ANNALS OF MATHEMATICS
Vol. 74, No. 3, November, 1961
Printed in Japan

RECURSIVE UNSOLVABILITY OF POST'S PROBLEM OF "TAG" AND OTHER TOPICS IN THEORY OF TURING MACHINES*

BY MARVIN L. MINSKY

(Received August 15, 1960)

TABLE OF CONTENTS

Introduction

The equivalence of the notions of effective computability as based (1) on formal systems (e.g., those of Post), and (2) on computing machines (e.g., those of Turing) has been shown in a number of ways. The main results of this paper show that the same notions of computability can be realized within

(1) the highly restricted *monogenic* formal systems called by Post the "Tag" systems, and

(2) within a peculiarly restricted variant of Turing machine which has two tapes, but can neither write on nor erase these tapes.

From these, or rather from the arithmetization device used in their construction, we obtain also an interesting basis for recursive function theory involving *programs* of only the simplest arithmetic operations.

We show first how Turing machines can be regarded as programmed computers. Then by defining a hierarchy of programs which perform certain arithmetic transformations, we obtain the representation in terms of the restricted two-tape machines. These machines, in turn, can be represented in terms of Post normal canonical systems in such a way that each *instruction* for the machine corresponds to a set of productions in a system which has the *monogenic* property (for each string in the Post system just one production can operate). This settles the questions raised

* The work reported in this paper was performed at Lincoln Laboratory, a center for research operated by Massachusetts Institute of Technology, under Contract AF 19(604)-5200.

by Post [1], since the productions obtained satisfy the "Tag" condition proposed in that paper.

Examination of the 2-tape non-writing machines was suggested by some work of Rabin and Scott [2] who showed the undecidability of a certain problem concerning finite automata with two tapes. It was suggested to us by John McCarthy that this might have some explanation simpler than that in the proof of the Rabin-Scott theorem, and he conjectured that a non-writing machine with some small number of tapes might be equivalent to a universal Turing machine. This proved correct, and we were able to establish the result first in a four-tape machine which used the basic arithmetic device of the present paper. Then, two of the tapes were eliminated by the prime-factor method used for Theorem I. (As a result we obtain the Rabin-Scott theorem as an immediate corollary of the unsolvability of a halting problem for Turing machines. In addition, we obtain easily Wang's result that one can construct essentially universal Turing machines which can write but not erase; and we obtain also a rather startling bound on the number of symbols needed on a Turing machine tape during a general computation. Three symbols are necessary, and no more.) We then proceeded to represent the two-tape machine as a Post normal system with the object of obtaining particularly simple unsolvable decision problems (which will be described elsewhere). On the suggestion of Martin Davis we examined Post's problem of "Tag". It proved possible to adapt the Post system to yield a representation in the "Tag" form as described in §2 below.

1. Two-tape non-writing Turing machines

We will first consider 2-*tape non-writing machines* each of which is composed of

(i) a finite automaton, and

(ii) two semi-infinite *tapes*, each of which is completely blank except for a single mark indicating the location of the square at the end of the tape.

These machines can move either tape in either direction, and can sense when a tape reaches its end. Our task is to show that, given an ordinary Turing machine T, we can construct a two-tape machine T^* which is, in a sense described below, equivalent to T. (The 2-tape machines, regarded as programmed computers, are described more precisely in §1.3 below.)

1.1. *Turing machines as programmed computers.* Suppose that T' is the Turing machine to be represented. Let T be the equivalent 2-sym-

bol Turing machine as obtained by Shannon [4]. This is non-essential here, but simplifies details of the arguments. We can think of a Turing machine as a formal system (e.g., Davis [6, p. 88 ff.]) or as a programmed computer (e.g., Wang [5]). Here we prefer the latter.

A *program* is a sequence of *Instructions*, I_α, each of which specifies

(i) an operation to be performed, and

(ii) the next instruction to be executed.

The "transfer" (ii) may be conditional on the outcome of (i) as in R_{q_i} below. The transfer (ii) is ordinarily specified by an expression *go to* I_β, but when (for brevity) no such expression appears, it is understood that (ii) is to be the next instruction in the program sequence. Execution of a program ordinarily begins with obeying the first instruction.

Now suppose that the formal quintuples for the machine T are

$$q_i s_j : s_{ij} q_{ij} d_{ij}$$

where the q's represent states, the s's represent symbols (0 or 1) and the d's represent directions of tape motion (*left* or *right*). Using these quintuples (or rather, their index functions) we can represent each *state* q_i of T by a sequence (*sub-routine*) of five instructions:

R_{q_i}: Read the tape. If the current symbol is 0, go to W_{i0}.
If the current symbol is 1, go to W_{i1}.

W_{i0}: Write s_{i0} in the current square. Go to M_{i0}.

M_{i0}: Move in direction d_{i0}. Go to $R_{q_{i0}}$.

W_{i1}: Write s_{i1} in the current square. Go to M_{i1}.

M_{i1}: Move in direction d_{i1}. Go to $R_{q_{i1}}$.

If, for some i and j there is no quintuple $q_i s_j$:, we can make W_{ij} a *Halt* instruction which terminates a program execution. This computer program embodies very smoothly the usual mechanical interpretation of the Turing quintuples.

For each instruction of T's program, the 2-tape non-writing machine T^* will have a corresponding *set* of instructions, and whenever T executes an instruction, T^* will execute instructions in the corresponding set. The *complete state* of T is described, after each instruction, by

(1) the entire contents of its tape,

(2) the location of its reading head on its tape, and

(3) its *internal state* as given by the index of its next instruction.

We describe in the next section how (1) and (2) are represented in the machine T^*; (3) is represented in T^* as in T by the index of the next instruction. We will refer to T's tape as the T-tape and the tapes of T^* as the T_1^*-tape and the T_2^*-tape. All three tapes are semi-infinite with

ends to the left. Counting from the left we can refer to the n^{th} square of a tape, and we will use the symbol x to denote the current square of the T-tape. It is convenient to assume that the extreme left-hand square of each tape contains a special symbol: in the case of T, encountering this symbol (on the -1^{st} square) should correspond to a Halt or undefined result; in the case of T^* encountering such a mark (on a 0^{th} square) will affect the choice of T^*'s next instruction.

1.2. *Representation of the Turing machine T as a two-tape non-writing machine T^*.* At any moment in the operation of T, the T-tape contains a finite amount of written data, all of which may be represented by the single integer k whose binary expansion is precisely the contents of the T-tape up to the 1 furthest to the right. The 0^{th} square of the tape is taken to contain the least significant digit of k. Then the content of the current square, x, is always equal to the x^{th} digit of the binary expansion of k. (Note: if $k = \sum_{i=0}^{m} a_i 2^i$ then a_i is the i^{th} digit of k.)

At the beginning of each instruction of T, the state of affairs of T^* will be represented in the following manner, suggestive of a Gödel-numbering. *Tape T_2^* will be set at its end. Tape T_1^* will be moved so that its reading head is $\boxed{2^k 3^{2^x}}$ squares to the right of its end.* After each *read*, *write*, or *move* instruction of T, we will find that this condition of the machine T^* will be restored, except for the appropriate changes in k and x. The meaning of the number $2^k 3^{2^x}$ will become clear as we proceed; we note only that we could use any other pair of prime numbers for '2' and '3'.

Execution of the instructions R_{q_i}, W_{ij} and M_{ij} require T^* to manipulate the length of T_1^* so as to change properly the exponents of 2 and 3 in its prime factorization.

(1) *Moving the T-tape; the M instructions.* Moving the T-tape to the right is equivalent to changing x to $x+1$; i.e., to changing $2^k 3^{2^x}$ to $2^k 3^{2^{x+1}}$. This means that tape T_1^* *must be moved to the right so that the number of* 3's *in its prime factorization will be doubled.* We will do this by an iterative method in which $2^k 3^{2^x}$ is converted first to $2^k 5^{2^x}$ and then to $2^k 9^{2^x} = 2^k 3^{2^{x+1}}$. The first step will be accomplished by a program $C(3, 5)$ which will

(i) divide the length of the T_1^* tape by 3; then

(ii) multiply its length by 5, and repeat this cyclically until step (i) fails because the length is no longer divisible by 3.

The second step is accomplished by a similar program $C(5, 9)$ which converts all 5-terms in the factorization of the length of T_1^* into $9 = 3^2$-terms. We will describe in detail the program for $C(S, T)$ in § 1.3; we must have

g.c.d.$(S, T) = 1$. We represent this process as:

$$2^k3^{2^x} \xrightarrow{C(3,5)} 2^k5^{2^x} \xrightarrow{C(5,9)} 2^k9^{2^x} = 2^k3^{2^{x+1}} .$$

Similarly, moving the T-tape to the left will be represented by the operations:

$$2^k3^{2^x} \xrightarrow{C(9,5)} 2^k5^{2^{x-1}} \xrightarrow{C(5,3)} 2^k3^{2^{x-1}} .$$

(2) *Writing on the T-tape; the W instructions.* If the x^{th} digit of k is a 0 and we wish to convert it to a 1, this is equivalent to adding 2^x to k, and can be done with the same $C(S, T)$ operations:

$$2^k3^{2^x} \xrightarrow{C(3,5)} 2^k5^{2^x} \xrightarrow{C(5,6)} 2^k6^{2^x} = 2^{k+2^x}3^{2^x} .$$

If we wish to replace a 1 by a zero, the process is reversed, applying first $C(6, 5)$ and then $C(5, 3)$:

$$2^k3^{2^x} \xrightarrow{C(6,5)} 2^{k-2^x}5^{2^x} \xrightarrow{C(5,3)} 2^{k-2^x}3^{2^x} .$$

(3) *Reading the T-tape; the R_q instructions.* This is the key operation. We have to determine whether the x^{th} digit of k is 0 or 1. Our method is to carry out repeated subtraction of 2^x from k, until k is exhausted. We keep track of the parity of the number of times the subtraction was completed, and if this parity is even, then x must have been 0; if odd, x was 1. (Higher digits of k cause even numbers of subtractions, and lower digits affect only the remainder.) The first step in the method is to *make a copy* of k so that, when the process is complete, we will be able to restore the T-tape. We will work with the exponent of 5:

$$2^k3^{2^x} \xrightarrow{C(2,35)} 5^k7^k3^{2^x} \xrightarrow{C(7,2)} 2^k5^k3^{2^x} .$$

We then repeatedly perform a process like that in (2) above:

$$2^k5^k3^{2^x} \xrightarrow{C(15,7)} 2^k5^{k-2^x}7^{2^x} \xrightarrow{C(35,3)} 2^k5^{(k-2\cdot 2^x)}3^{2^x} \xrightarrow{C(15,7)}$$

$$2^k5^{(k-3\cdot 2^x)}7^{2^x} \xrightarrow{C(35,3)} \cdots ,$$

checking at each step to see whether the subtraction was completed (i.e., whether there remain both 3's and 7's), and keeping track of the parity (i.e., whether the last operation was a $C(35, 3)$ operation or a $C(15, 7)$ operation). When the process terminates, the original tape will be restored by a single $C(7, 3)$ operation.

This completes the description of the required exponent arithmetic. It remains to show that all these operations can be realized in a program built up from the basic operations available to T^*. We describe first pro-

grams for certain operations MPY(T) and DIV(S) which respectively multiply the length of T_1^* by an integer T, or divide it by an integer S. These operations are essential also in our proof of the Post result. Then, using MPY(T) and DIV(S), we show how to construct the program for $C(S, T)$ used above. Finally, we show how copies of these programs are linked together to form the complete program for T^*.

1.3. *The programs (sub-routines) for* MPY *and* DIV. We first define the instructions for the machine T^*. There are four possible forms:

R_1: Move T_1^* to the right. Go to I_α .

R_2: Move T_2^* to the right. Go to I_α .

L_1: Move T_1^* to the left. If tape ends, go to I_{α_1} .
If not, go to I_{α_2} .

L_2: Move T_2^* to the left. If tape ends, go to I_{α_1} .
If not, go to I_{α_2} .

Thus on moving a tape to the left, the machine may encounter the terminal square, and can use that information to affect the choice of its next instruction. This cannot happen, of course, in moving to the right. (We assume that the machine actually moves onto the terminal square when it encounters a left end.)

The program for "MPY(T). Go to I_α" is:

R_1.

(1) L_1. If tape ends go to (2). ----→ (2) R_1.

R_2. } $T - 1$ terms

... }

R_2. }

L_2. If tape ends, go to I_α .
If not, go to (2) .

R_2. Go to (1) .

The first part of the program moves the T_2^*-tape T squares to the right for each motion of T_1^* to the left. The second part of the program transfers the multiplied tape length back to T_1^*. Whenever, in the programs below, we see the symbol MPY(T), it is understood that a copy of the above program is to be inserted there, with appropriate substitution for the instruction names '(1)' and '(2)'.

The program for DIV(S) is complicated by the requirement that if the

length of T_1^* is *not* exactly divisible by S it should be restored to its original length before leaving the program; while, if the length is divisible by S, it should be so divided. The specification of the next instruction is conditional on which of these events occurs.

"DIV(S). If possible, go to I_{α_1}. If not, go to I_{α_2}":

R_2.

(1) L_1. If tape ends go to (S).
L_1. If tape ends go to ($S-1$).
...
L_1. If tape ends go to (2).
L_1. If tape ends go to (E).
R_2. Go to (1).

(2) R_1.
(3) R_1.
...
(S) R_1.
L_2. If tape ends go to I_{α_2}.
R_1. Go to (2).

(E) R_1.
L_2. If tape ends, go to I_{α_1}.
If not, go to (E).

If division is exact, the program executes the iteration under (E) and goes on to I_{α_1}. If not, the operations (2), ..., (S) restore the remainder, and then the original length of T_1^*, and the program goes on with the instruction I_{α_2}.

Next we assemble the programs for the operations $C(S, T)$ which replace all factors of S in the length of T_1^* by factors of T, (where S and T have no common factors). It is important for the solution of the Post problem that we write a program for $C(S, T)$ using only MPY and DIV sub-programs with prime parameters. Suppose that $S = p_{i_1}p_{i_2}\cdots p_{i_m}$ and $T = p_{j_1}p_{j_2}\cdots p_{j_n}$ where no p_j is one of the p_i's. Then we can write

"$C(S, T)$" $\equiv$ (1) DIV(p_{i_1}). If no division go to I_{α_1}.
DIV(p_{i_2}). If no division go to (m).
... ...
DIV(p_{i_m}). If no division go to (2).
MPY(p_{j_1}).
MPY(p_{j_2}).
...
MPY(p_{j_n}). Go to (1).

(2) MPY($p_{i_{m-1}}$).

...

(m) MPY(p_{i_1}). Go to I_{α_1}.

We can now write the instructions for the program of the equivalent machine T^*:

If d_{ij} = "LEFT"	d_{ij} = "RIGHT"
$\boxed{M_{ij}} \equiv C(9, 5)$.‡	$\boxed{M_{ij}} \equiv C(3, 5)$.
$C(5, 3)$. Go to $R_{q_{ij}}$.	$C(5, 9)$. Go to $R_{q_{ij}}$.

If	$S_{i0} = 0$,	If $s_{i1} = 1$,	If $s_{i0} = 1$,	If $s_{i1} = 0$,
then:	$\boxed{W_{i0}} \equiv$ Go to M_{i0}	$\boxed{W_{i1}} \equiv$ Go to M_{i1}	$\boxed{W_{i0}} \equiv C(3, 5)$. $C(5, 6)$. Go to M_{i0}	$\boxed{W_{i1}} \equiv C(6, 5)$ $C(5, 3)$. Go to M_{i1}.

That is, if no change in a symbol is required, no program is required.

Finally $\boxed{R_{q_i}} \equiv$

$C(2, 35)$
$C(7, 2)$
(I_i') $C(15, 7)$
DIV(3). If no division go to (I_i'').
MPY(3).
$C(7, 3)$. Go to W_{i0}.
(I_i'') $C(35, 3)$.
DIV(7). If no division go to (I_i').
MPY(7)
$C(7, 3)$. Go to W_{i1}.

Note how the parity of the number of subtractions determines the choice of the next instruction W_{ij}. This completes construction of the program for the machine T^*, and hence the proof of

THEOREM I. *T^* represents the machine T (and accordingly, the machine T') in the following sense. Suppose that the machine T is started in state q_i at the x^{th} square of its tape and with the binary number k written on its tape. Suppose also that the machine T^* is started at instruction R_{q_i} with its tape T_1^* at its $2^k 3^{2^x\text{th}}$ square (and its tape T_2^* at its end square). Then if T ultimately halts on its y^{th} square with the*

‡ At this point one could test to see if the machine T has tried, illegally, to pass the left end of its tape, and if so, halt the machine T^*.

binary number N on its tape, T^* will ultimately halt with tape T_1^* at its $2^N 3^{2^N}$th square. If T is a universal machine, then so will be T^*, with computations understood in the above sense.

We could, if desired, remove the exponent of 3 in the terminal result by replacing each Halt instruction by the program [$C(3, 1)$. Halt.]. No way, however, is seen to replace the terminal number 2^N by the number N, or to perform the inverse operation. (This can, however, be done when we represent the machine T^* by a Post normal system.)

2. Monogenic normal systems

2.0. *The problem of "Tag".* Let A be a finite alphabet of letters $a_1, \cdots, a_n$; and let W be an associated set of *words*: for each i, W_i is some fixed string or *word* of letters of A. Let P be some integer, and consider the following process applied to some initially given string S of the letters: *Examine the first letter of the string S. If it is a_i then*

(i) *remove from S the first P letters, and*

(ii) *attach to its end the word W_i.*

Perform the same operation on the resulting string, and repeat the process so long as the resulting string has P or more letters. The "Tag problem", for given A, P, and W is to give a decision procedure which, given S, will tell if the process will ever terminate. A second problem, given A, P, W, and S is to decide, for any string S', whether that string will occur during the process.

One can think of this as a process in which one end of the string, advancing at a constant rate, is trying to catch up with the other end of the string. The problem is significantly different from the usual *word* problems in that the process considered is *monogenic* (*each string can be replaced by just one other string*), and one might suppose that this would make it easier to decide such problems. However, our result is that there are problems of both kinds which cannot be decided, (with $P = 6$).

2.1. *Recursive unsolvability of the Post "Tag" systems.* The "Tag" systems which concerned Post are normal (canonical) systems with the following constraint on the constants in the productions. In a normal system, all productions have the form

$$g_i\alpha \rightarrow \alpha h_i$$

where g_i and h_i are fixed strings of letters of some alphabet and α is an arbitrary string. The special constraint is that

(1) *all the g_i have the same length P, and*

(2) *the string h_i depends only on the first letter in the string g_i.*

We will show (Theorem II) that any Turing machine can be represented, in a simple sense, as a "Tag" system, obtaining the unsolvability of the "Tag" problems for those systems representing, say, universal Turing machines. Incidentally, we obtain an independent proof of the basic Post theorem on the universality of normal canonical systems, and obtain a positive answer to his question of whether monogenic normal systems can be universal.

Consider a Turing machine T' to be represented. It follows from the proof of Theorem I that the equivalent 2-symbol machine T can be represented by a 2-tape non-writing machine with a program of the form $I_1, \cdots, I_M$ where each instruction has one of the two forms:

(1) (I_j) MPY(K_j). Go to I_{j_1}

or

(2) (I_j) DIV(K_j). If division is even go to I_{j_1}. If not go to I_{j_2}.

In each case, K_j is one of the four primes 2, 3, 5, 7 used in the above proof. We have to show how an arbitrary program composed of instructions of these two types can be represented by a Post Normal System with the given constraints on the string constants g_i and h_i.

To represent such a program, we introduce a set of distinct symbols and productions for each of the instructions. In each production $g\alpha \rightarrow \alpha h$ the antecedent string g will have precisely P letters (where $P = 2 \cdot 3 \cdot 5 \cdot 7 = 210$), meeting Post's "Tag" condition.[1] If the j^{th} instruction is MPY(K_j), we will use $P + 3$ letters $A_j, a_j, B_{j1}, B_{j2}, \cdots, B_{jP}, b_j$. The state of the program at the start of the instruction I_j will be represented by the existence of a string of the form $A_j a_j^n$ and the productions below will convert this to a string of the form $A_{j_1} a_{j_1}^{nK_j}$. (Since the consequent h of each production depends only on the first letter of the antecedent g, we will write only that letter on the left, it being understood that P letters are to be removed from the beginning of the string in each case.)

$I_j =$ MPY(W_j):

$$\begin{cases} A_j \rightarrow B_{j1} \cdots B_{jP} \\ a_j \rightarrow b_j^{P^2} \\ B_{ji} \rightarrow b_j^{(P-i+1)(P-1)} A_{j_1} \\ b_j \rightarrow a_{j_1}^{K_j}. \end{cases}$$

If $n = \alpha P + \beta$, $(\beta < P)$ we can trace the generation of strings:[2]

[1] In §3 we show P can be reduced to $2 \cdot 3 = 6$.

[2] The idea of this multiplication process is based on discussion with R. Silver.

$$A_j a_j^n = A_j a_j^{\alpha P+\beta} \to a_j^{\alpha P+\beta-P+1} B_{j1} \cdots B_{jP}$$
$$\to a_j^{\beta+1} B_{j1} \cdots B_{jP} b_j^{(\alpha-1)P^2} \to B_{j(P-\beta)} \cdots B_{jP} b_j^{\alpha P^2}$$
$$\to b_j^{\alpha P^2-P+\beta+1+(P-P+\beta+1)(P-1)} A_{j_1}$$
$$\to b_j^{\alpha P^2+\beta P} A_{j_1} = b_j^{nP} A_{j_1} \to A_{j_1} a_{j_1}^{nK_j} .$$

For division we use the productions:

$I_j = \text{DIV}(K_j)$:

$$\begin{cases} A_j \to B_{j1} B_{j2} \cdots B_{jP} & \\ a_j \to b_{j1} b_{j2} \cdots b_{jP} & \\ B_{ji} \to A_{j_1}^i a_{j_1}^{(P-i)/K_j} & \} K_j \mid i \\ b_{ji} \to a_{j_1}^{P/K_j} & \} K_j \mid i \\ B_{ji} \to A_{j_2}^i a_{j_2}^{P-i} & \} K_j \nmid i \\ b_{ji} \to a_{j_2}^P & \} K_j \nmid i \end{cases}$$

Let $n = \alpha P + \beta K_j + \gamma$ with $\beta < P/K_j$ and $\gamma < K_j$. Then

$$A_j a_j^n \to a_j^{n-P+1} B_{j1} B_{j2} \cdots B_{jP} = a_j^{(\alpha-1)P+\beta K_j+\gamma+1} B_{j1} \cdots B_{jP}$$
$$\to a_j^{\beta K_i+\gamma+1} B_{j1} \cdots B_{jP} (b_{j1} \cdots b_{jP})^{\alpha-1}$$
$$\to B_{j(P-\beta K_j-\gamma)} \cdots B_{jP} (b_{j1} \cdots b_{jP})^{\alpha}.$$

Now if $\gamma = 0$, this

$$\to [b_{j(P-\beta K_j)} \cdots b_{jP} b_{j1} \cdots b_{j(P-\beta K_j-1)}]^{\alpha-1} b_{j(P-\beta K_j)} \cdots b_{jP} A_{j_1}^{P-\beta K_j} a_{j_1}^{\beta}$$
$$\to A_{j_1} a_{j_1}^{\beta+(\alpha-1)P/K_j+P/K_j} = A_{j_1} a_{j_1}^{n/K_j} .$$

Otherwise $\gamma \neq 0$, and

$$\to [b_{j(P-\beta K_j-\gamma)} \cdots b_{jP} b_{j1} \cdots b_{j(P-\beta K_j-\gamma-1)}]^{\alpha-1} b_{j(P-\beta K_j-\gamma)} \cdots b_P$$
$$A_{j_2}^{P-\beta K_j-\gamma} a_{j_2}^{\beta K_j+\gamma}$$
$$\to A_{j_2} a_{j_2}^{\beta K_j+\gamma+(\alpha-1)P+P} \to A_{j_2} a_{j_2}^n .$$

Hence if division is possible we obtain a string $A_{j_1} a_{j_1}^{n/K_j}$, but if not, we obtain a string $A_{j_2} a_{j_2}^n$ of the original length.

Observe that the symbol A_j can be encountered as the first symbol in a string *only* when that string has the form $A_j a_j^m$, and that it occurs as the first symbol only in the first production of the instruction I_j. The productions of that instruction can yield precisely one new string beginning with an A, hence the system has the *monogenic* property: each string can be transformed into precisely one new string by the entire system of

productions. If we begin with, say, a string of the form $A_1 a_1^{2^{k_{3^{2^x}}}}$, the formal system will represent, in its production of one string after another, precisely the steps of the process carried out by the corresponding machine T^*. This completes the proof of Theorem II. *Any Turing machine has a representation as a monogenic "Tag" normal canonical system.*

A few further remarks will help to expose the power, even with the "Tag" condition, of monogenic normal systems.

2.2. A set S of strings in an alphabet A has a *normal extension* C if the set S is precisely the subset of strings in A produced by a normal system C in some larger alphabet. (We do not require that the initial string or *axiom* be a string in A.) We can show that any recursively enumerable set of integers can be represented by a *monogenic* normal canonical extension over a 1-symbol alphabet. The proof is complicated by the fact that, in a monogenic system, we cannot realize every recursive enumeration procedure directly because a non-terminating process cannot be circumvented by generation of other trees of strings. We will only sketch the method of proof.

Suppose that an infinite set J of integers is r.e. Choose a Turing machine T^J which somehow enumerates J *without repetition*. Construct a larger Turing machine M^J which can perform the operations of T^J while remembering (e.g., on a special part of its tape) an arbitrary integer, as follows: Let $J(n)$ be the n^{th} integer in the given enumeration of J. Suppose, inductively, that M^J has a record of $J(n)$. Let M^J then proceed to go through the operations of T^J, starting from the very beginning, and let it compare each integer generated with $J(n)$. When T^J generates $J(n)$, M^J allows T^J to generate the next value $J(n+1)$. At this point M^J converts $J(n+1)$ into a string of the form $0^{J(n+1)}1$, beginning at the left end of its tape, erases everything else on its tape, and moves to the extreme left, entering a special state q_w (which is not entered in any other case). The process is then repeated, storing and comparing the new value $J(n+1)$. If we let the symbols A_w and a_w in the corresponding monogenic "Tag" system of Theorem II be the letters of a restricted alphabet A_J, then the only pure strings in this alphabet will be the strings $A'_w a'^{3\cdot 2^{2^{J(n)}}}_w$. The factor 3 could be removed, if desired. A more elegant representation can be obtained by adjoining productions which convert $A_w a_w^{3\cdot 2^{2^{J(n)}}}$ into $Y^{J(n)}$ and back to $A'_w a'^{3\cdot 2^{2^{J(n)}}}_w$, with $Y^{J(n)}$ being the only pure strings in the alphabet with just the letter Y. Productions have been found which thus preserve the system as a monogenic normal extension over a one-letter alphabet, but we have been unable to preserve also the "Tag" property. The difficulty is in restoring the exponent. Perhaps this difficulty is re-

lated to that of going from exponential to polynomial diophantine equations?

2.3. *Halting Problems.* A "Tag" system *halts* when a string is produced with length P. We can arrange that this will happen for some chosen instruction I_j by assigning to I_j the productions

$$A_j \to H$$
$$a_j \to H$$
$$H \to H .$$

We could also *trap* the system at a special configuration of the Turing Machine: If we introduce the prime 37 into our representation, using strings of the form $A_j a_j 2^K 3^{2^x} \cdot 37$, then (since $3 \cdot 37 < P < 6 \cdot 37$) the machine will stop if and only if it reaches the condition $K = 0$, $x = 0$. It is easy to express an unsolvable Turing Machine problem in this form: does the machine ever erase its entire tape and return to the end square?

3. Recursive functions based on programs of arithmetic instructions

The proofs of Theorems I and II show that recursive functions may be calculated by programs of arithmetic operations on strings of symbols; in particular, we may formulate these results so that the operations act essentially only on the *lengths* of the strings. In the first case there are two integers S_1 and S_2 involved:

THEOREM Ia. *We can represent any partial recursive function $T(n)$ by a program operating on two integers S_1 and S_2 using instructions I_j of the forms*

(i) ADD 1 to S_j. Go to I_{j_1}.

(ii) SUBTRACT 1 from S_j, if $S_j \neq 0$, and go to I_{j_1}. Otherwise go to I_{j_2}.

That is, we can construct such a program with an I_x and I_y such that if we start at I_x with $S_1 = 2^n$ and $S_2 = 0$, the program will eventually stop at I_y with $S_1 = 2^{T(N)}$ and $S_2 = 0$.

THEOREM IIa. *We can represent any partial recursive function $T(n)$ by a program operating on one integer S using instructions I_j of the forms*

(i) MULTIPLY by K_j and go to I_{j_1}.

(ii) DIVIDE by K_j and go to I_{j_1}, if K_j/S. Otherwise go to I_{j_2}.

In this system we can again start at I_x with $S = 2^n$ and halt at I_y with $S = 2^{T(n)}$.

The proof of Theorem I shows that the values of K_j can be limited to

any four distinct prime numbers; e.g., 2, 3, 5, and 7, but we can remove two of these as follows;

We apply the system Ia to the system IIa. Suppose that the integer S in system IIa has the form 2^m3^n. And suppose that $m = S_1$ and $n = S_2$ are also the integers in a system Ia. Then there is an isomorphism between the effects of the following instruction pairs:

Ia		IIa
ADD 1 to S_1		MPY(2)
ADD 1 to S_2	$\Longleftrightarrow$	MPY(3)
SUBTRACT 1 from S_1		DIV(2)
SUBTRACT 1 from S_2		DIV(3) .

It follows from the universality of the system Ia (or from the proof of Theorem I) that the system IIa accordingly requires only two primes, e.g., 2 and 3, for its instructions. We can apply this result directly to the proof of Theorem II yielding the result that *we can choose* $P=6$ *and still obtain a universal "Tag" system.* In this case the representation of the original Turing machine involves an additional level of exponentiation, and we must supply that system with strings of the lengths $s = 2^{2^n}$ to obtain strings of length $2^{2^{T(n)}}$. It may be possible to simplify this form further. We have been unable to reduce P further, and the prospects seem gloomy. (We have been unable, as was Post, to prove a negative result for $P = 2$.)

It would be desirable to reduce the exponentiation level in this representation but the "Tag" systems seem intractible in regard to lower level manipulations. We have been unable even to find productions which can reduce the length $n > P$ of a string to $n - 1$, for arbitrary n.

4. Corollaries

4.1. *Unsolvability of certain 2-tape finite automata problems.* In [2] Rabin and Scott show the unsolvability of the problem of the existence of a pair of finite tapes which will cause a certain kind of non-writing two-tape automaton to enter a certain state. It is a consequence of our Theorem I that such automata are equivalent to arbitrary Turing machines, and we can obtain the Rabin-Scott result directly from a halting problem for a class of Turing machines. To see this, consider two-tape machines like those of § 1 with the additional ability to sense a certain symbol $\varnothing$. Let $V_1, V_2, \cdots$ be two-tape machines with V_j equivalent (in the sense of Theorem I) to T_j but with the modifications below:

(1) V_j is to be presented with two finite tapes on which are printed

arbitrary sequences of symbols. Imagine that the reading heads of V_j are placed initially at the right hand ends of these tapes. We assume that the machine can sense

(i) the symbol $\varnothing$, and

(ii) when a tape has passed one of its ends (in which case that tape is irretrievably lost).

(2) When actuated, V_j begins with the following program.

(1) L_1. If symbol is not $\varnothing$, go to (1).
If symbol is $\varnothing$, go to (2).
If tape falls off, decision is NO.

(2) L_2. If symbol is not $\varnothing$, go to (2).
If symbol is $\varnothing$, go to (3).
If tape falls off, decision is NO.

(3) R_1. R_1. R_1. If tape falls off, decision is NO.

This means that the pair of tapes is to be *rejected* in the Rabin-Scott sense unless there is at least one $\varnothing$ on each tape. At the completion of this program, the machine has one head located at the occurrence of $\varnothing$ farthest to the right on tape 2, and the other head located three squares to the right of the rightmost $\varnothing$ of tape 1. The remainder of the program for V_j is identical to that for the equivalent machine T_j^* of Theorem I except that

(i) wherever T_j^*'s program depends on the end-of-tape mark, V_j's program depends on the occurrence of $\varnothing$,

(ii) whenever T_j^* has a program Halt, V_j has a YES decision, and

(iii) if a tape falls off V_j it makes a NO decision (unless it has already made a YES decision). A YES decision in the Rabin-Scott sense consists of moving both tapes to the left until they fall off; a NO decision consists of moving the remaining tape to the right until it falls off.

A YES decision can occur only in the case that the corresponding machine T_j reaches a programmed halt. The corresponding event can occur for V_j with a given pair of tapes if and only if

(i) T_j eventually reaches a halt, and if

(ii) there is enough room to the right of the $\varnothing$'s on the two tapes to admit the representation of T_j's (finite) calculation.

Hence the problem of the existence of some pair of tapes which will be accepted by V_j is equivalent to the halting problem for the Turing machine T_j if that machine is started, with a blank tape, on its first square. (We required motion of tape 1 three squares to the right in instruction (3) above so that V_j could start with the initial data $n=2^0 3^{2^0}=3$.) Such a halting

problem is known to be recursively unsolvable: e.g., for each Turing machine T_n there is another Turing machine T'_n, which halts, for any initial tape, if and only if $T_n(n)$ halts.

4.2. *Densities of symbols on Turing machine tapes.* Application of Theorem I to ordinary Turing machines yields: *There exists a Universal Turing Machine with the property that at no time in any calculation will its tape contain more than three* 1's *with the remainder of the tape blank.* For three marks are sufficient to de-limit two intervals along the tape, and then it is a simple matter to adapt the above techniques for standard Turing machine. If desired, one of the three marks need never be moved. This result is obtained by directly simulating the 2-tape machine with a Turing machine.

This result shows that it would be futile to try to find interesting theorems relating classes of computations to the *densities* of data entries on the tapes of the corresponding Turing machines, unless some novel restriction is placed on the machines.

4.3. *Non-erasing Turing machines.* In [5] Hao Wang shows that one can make a Universal Turing machine which writes, but never erases or changes a symbol that has been written. This is done by re-copying with changes all the essential data, at some distance along the tape, each time a change would be required. Wang's construction is rather complicated, and can be simplified in view of the fact that one needs to copy only three symbols and the corresponding intervals. We begin by recoding our tape in the form

$$\cdots(\text{old data})\cdots a, 0, 1, 0, 1, 0, \cdots b, 0, 1, 0, 1, 0, \cdots c, 0 .$$

The letters represent our three marks, the 1's represent the old blanks, and the blanks in the odd-numbered squares will serve as markers for the copy operation. Copying without any alterations is done as follows:

Begin with the a at the left. Read and remember the current symbol; move one unit to the right and write a 1. Then move right to the first unoccupied *odd* square and there copy the stored symbol. Move left to the first occupied even square; go right one square, and repeat the cycle until the terminal symbol c has been copied. Finally return to the left until an a is encountered.

In all this we have written only on previously blank squares. If the copy is to involve an alteration, this will involve moving b or c or both to the right or left. The copying program will have to be modified to do this, but the details are quite simple and will be omitted here. Finally the above non-erasing machine could be converted into a 2-symbol machine

by, for example, using blocks of three squares and a representation

$$\cdots 110, 100, \cdots 100, 110, 100 \cdots 100, 110 .$$

4.4. *A simple diophantine predicate for* $\text{Prod}(y, x)$. The simple form of the instructions for the machines of Theorem IIa yield some simplifications in the details of the theory of diophantine predicates (6). We have not explored this thoroughly, but the argument below will show some of the possibilities.

For each instruction I_j of the system IIa, we will assign a distinct prime P_j (reserving the primes K_j for other purposes). We will use these primes to represent, in polynomial form, the *flow* of the program. Consider the polynomial equation

$$F_j(y, x) = (P_j y - P_{j_1} K_j x)^2 = 0 .$$

This will have exactly one integer solution given x, provided that $P_j \nmid x$, in which case y is obtained from x by

(i) multiplying x by K_j, and

(ii) removing a factor P_j from x and replacing it by a factor P_{j_1}.

We wish to restrict the existence of such a solution to the case in which x contains just one of the factors P_j (we do not mind if that factor is repeated). The following polynomial has that property, but requires an extra variable. Let $\psi_j = \prod_{i \neq j} P_i$ and let

$$(1) \qquad P_j(y, x, t) = (P_j y - P_{j_1} K_j x)^2 + \prod_{s=1}^{\psi_j} (sx + t\psi_j - 1)^2 = 0 ,$$

which will have just one solution provided that $P_j \mid x$ and no other $P_i \mid x$. In this solution the extra variable t is bounded by x. Thus, (1) represents the instruction

$$I_j\text{: MPY}(K_j), \quad \text{Go to } I_{j_1} .$$

Similarly, if I_j is an instruction "DIV(K_j), Go to I_{j_1}; otherwise Go to I_{j_2}" we can use the expression

$$\begin{aligned} P_j(y, x, t) = {} & [(P_j K_j y - P_{j_1} x)^2 + \prod_{s=1}^{\psi_j} (sx - t\psi_j - 1)^2] \\ & \cdot [(P_j y - P_{j_2} x)^2 + \prod_{s=1}^{K_j \psi_j} (sx - tK_j \psi_j - 1)^2] = 0 , \end{aligned}$$

which again will have a solution under the above conditions in which, if $K_j \mid x$, y will be obtained by dividing x by K_j and replacing P_j by P_{j_1}. In the other case, P_j is just replaced by P_{j_2}. Now we can put all the polynomials together by forming their product

$$\text{Prod}(y, x) = (E!\, t)(1 \leqq t \leqq x \wedge \prod_j P_j(y, x, t) = 0) .$$

Now if this has a solution (and it can have at most one, for given x) this is equivalent to the statement *y is an immediate successor of x* for a sequence of numbers produced by the system IIa. Hence this is the equivalent of Davis's predicate Prod$(x, y, z; x', y', z')$ but with a single bounded existential quantifier with a peculiar *unique solution* property. It seems likely that this possibly useful quantifier restriction could be extended by such methods to other quantifiers in the theory of diophantine predicates.

The polynomial $F(y, x, t)$ so obtained has the property that, for given x there is at most one pair (y, t) for which $F(y, x, t)=0$, and t is bounded by x. We can obtain from this some rather simple kinds of unsolvable problems. Choose an initial value x_0 for x. Obtain (at most one) solution for y and call it x_1. Substituting x_1 for x we obtain a new value for y, called x_2. The sequence so obtained may be infinite or it may terminate. Now we can, by appropriate Turing machine techniques, obtain such polynomials $F(y, x, t)$ for which the following problems are recursively unsolvable:

1. GIVEN x, to decide whether an arbitrary x' will occur in the sequence.
2. To decide whether an arbitrary x yields an infinite sequence.
3. To decide whether an arbitrary x yields a periodic sequence.

And so on. The construction of such problems is straightforward, in that there is a direct translation from well-known equivalent Turing machine problems. (By *solution* we mean, of course, solution in non-negative integers.) Similar unsolvable problems are known, of course, but not (to my knowledge) with the unique solution property, or with so few as just one extra variable. The degree of such polynomials, as obtained by our constructions, are unfortunately too large to be of practical interest. Careful construction might yield degrees of, perhaps, several hundred.[3]

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

REFERENCES

1. E. POST, *Formal reductions of the combinatorial decision problem*, Amer. J. Math., 65 (1943), 196-215.
2. M. O. RABIN and D. SCOTT, *Finite automata and their decision problems*, IBM Journal of Res. and Devel. 3, no. 2, April, 1959. (Our Cor. 4.1 is the final Theorem 19 of this paper.)
3. A. M. TURING, *On computable numbers*, etc., Proc. London Math. Soc., Ser. 2, 42 (1936) and 43 (1937).

[3] Note added in proof.
We have obtained, by use of Theorem II, a Universal Turing Machine with only 6-symbols and 6-states.

4. C. E. SHANNON, A universal Turing machine with two internal states, *Automata Studies*, Princeton, 1956.
5. H. WANG, *A variant of Turing's theory of computing machines*, J. Assoc. Computing Machinery, Jan. 1957.
6. M. DAVIS, *Computability and Unsolvability*, McGraw-Hill 1958. (Our predicate Prod(y, x) is equivalent to Davis's Prod$(x, y, z; x', y', z')$ as used in the proof of his Theorem 3.1, pp. 108–111.)