begin{cases} a + 0 = a \\ a + S(b) = S(a + b); \end{cases}$

multiplication ==> $\begin{cases} a * 0 = 0 \\ a * S(b) = a*b + a \end{cases}$

and exponential => $\begin{cases} a^0 = 1 \\ a^{S(b)} = a^b * a. \end{cases}$

Definition 3.4:

A primitive-recursive function is a mapping of k-tuples of natural numbers(denoted $\mathbb{N}$) into natural numbers using composition or recursion on the initial functions. The initial functions are as follows:

i)the successor function, $s:N \rightarrow N$, where $\begin{cases} s(0) = 1; \\ s(n) = n + 1, \quad \text{for all } n \in N. \end{cases}$

ii) the k-place projection function, $\xi_i^k : N^k \longrightarrow N$, where $k >= 1$, $1 <= i <= k$, and $\xi_i^k (n_1, \ldots, n_k) = n_i$ for all $n \in N$.

iii) the zero function, $\zeta : N^0 \longrightarrow N$), where $\zeta(\ ) = 0$.[6, 232-3]

Multiplication can be shown as an example of a primitive recursive function.

<u>EXAMPLE</u>:

The product of m and n, *(m, n) is as follows:

$*(m, 0) = 0$

$*(m, (s(n)) = +(m, *(m, n))$

A function is primitive recursive if and only if its characteristic function is primitive recursive.

Definition 3.5:

A function, C, is a characteristic function if and only if it completely describes another function, f, in the following manner:

$$C(x) = \begin{cases} 0 & \text{if } x \quad \text{range } f \\ 1 & \text{otherwise.} \end{cases}$$

The definition of primitive recursion specifies a method for describing all elements of the function and can be considered an algorithmic process. The Turing machine is capable of computing the values of a function using the successor function as was shown on page 16, but the question is if it is capable of computing all functions which can be generated by using composition or primitive recursion on the initial functions.

Since the Turing machine is encoded with strings (numerical or alphabetical), and primitive recursion is defined over a formal numbering system, it is of interest to adopt a method for representing strings over an alphabet. This would enable a correlation between the strings and the natural numbers which in turn would be a form for representing the strings as primitive recursive functions. This method is known as **Gödel numbering.**

Gödel showed that all primitive recursive functions are expressible in the formal theory of numbers. His technique to show this consisted of assigning numbers to logical formulas and proofs. He could then define as a primitive recursive function, some function B,

where

$$B(x,y) = \begin{cases} 0 & \text{if } x \text{ is the Gödel number of a proof of the formula with Gödel number } y, \\ 1 & \text{otherwise.} \end{cases}$$

Gödel numbers, then, can always be assigned to a computational system so that, regardless of subject matter, the procedure can be computed as a primitive recursive function over the nonnegative integers. The uniqueness of the Gödel numbering system relies on a fundamental theorem of arithmetic: a positive integer can be decomposed in only one way, as a product of primes.

It is possible to use Gödel numbering to determine whether every computable function can be written as a primitive recursive function. The definitions of primitive recursive functions can be listed sequentially. In other words, choose a Gödel numbering system to represent the initial functions and the equations which define functions in terms of other functions by composition and primitive recursion. List all the strings over this set eliminating those which do not represent the definitions of primitive recursive functions. The infinite list which is left, say A_1, A_2, A_3,..., has each member representing the definition of some primitive recursive function. Every definition of a primitive recursive function appears somewhere on the list. If f_i is the primitive recursive function defined by A_i, consider the 1-place function f to be evaluated on an argument n by finding the n^{th} string defining a primitive recursive function. Then apply that definition to as many arguments as f_n requires and add 1. This algorithm will evaluate f, so f must be computable. But f cannot be primitive recursive. If it were, then f must be f_i for some i; that is, some A_i represents a definition of f. But when evaluated with argument i, f and f_i will differ in value by 1, and so f cannot be f_i.

Therefore, a computable function can be found that is not in the list of primitive recursive functions. Some extension must be made to the method of defining functions in order to obtain all computable functions.

One example of a function which has been shown to be computable and yet not in the class of primitive recursive functions is Ackermann's Function. For a discussion of this function, see [1, 252-9], [7, 104-9].

An example of Gödel numbering follows.

Let $I = \{i_1, i_2, \ldots, i_n\}$ be a sequence of positive integers.

Let $P = \{2, 3, 5, \ldots, p_n\}$ where p_n is the nth prime number.

Then the Gödel number of I is denoted as:

$$G(i_1, i_2, \ldots, i_n) = \prod_{k=1}^{n} (p_k)^{i_k} .$$

For example, consider the sequence

$$\{2, 1, 3, 1\}.$$

This sequence has Gödel number

$$2^2 * 3^1 * 5^3 * 7^1 = 10,500.$$

As, for longer sequences, a Gödel number will quickly become very large, the computation of these numbers is not as relevent as the ability to assign such a number. If a

Gödel number can be assigned to any positive integer value, then similarly they can be assigned to any numerical value by considering it a sequence of positive integers. Since there are many integers which will not be the Gödel number of a sequence of numbers, then any mathematical symbols, and any written text can be assigned a unique Gödel number. (G is a Gödel number if and only if the following is true: for all m and n, if m < n and the prime factorization of n divides G, then the prime factorization of m divides G.[7,150])

One method of assigning a Gödel number to written text is as follows: blank, 1; small a through z, 2 through 27; capital A through Z, 28 through 53, comma, 55; and so on. Then the Gödel number of any text is simply the Gödel number of the corresponding sequence of integers. Consider the phrase:

So be it.

The sequence for this phrase is

{46, 16, 1, 3, 6, 1, 10, 21, 54}

Its Gödel number is

$$2^{46}*3^{16}*5^{1}*7^{3}*11^{6}*13^{1}*17^{10}*19^{21}*23^{54}$$
$$\approx 5.88 * 10^{144}.$$

In a similar way one can assign Gödel numbers to strings over any alphabet or to any set of symbols.

Mu-recursive Functions

As an extension of the primitive recursive functions, some method of extending the class must be described other than by composition or primitive recursion itself. The method to be employed is that of unbounded minimalization.

Definition 3.6:

First, define $m = \min(A_x)$ (the smallest member of A_x with

$A_x = \{y : f(x,y) = 0$ and $f(x,y')$ is defined for $y' < y \})$

If f is a (k + 1)-place function for $k >= 0$, the unbounded minimalization of f is that k-place function g such that for any $n \in N^k$,

$$g(n) = \begin{cases} \text{the least } m \text{ such that } f(n,m) = 0, & \text{if such an } m \text{ exists,} \\ 0 & \text{otherwise.} \end{cases}$$

[6,249]

In general, the unbounded minimalization of a primitive recursive function need not be primitive recursive. There is no method for determining whether an m of the required type exists. To avoid this problem, a restriction will be placed on the application of unbounded minimalization so that it is to regular functions only.

Definition 3.7:

A regular function is a (k + 1)-place function such that for every $n \in N^k$, there is an m such that $f(n,m) = 0$.

Finally, the extension of the primitive recursive functions, the **mu-recursive functions** can be defined.

Definition 3.8:

A function is <u>mu-recursive</u> if and only if it can be obtained from the initial functions by
composition,
primitive recursion, or
unbounded minimalization to regular functions

(Recall the initial functions are
1.the zero function,
2.the projection function, and
3.the successor function,
which are defined for nonnegative integers only.)[6, 249]

The class of primitive recursive functions then, by definition, is contained in the class of mu-recursive functions. Recall Church's thesis in the form stating that a function over the nonnegative integers is computable if and only if it is a mu-recursive function. S.C. Kleene verified that every mu-recursive function is Turing-computable and vice-versa.(Kleene cited in [6,271]) This gives a basis for determining solvability for the Turing machine and a great many functions. Then the class of primitive recursive functions, as well as the class of mu-recursive functions, are computable.

Recursive Enumerability

From chapter two, the four types of Turing machines discussed were: the accepting machine, the generating machine, the decision machine, and the emulating machine. It was stated that any set of strings which is accepted by a Turing machine is a set which is enumerable and any set of strings that is enumerable will be accepted by some Turing machine. This property is known as recursive enumerability.

Definition 3.9:

A set S is recursively enumerable if either it is the empty set or else there is an algorithmically computable function, f, mapping the set of positive integers onto the set S, that is,

$$S = \{f(i): i \text{ is a positive integer}\}. [7, 10]$$

The recursively enumerable sets or functions have certain characteristics which aid in the ability to determine if and when a function is recursive. If a set is recursively enumerable, and the set's compliment is recursively enumerable, then the set is recursive. Thus, if a set is recursive, then it is recursively enumerable.[7,11] However, there exist recursively enumerable sets which are not recursive; that is, recalling the generating and the decision machines from chapter two, there is no Turing machine which can decide if an object will be in the recursively enumerable set.[6, 276]

A characteristic of the recursively enumerable sets that affects its membership in the class of recursive functions is that some of the methods for computing them may never be completed. Not all recursively enumerable sets are recursive, but if the restriction is placed on the recursively enumerable sets that it must be finite, then it will be recursive.[1, 142]

The problem of deciding which functions a Turing machine can solve, and if there is a method for determining if a function is of the correct type, present another question. What is an example of a problem a Turing machine cannot solve? Are mu-recursion and recursive enumerability the only requirements for solvability? The next chapter will present an <u>undecidable</u> problem for Turing machines and a method for determining if a problem is equivalent to this known undecidable problem.

Chapter Four

The Halting Problem

Decidability

The problem of calculating the characteristic function of a particular set is called the decision problem for that set. Another way of stating this is, the decision problem for a particular set is the problem of effectively establishing whether or not an arbitrary element x is contained in the set. Church's thesis states that a set is decidable if and only if the set is recursive(see p. 5). Then, by Church's thesis, if a set is not recursive, it is undecidable. But what does this say about a Turing machine and the algorithm for deciding if a Turing machine halts on input x?

A solution to a decision problem requires the computation to return an answer for every instance of the problem. Relaxing this restriction, the notion of a partial solution can be obtained.

Definition 4.0:

> A partial solution to a decision problem, P, is an effective procedure which returns a positive response for every $p \in P$ whose answer is yes. If the answer to p is negative, however, the procedure may return no or fail to produce an answer.[7, 257]

Just as a solution to a decision problem can be formulated as a question of membership in a recursive set, a partial solution to a decision problem is equivalent to the question of membership in a recursively enumerable set.

The Church-Turing thesis for decision problems states, "There is an effective procedure to solve a decision problem if, and only if, there is a Turing Machine accepting a recursive language[set] which solves the problem."

[7, 257]

The extended Church-Turing thesis for decision problems, which uses the notion of a partial solution, states,

"A decision problem P is partially solvable if, and only if, there is a Turing machine that accepts precisely the elements of P whose answer is yes" [7, 257].

Keeping these things in mind, define the set H, such that,

$$H = \{x: \phi_x(x)\downarrow\}.$$

In other words, H is the set containing all x such that the value of the function $\phi_x(x)$ is decidable, denoted $\downarrow$. The decision problem for H(x) is one method of stating the halting problem (defined on p. 23).

Next, consider the function:

$$h(x) = \begin{cases} 1 & \text{if } \phi_x(x)\downarrow \\ 0 & \text{otherwise.} \end{cases}$$

Note that the function h(x) = 1 for input x if the Turing machine T = T(x) halts on input x_i. That is, T halts on an input consisting of its own description x. This h describes a function and a set which are not recursive. To show this, assume that h is recursive.

Define $$g(x) = \begin{cases} \phi_x(x)+1 & \text{if } h(x)=1 \\ 0 & \text{if } h(x) = 0. \end{cases}$$

Now notice that if h(x) is recursive, so is g(x), as

$\phi_x(x)$ will always be defined when h(x) = 1.

Let m be an index for g so that $g = \phi_m$.

Then h(m) = 1 since g is recursive, and

$$\phi_m(m) = g(m) = \phi_m(m) + 1$$

which is a contradiction.

Therefore, the assumption that h(x) is recursive must be false.

One result of this, using the function h, is to show that $H = \{x\colon \phi_x(x)\downarrow\}$ is not recursive by noticing that

$$h(x) = \begin{cases} 1 & \text{if } \phi_x(x)\downarrow \\ 0 & \text{otherwise} \end{cases}$$

is the characteristic function of H. The question of solvability given a nonrecursive function suggests that there is no algorithm which will accept some program x with input y, and decide whether or not program x will halt using input y.

Unsolvability of the Halting Problem

To this point, various background information has been presented. All of these facts will allow a more in-depth discussion of the decision problem and its ramifications when applied to the Turing machine. Questions on the solvability of various functions and algorithms in terms of each of the following can now be approached:

1)given a Turing machine and a specific algorithm, whether or not the Turing machine will halt;

2)given specific input whether or not the Turing machine will ever halt, or;

3)whether the Turing machine will loop infinitely or just hang on a given input.

These questions are of legitimate concern to mathematicians and computer scientists because of the implications in these fields. If various problems are known to be solvable - or it is known that a Turing machine will halt (find answer to the given question), then any equivalent problem also is solvable.

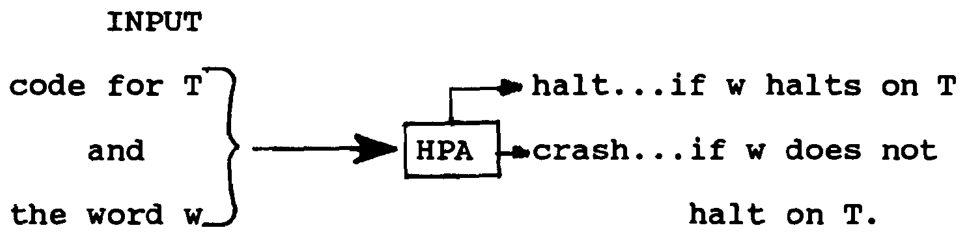
To this point the discussion has been in terms of decidability and the ability to determine if a decision procedure exists in relation to some given countably infinite set of problems. The problem of solvability requires a different approach and statement of the halting problem in terms of the infinite sets of problems.[5, 247]

Another method of defining the halting problem in terms of an algorithmic procedure is,

> "the halting problem for a given algorithm is the problem of determining whether or not a procedure exists which will determine if a given machine when executed with a given input eventually terminates." [7, 169]

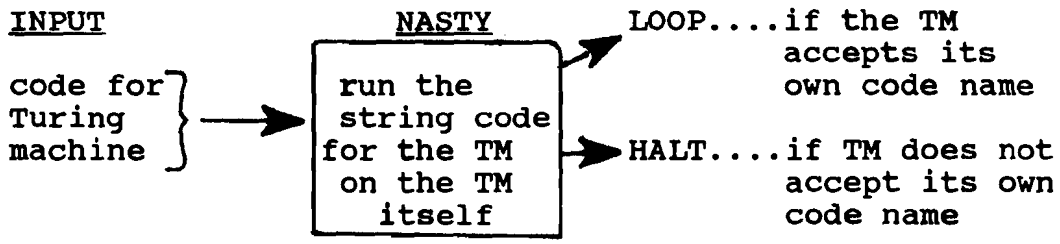
Theorem: Given some arbitrary Turing machine, T, and some arbitrary string, w, there is no algorithm to decide whether T halts when given the input w.

PROOF: Assume there were some Turing machine called HPA (halting-problem-answer) that takes as input the code (recall Gödel numbering) for any Turing machine, T, and any string, w, and leaves an answer on its tape yes or no. Assume HPA reaches h if w halts on T and HPA loops or hangs if w does not halt on T:

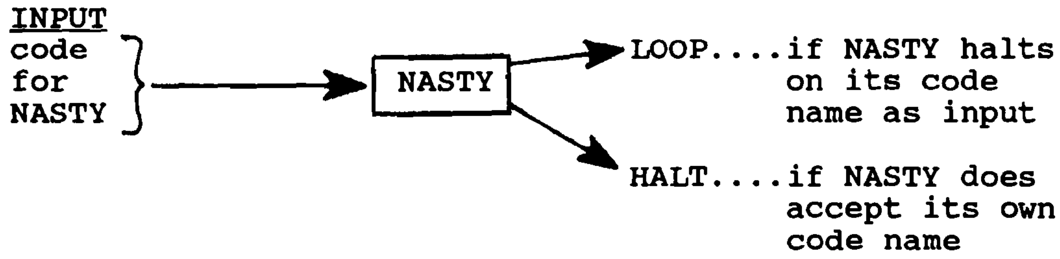


Using this HPA, construct a different Turing machine called NASTY. The input into NASTY is the code of any Turing machine. NASTY then asks whether the Turing machine can accept it's own code word as input. NASTY will act like HPA with the input. The input is the code of the Turing machine and the code of the string w to be tested. Now, change the halt state of HPA into an infinite loop and change all of the crashes in HPA into successful halts.

NASTY will do this:


INPUT
code for Turing machine
NASTY
run the string code for the TM on the TM itself
LOOP....if the TM accepts its own code name
HALT....if TM does not accept its own code name

Now feed NASTY to itself.


INPUT
code for NASTY
NASTY
LOOP....if NASTY halts on its code name as input
HALT....if NASTY does accept its own code name

From this, notice that NASTY will halt when fed its own code name as input if NASTY does not halt when fed its code name as input. NASTY will loop when fed its code name if NASTY halts when fed its own code name as input.

This paradox is similar to Russell's paradox, (simply stated, does the set that contains all sets contain itself), and no such Turing machine as NASTY can exist. Therefore, no Turing machine such as HPA can exist and the halting problem for Turing machines is unsolvable.

Church's thesis states that there is an effective procedure to solve a decision problem if and only if there is a Turing machine accepting a recursive set that solves the problem. Church's thesis then, implies that the halting problem for Turing machines is undecidable. This follows since all recursive functions are computable by Turing machines and the halting problem itself is not recursive (p. 37). The undecidabilty of the halting problem is significant considering how valuable such a decison procedure would be if, in fact, there was one.

<u>Method of Reduction to Halting Problem</u>

Recall the set H = $\{x: \phi_x(x)\downarrow\}$ and the decison problem for this set. Since it has been previously determined that H is not recursive, and hence undecidable by Church's thesis, one may use the undecidability of the set H (or equivalent statements implied by Church's thesis) to prove that other problems are also undecidable. This may be accomplished by demonstrating that the decidability of a given problem implies the decidability of the halting problem. Since the halting problem is undecidable, the other problem must also be undecidable. This method of <u>reducing</u> the halting problem to another problem, in order to prove the undecidability of the latter, is one of the most important methods for proving undecidability. In order to discuss the halting problem in terms of

Turing-decidability, once having established that a particular set is not Turing-decidable, the unsolvability of a great variety of problems follows. These results can be proven using the method of reduction; it can be shown that if some set, say L_1, were Turing-decidable, then so would be some other set L_2 already known not to be Turing-decidable. In order to show that if L_1 were Turing-decidable, then L_2 would be as well, it suffices to show any algorithm for deciding membership in L_1 could be used, with some modification, to decide membership in L_2.

In this method of reduction, it is of utmost importance to understand the "direction" in which the reduction is to be applied. To show that a set L_1 is not Turing-decidable, a set L_2 which is not Turing-decidable must be identified and then L_2 reduced to L_1. To reduce in the other direction would achieve nothing.

The method of reduction can be used to prove some statements which are examples of unsolvable problems. Given a Turing machine, TM, and an input string w, it is not possible to determine if TM will halt on input w. Certainly with only a minor adjustment in the proof, it could similarly be shown that for a certain fixed machine TM_0, given an input string w, it is an undecidable problem to determine if TM_0 will halt on input w. With the knowledge that these two similar statements are undecidable, other statements of problems for Turing machines can be considered, such as:

(a) Given a Turing machine M, does M halt on the empty tape(λ)? and

(b) Given a Turing machine M, does M halt on every input string?

Using known undecidable problems, proceed by reducing one of these to (a) or (b).

For (a), begin with M_0, a Turing machine that decides the set $\{L(M):M \text{ accepts } \lambda\}$. Given any Turing machine TM and input string w, produce the Turing machine TM_w which will operate in the following manner.

Starting on the empty tape, TM_w writes w on its tape and then simulates M.

Choose some $w' \in L(TM_w)$ as an input for M_0.

If M_0 halts with yes on its tape, then TM_w accepts λ, and therefore M accepts w.

If M_0 halts with no on its tape, then TM_w does not accept λ, and therefore M does not accept w.

So M could be used to answer the general question of whether an arbitrary Turing machine accepts an arbitrary string, and this contradicts the original statement. Therefore, by reduction, it has been shown that (a) is undecidable.

Similarly the use of reduction can show that if (b) were a solvable problem, then (a) would be solvable as well. If one could tell whether there were any input on which a Turing machine halted, then one could tell whether a Turing machine halted on the empty string as follows. Given a Turing machine TM, to tell whether TM halts on the empty string, first modify TM so that it erases any input given to it and then proceeds to compute as though it had actually been given the empty string. This modified machine TM′ halts for no input string or it halts for every input string. So, if TM′ halts for any input string, then TM′ halts for all input strings, and TM halts on the empty string. There is only a need to check whether TM′ accepts some string. Therefore (b) is also undecidable.

Almost all undecidability proofs either directly or indirectly involve a reduction of the halting problem to some other problem. Some equivalent unsolvable problems for Turing machines can be stated at this point. The method of reduction can be used to show their respective undecidabilities also (see [6, 283-5]). These are:

(1) Given two Turing machines TM_1 and TM_2, do they halt on the same input string?

and

(2) Given a Turing machine TM, is there any string at all on which TM halts?

Since the method of reduction has been illustrated, and similar problems for Turing machine have been stated, the question of how to apply the undecidability of these problems to other mathematical problems may be approached.

Chapter Five

The Busy Beaver: An Uncomputable Function

Chapters one through four dealt with the idea of solvability, computability, and methods for determining which functions are computable. This chapter examines a function which is uncomputable.

One of the early examples of an uncomputable function was created by Tibor Rado of Ohio State University in 1962. Rado's uncomputable function involves the search for a specific Turing machine (now called the "busy beaver") which satisfies the following requirements: 1)Collect all Turing machines with some fixed number of states N. 2)These machines begin operation on a tape initially filled with blanks. Suppose that all the Turing machines which do not halt are excluded from this set of machines. 3)From the remaining machines, the set of "busy beavers" are the machines that print the largest number of distinct ones for a given N in succession, before halting. This number of ones, for each value of N, is the value of Rado's function; usually designated $\sigma(N)$. The constructive proof that $\sigma(N)$ is not computable proceeds by assuming it can be computed and then deriving a contradiction.

This construction seems defective as it is based on the assumption that all the N-state Turing machines which do not halt may be sorted out in advance. As it has been shown, this may not be possible. Consider, then, how an attempt could be made to compute $\sigma(N)$ by brute force. All possible N-state Turing machines could be listed in some numerical order (perhaps using Godel numbering) and each one then simulated on a universal Turing machine (a TM that is capable of solving any problem which is solvable). Then the machine or machines that print the most distinct ones is the set of "busy beavers."

This does not seem to avoid the defect in the original construction. Although some of the N-state machines that do not halt may be eliminated by simple algorithms, there will be other machines for which a decision can not be made. Since one cannot determine that a particular machine does not halt, the machine cannot be eliminated from the list of N-state machines and the simulation must continue. For this reason, the machine may actually never halt, and there is no guarantee the computation of $\sigma(N)$ can be carried through to completion.

A one-state busy beaver can write only 1 one, and a two-state busy beaver will write three distinct ones before halting. It is now known that a three-state busy beaver writes six distinct ones before it halts; a four-state busy beaver writes thirteen distinct ones. Until late 1984, the top candidate for a five-state busy beaver printed 501 distinct ones. An amateur mathematician interested in the problem of the five-state busy beaver, George Uhing, built a small computer which tested sequentially the selection of 64,403,380,965,376 different possible five-state Turing machines. [Science News, 89] Uhing let his computer run for about three weeks sifting through several million possibilities and found one five-state Turing machine that printed 1,915 distinct ones as it went through more than two million moves. It is not yet known if there is a machine which will print out even more distinct ones, but the jump in the value of $\sigma(N)$ from a four-state busy beaver to a five-state busy beaver is already very large. The ability to distinguish between machines which halt and machines which do not halt diminishes quickly with the addition of just one more state. This is just one example of an uncomputable function and there is ongoing research to determine the largest value of N for which $\sigma(N)$ can be found.

BIBLIOGRAPHY

[1] Brainerd, Walter and Landweber, Lawrence. Theory of Computation. John Wiley and Sons, Inc.: 1974.

[2] Cohen, Daniel. Introduction to Computer Theory. John Wiley and Sons, Inc.: 1986.

[3] Copi, Irving. Symbolic Logic. Macmillan Publishing Co., Inc.: 1979.

[4] Hopcroft, John. "Turing Machines". Scientific American. May, 1984, pp. 94-96.

[5] Kleene, Stephen Cole. Mathematical Logic. John Wiley and Sons, Inc.: 1967.

[6] Lewis, Harry and Pappadimitriou, Christos. Elements of the Theory of Computation. Prentice-Hall, Inc.: 1981.

[7] McNaughton, Robert. Elementary Computability, Formal Languages, and Automata. Prentice-Hall, Inc.: 1982.

[8] Science News. "Looking for a Busy Beaver", February 9, 1985, p. 89.

[7] Sudkamp, Thomas. Languages and Machines: An Introduction to the Theory of Computer Science. Addison-Wesley Publishing Company, Inc.: 1988.