THE SPEED OF AN INFINITE COMPUTATION

By BARRY ABRAM BURD

A thesis submitted to
The Graduate School–New Brunswick
Rutgers, The State University of New Jersey,
in partial fulfillment of the requirements
for the degree of
Master of Science
Graduate Program in Computer Science,

Written under the direction of
Professor Ann Yasuhara
and approved by

Ann Yasuhara
Martin Dowd
Maureen C. Faull

New Brunswick, New Jersey
May, 1984

ABSTRACT OF THE THESIS

The **Speed** of an Infinite Computation

by BARRY ABRAM **BURD**, Ph.D.

Thesis Director: Professor Ann Yasuhara

The definition of a Turing Machine is generalized so that the machine performs compuations that have infinite input, and take infinitely many steps. The notion of an Algorithm is similarly generalized. We present an infinite algorithm that finds the real-number limit of an infinite sequence of rational numbers. The worst case runhing time of this algorithm is $\omega 2$. We show that this algorithm is time-optimal for the given problem.

TABLE OF **CONTENTS**

0. Introduction

In this paper we generalize the notion of a "computation" to include a process whose execution may require infinite time and infinite space. Various other notions of "infinite computaion" have been proposed in the literature. (See references [2] through [8].) We will discuss these and compare them to the model proposed here.

The intuitive idea behind our model is to consider a Turing machine with finitely many states, a finite alphabet, infinitely many non-blank symbols on its input tape, and an infinite amount of time to perform its computation. Thus the problem given to the machine can be infinitely large, and the machine can take an infinite amount of time to solve it. One consequence of the "infinite time" property will be that any given square of the machine's output tape can be written on infinitely often. This point will be discussed in detail when we give the formal definition. Our definition will then be extended to include computation by a finite sequence of such machines. Roughly speaking, the output tape of one machine will be the input tape for the next machine.

In this paper we will investigate certain aspects of the power of these machines. The motivation for this effort is twofold:

(1) Most of the theorems in the conventional theory of computation describe limitations on the power of a particular class of machines. These theorems owe their existence to the requirement that computations take a finite amount of time. We are interested in knowing what limitations, if any, are inherent in infinite machines.

Certain set-theoretic limitations are immediately obvious. For instance, let ω_1 be the smallest uncountable ordinal and let s be a function from ω_1 into the set of integers, The function s defines an uncountably long sequence s(0), s(1), ... Let S be the collection of all such sequences, Let S_0 be the sub-collection of S satisfying

$s \in S_0$ iff $\exists$ ordinal $i<\omega_1$ such that $s(i) = 0$.

Now consider any "reasonable" definition of an infinite-time Turing machine. Let T be such a maohine. If T has uncountably many squares on its input tape then a sequence s, from S, can be written in its entirety on the input tape of T. In order for T to solve even the simplest problem such as

given s on the input tape of T, with $s \in S$, determine whether $s \in S_0$

T must be allowed to perform uncountably many steps. Otherwise, T cannot solve the problem. Although this observation identifies one of the limitations of infinite Turing machines, it is by no means a profound observation. The observation depends on the set-theoretic properties of T rather than on the computational properties of T.

When we undertake to define the notion of an infinite-time computation, our immediate concern is of the following sort: A machine that can perform infinitely many steps is surely very powerful. Is it possible that the only limitations on the power of such a machine are set-theoretic in nature? If so, our definition may be doomed to triviality.

(2) There are processes that arise naturally in mathematics which require an infinite amount of time to execute. These processes would be considered "algorithms" but for the lack of finiteness. Here are a few examples:

Let f be a function defined on the real interval [0,1]. Assume that the value of f on any argument can be computed in finite time. (This assumption is neither reasonable nor necessary. It will, however, make the technicalities of our example easier to digest. The interested reader can generalize.) We can find the integral of f, from 0 to 1, by taking successive approximations. Let s(1) be f(0). Then

s(1) approximates the integral of f(x) from 0 to 1, by dividing the area under the curve into one big pieoe. Let s(2) be (0.5)·f(0) + (0.5)·f(0.5), a better approximation obtained by dividing the area into two pieces. Generalize this for arbitrary s(i), where i is a positive integer. With a finite algorithm, we can compute arbitrarily good approximations for the value of the integral of *f*. The actual value of the integral is the limit of the sequence s(1), s(2), ... If R is an infinite-time algorithm that aomputes s(1), s(2), ..., s(i), ... for all positive intsgers i, then R may, after infinitely many steps, yield the value of the integral. of f. In Section, 2 we will see how to define an infinite algorithm to compute the limit of an infinite sequence, Given that, the design of algorithm R is straightforward.

One can "construct" the algebraic numbers using an infinite process. Let Q be the set of rational numbers. Let Q[x] be the set of polynomials in x over Q. Choose p(x) in Q[x]. Let Q_1 be Q[x]/<p(x)>, where <p(x)> is the ideal of Q[x] generated by p(x). Choose q(x) in Q_1 and let Q_2 be Q_1[x]/<q(x)>, and so on. Using, a canonical homomorphism, we can consider Q_i to be a subset of Q_{i+1}, for each positive integer i. Let A be an infinite-time algorithm whose input is Q. At each "step" in the execution of A, the algorithm examines Q_i and replaces it with Q_{i+1}. After infinitely many such steps, algorithm A produces the set $\cup_i Q_i$, which

is precisely the set of algebraic numbers.

Notice that the input to this algorithm is an infinite set (the set of rational numbers). The output of each step of the algorithm is also an infinite set. Each step of the algorithm examines infinitely many input symbols and prints infinitely many output symbols. The algorithm A will have infinitely many steps, each taking an infinite amount of time. In Section 1 we will use dovetailing to compress many infinitely long steps into one infinitely long step.

Both of the above examples illustrate the same point: that some so-called "pure existence" definitions in mathematics are actually algorithms that require an infinite amount of time and space.

In [1] we have an infinite process which decomposes an infinitary boolean polynomial into its normal form. The input to the process is a statement in the infinitary homogeneous propositional calculus. The process goes through infinitely many iterations. At the end of each iteration the statement is rewritten in such a way that a larger sub-statement of the original statement has been converted into "normal form". In this case, "normal form" means that no atomic variable in the sub-statement occurs more than once. After infinitely many iterations, the entire statement is in normal form. In [1], as in the literature on Riemann

sums and algebraic numbers, the output of the process after infinitely many steps is understood by virtue of a particular mathematical context (i.e., the limit of an infinite sequence, the union of infinitely many sets, etc.) In this paper we give a general definition, in terms of Turing machines, to describe the output of an infinite computation.

Section 1 of this paper defines an infinite Turing machine, setting the stage for the analysis of an infinite-time computation. In this section we also relate our definitions to the work of others (references [2] through [8]).

In Section 2 we present an infinite algorithm to solve a common mathematical problem - the calculation of the limit of a sequence of rational numbers. Of course we cannot use the algorithm to "solve" the problem in the usual sense, since no real-world computer can read in infinitely many rational numbers and, after infinitely many steps, produce an output. Instead we can take the algorithm as a new definition of the limit concept. (In reviewing the standard ε-δ definition of a limit, we are tempted to believe that its creators modeled the definition after some more intuitive notion of an infinite process which makes the sequence "go to" its limit.)

In Section 3 we prove that finding the limit of a se-

quence of integers requires $\omega 2 + k$ time, where k is a finite non-negative integer, and ω is the smallest ordinal greater than any finite non-negative integer. Since the notion of an algorithm plays an important role in the proof of the result, the proof helps to quiet our concern that all limitations on the power of infinite computations are set-theoretic in nature.

1. Definitions

1.1 Augmented and infinite Turing machines

We begin with a standard definition of a deterministic on-line Turing machine.

Definition. A finite Turing machine is an 8-tuple

(States, initial-state, Input-Alphabet, Output-Alphabet,
fnext-state: X → States,
fnext-symbol: X → Output-Alphabet,
fnext-move-of-tape-in: X → {right, left),
fnext-move-of-tape-out: X → (right, left, halt}>,

where States, Input-Alphabet and Output-Alphabet are disjoint finite sets, initial-state ∈ States, and X = States x Input-Alphabet x Output-Alphabet.

Our Turing machines have two one-way infinite tapes: a read-only tape for input, and a read/write tape for scratch work and output. Thus, an instantaneous description (i.d.) of a finite Turing machine is a 5-tuple of the form

<state, tape-in, position-in, tape-out, position-out),

where state is an element of States (representing the

current state);

tape-in is a sequence of elements of the Input-Alphabet, all but finitely many of which are blank (representing the string of symbols on the input tape - since the input tape is a read-only tape, this sequence does not change at any time during the computation);

position-in is a positive integer (representing the current position of the read head on the input tape, with the leftmost square counted as position 1);

tape-out is a sequence of elements of the Output-Alphabet, all but finitely many of which are blank (representing the string of symbols currently on the output tape); and

position-out is a positive integer (representing the current position of the read/write head on the output tape, with the leftmost square counted as position 1).

At time 1, the instantaneous description of the machine is

(initial-state, tape-in, 1, sequenoe-of-blanks, 1>.

At time n+1, the machine uses the i.d. of time n, and the four functions $f_{next\text{-}state}$, $f_{next\text{-}symbol}$, $f_{next\text{-}move\text{-}of\text{-}tape\text{-}in}$, and $f_{next\text{-}move\text{-}of\text{-}tape\text{-}out}$ in

order to determine the i.d. of time n+1.

Let T be a Turing machine, and let s be a sequence of symbols. The action of T with s on its input tape is denoted T(s). If t is a positive integer, then the sequence of symbols on the output tape of T at time t is denoted tape-out$_t^{T(s)}$. Since tape-out$_t^{T(s)}$ is itself a sequence of symbols, the i^{th} entry in this sequence can be denoted tape-out$_t^{T(s)}$(i). Subscripts and/or superscripts will be omitted when the context permits. Similar notations such as state$_t^{T(s)}$ and position-out$_t^{T(s)}$ will be used when appropriate,

To define an augmented Turing machine, T_A, we modify the definition of a finite Turing mauhine in two ways:

(1) We allow tape-in^{T_A} to have infintely many non-blank symbols, and

(2) We define tape-out at time ω as follows:

If T_A never halts

then tape-out$_\omega^{T_A}$ is the sequence tape-out$_\omega^{T_A}$(1), tape-out$_\omega^{T_A}$(2), ..., where, for each positive integer i,

if there is a symbol, x, and an integer, n, such that tape-out$_t^{T_A}$(i) = x for all t≥n

then tape-out$_w^{T_A}(i)$ = x
else tape-out$_\omega^{T_A}(i)$ = "blur"

else tape-out$_\omega^{T_A}$ is undefined.

In the above definition it is assumed that "blur" is a symbol, and "blur" is not an element of the Output-Alphabet of T_A. Notice that tape-out may have infinitely many non-blank symbols at time ω.

If tape-out$_w^{T_A}$ is defined, for some value of tape-in^{T_A}, then we say that the worst case running time of T_A is ω. Otherwise the worst case running time of T_A is finite.

Intuitively, an augmented Turing machine is one whose input and running time can be of size ω. The problem given to T_A can be infinitely large, and T_A can take an infinite amount of time to solve the problem. After the entire run of T_A (at time ω), the content of a square on the output tape is non-"blur" if and only if, at some time during the run of T_A (i.e., before time ω), some symbol is written on that square and is never changed thereafter. We can think of the "blur" symbol on a square as the situation where the content of the square never "settled" on any one particular symbol. Looking at the square at time ω, one sees the fuzzy overprinting of symbols that changed infinitely often during the run of T_A and calls this a "blur". Since the "blur" symbol is not an element of Output-Alphabet^{T_A}, machine T_A

cannot decide, at any finite time during its run, to write "blur" on any square of its output tape. Thus "blur" is a very special output symbol for T_A.

The decision to have separate tapes for input and output is quite intentional. We do this in order to allow the augmented Turing machine to write an unbounded (or even infinite) amount of output without having to (a) destroy the input, or (b) take time to move input symbols to make room for the output.

Note that an augmented Turing maohine has finitely many states, a finite input alphabet, etc, One can write the instructions that prescribe the action of an augmented Turing machine on a finite amount of paper.

Example 1. There is an augmented Turing machine, H, which solves the halting problem for finite Turing machines. For any finite Turing machine, T, let tape-in^H contain the 8-tuple describing T, and a copy of tape-in^T. Let Output-Alphabet^H = Output-Alphabet^T $\cup$ {"T-halts", "T-does-not-halt"}. Machine H begins by writing "T-does-not-halt" on tape-out^H(1). Then H simulates the action of T on input tape-in^T, using squares 2,3,4,... of tape-out^H for scratch and output. Machine H never changes the symbol on tape-out^H(1) unless T halts. If T halts at time n, then H changes the content of tape-out^H(1) to

"T-halts", and, from time n+1 on, performs some harmless nonsense action (e.g., it idles in some special state). At time ω, one can determine whether or not T halted by examining $\text{tape-out}^H(1)$.

At this point we extend the definition of a Turing machine even further by forming a sequence of augmented Turing machines. Each machine in the sequence produces an output tape, which is supplied as part of the input to the next machine. For the moment let us call these two adjacent machines the predecessor machine and the successor machine. The successor machine is capable of reading the input and output symbols of its predecessor. In addition to reading symbols in the predecessor's Output-Alphabet, it reads the "blur" symbol on squares where "blur" was produced by the predecessor. Whereas "blur" was a special output symbol for the predecessor machine, we will see in the formal definition that "blur" is treated as an ordinary input symbol to the successor machine.

Definition, Let T be an augmented Turing machine. We define the union of tape-in^T and tape-out^T as the tape obtained by combining the squares of tape-in^T and tape-out^T in alternating fashion. More formally, for any positive integer i,

$$(\text{tape-in}^T \cup \text{tape-out}^T)(i) = \begin{cases} \text{tape-in}^T(i/2) & \text{if } i \text{ is even} \\ \text{tape-out}^T((i+1)/2) & \text{if } i \text{ is odd} \end{cases}$$

Definition. Let $T_1, T_2, \ldots, T_n$ be a finite sequence of augmented Turing machines satisfying

$$\text{Input-Alphabet}_{T_{i+1}} = \text{Input-Alphabet}_{T_i} \cup \text{Output-Alphabet}_{T_i} \cup \{\text{"blur"}\}$$

for each i from 1 to n-1. Then the sequence is called an infinite Turing machine. For each i from 2 to n, the initial i.d. of T_i is

$$(\text{initial-state}_{T_i}, \text{tape-in}_{T_{i-1}} \cup \text{tape-out}^{\omega}_{T_{i-1}}, 1, \text{sequence-of-blanks}, 1\rangle$$

if $\text{tape-out}^{\omega}_{T_{i-1}}$ is defined, and

$$(\text{initial-state}_{T_i}, \text{tape-in}_{T_{i-1}} \quad \text{tape-out}^{t}_{T_{i-1}}, 1, \text{sequence-of-blanks}, 1\rangle$$

if T_{i-1} halts at time t.

The idea of taking the union of tapes is somewhat con-

trary to, our notion of the action of a computation. After an augmented Turing machine has performed ω steps, some unseen agent magically copies infinitely many symbols in combining the machine's input and output tapes. We could overcome this difficulty by allowing each augmented machine to have more than one input tape. We chose not to do this in order to keep the notation as simple as possible. The main thing to keep in mind is that an augmented (or infinite) Turing machine does not destroy its input tape. The input tape is available for examination by the next machine in the sequence.

We now define a notion of "running time" for infinite machines.

Definition. Let $T_1, \ldots, T_n$ be an infinite Turing machine. Let k be the largest integer satisfying

there is a sequence of symbols on tape-in^{T_1} that forces k of the n augmented machines in $T_1, \ldots, T_n$ to have running time ω.

Then we define the worst case running time of $T_1, \ldots, T_n$ to be ωk.

Example 2. Let s be an infinite sequenoe of positive integers, and T_1 be an augmented Turing machine with

Input-Alphabet {0,1,2,...,9,"comma","blank"}. Sequence s can be encoded on the input tape of T by writing one digit per square, beginning with the leftmost square, using the comma as a separator between integers. We want T_1 to help decide whether sequence s contains infinitely many occurrences of the number 42, To do this, we make the Output-Alphabet of T_1 contain the symbols "even" and "odd". Machine T_1 begins by writing "even" on tape-out^{T_1}(1). Then it reads tape-in^{T_1}, examining the integers of s in order until it reaches a 42. At that time it changes the content of tape-out^{T_1}(1) to "odd". The machine proceeds to examine sequence s, changing its mind about the number (mod 2) of occurrences of 42 whenever it encounters a 42, At time w, machine T_2 examines tape-out^{T_1}(1). If it reads "even" or "odd", it prints "finite" on its output tape. If it reads "blur", it prints "infinite".

The worst case running time of T_1, T_2 is w. Clearly this is optimal performance for the given problem.

1.2 Augmented and infinite algorithms

Waving defined infinite Turing machines, we can use Church's thesis to talk about infinite algorithms. An augmented algorithm can have infinite input, and can have a

defined output at time ω. During any finite portion of its execution, an augmented algorithm can do anything that a finite algorithm can do. An (infinite algorithm) is a finite sequence of augmented algorithms.

Our algorithms will be written in a Pascal-like language. Our goal, in using the language, will be to make the writing of algorithms as independent as possible from the details of machine operation. In the following paragraphs, we describe the ways in which our algorithm language will differ from Pascal.

We will omit the explicit use of the Pascal tokens "begin" and "end". Instead we will use informative indentation. The end of a line, rather than a semicolon, will be used as the statement separator. We will often replace the exact syntax of Pascal statements with statements written in "rigorous English".

In addition to the standard Pascal constructions, we define

```
For i:=1 toward ω
    S
```

to mean

```
Let i:= 0
While i <ω do
    i:=i+1
    S
```

where S is a statement, or sequence of statements.

We also enhance standard Pascal with the following construction for dovetailing:

Let k be any positive integer, Let $S_1, S_2, \ldots$ be statements, or sequences of statements. Let each of $S_1, S_2, \ldots$ take ωk time to execute. Assume that successful execution of any of $S_1, S_2, \ldots$ does mot depend on the execution or results of any of the others. Then let the notation

```
Dovetail
   [S1]
   [S2]
    .
    .
    .
End-dovetail
```

have the obvious meaning. Since $S_1, S_2, \ldots$ each take ωk time, the whole dovetailed computation takes ωk time to execute.

The Pascal token "var" will be used to declare all variable names and their types. The values of all variables will be read from an input tape, or read from, and printed on, an output tape. For instance, the declaration

```
var v: digit
```

will mean that the effect of an assignment statement, such as

```
Let v:= 0
```

is to write the digit symbol "0" on that square of the output tape which is reserved for the value of v. The effect of a statement such as

```
Let v:= x + y
```

is to read the values of x and y from an input or output tape (wherever they are recorded) and write their sum on the square of the output tape reserved for the value of v.

We will use abbreviations such as

var $\{v_i\}_{i=1}^{\omega}$: sequence of digits

and

var r: infinite decimal expansion.

These two declarations are equivalent, since an infinite decimal expansion is an infinite sequence of digits. The only differenae is that in the first declaration, the entries of the sequence are named separately, while in the second deolaration, the sequence itself is named. Either declaration can be taken to represent a real number between 0 and 1, To represent a real number in any particular algorithm, we will use either of the above methods, whichever is notationally more convenient,

The multiple declaration

var $\{v_i\}_{i=1}^{\omega}$: sequence of digits
r: infinite decimal expansion

saves space on the tape for two infinite sequences, By referring to the definition of an augmented Turing machine, we see that the length of the output tape is ω. In order to fit two infinite sequences on this tape, we have to print the digits of the sequences in alternating fashion. For instance, we can reserve even numbered spaces for the digits of r, and odd numbered spaces for the v_i digits.

As an example of the use of our notation for infinite algorithms, we consider the following:

```
'var r: infinite decimal expansion
        .
        .
        .
 For i:=1 toward ∞
    let r:= ...(some expression)...
        .
        .
        .
```

The variable r represents a real number whose digits are re-computed each time through the loop. Let 1^{st}-digit(r) be the digit of r which is immediately to the right of the decimal point, That digit occupies a square on the output tape. If the digit-symbol on that square changes only finitely many times during the execution of the loop, then at time w, after the loop has been executed in its entirety, 1^{st}-digit(r) is a certain digit. If, however, the content of the square for 1^{st}-digit(r) changes infinitely often during the execution of the loop, then that square has "blur" in it at time ω. Therefore, at time ω, an output tape variable, or a part of an output tape variable, can have the value "blur", regardless of the manner in which it was originally declared.

As in the language Pascal we will use subprograms to divide our algorithms into their logical components. A subprogram is either a Funation, which returns the value of a specific variable, or a Procedure, which does not necessarily return the value of a specific variable. 'A typical subprogram will be presented in the following form:

var ...{variables used in Function F that are declared in the program which calls F}

Function F(...{argument list)...):,,.{type of the result)

var ...{variables which are local to Function F}

.

.

.

{executable statements in the body of Function F}

.

.

.

exit from F.

In the use of subprograms, the reader is warned of the following two facts:

(1) there is not necessarily a one-to-one correspondence

between subprograms and augmented Turing machines, and

(2) there is not necessarily a one-to-one correspondence between var declarations and augmented Turing machine tapes.

These points are illustrated by the next example.

Example 3. Consider two procedures, P and Q, written as follows:

Procedure P

```
var ip,jp:integer
      .
      .
for ip:=1 toward w
      .
      .
for jp:=1 toward ω
      .
      .
```

Procedure Q

```
var iQ,jQ:integer
      .
      .
for iQ:=1 toward ω
      .
      .
for jQ:=1 toward ω
      .
      .
```

These two procedures may be called simultaneously by a program as follows:

```
Program MAIN
  Dovetail
    [call P]
    [call Q]
  End-dovetail
```

Execution of Program MAIN will take place in two parts, each part taking ω time. In the first part, the "For i_P" loop of Procedure P and the "For i_Q" loop of Procedure Q are executed simultaneously. In the second part, the "For j_P" loop of Procedure P and the "For j_Q" loop of Procedure Q are executed simultaneously. Implementation of Program MAIN by augmented Turing machines will involve one augmented machine to execute the "For i_P" and "For i_Q" loops, and another augmented machine to execute the "For j_P" and "For j_Q" loops. So the augmented machines do not correspond to Procedures P and Q. Let us call these machines T_i and T_j.

Let us look at Procedure P in slightly more detail:

```
Procedure P
  var ip,jp:integer
      a :integer
      .
      .
  for ip:=1 toward ω
    a:= ...
```

```
        .
        .
for jp:=1 toward ω
   a:= a + ...
        .
        .
```

The variable a is declared only once, but its values are recorded at different times on at least three tapes:

(1) the output tape of T_i,
(2) the input tape of T_j, and
(3) the output tape of T_j.

Depending on our conventions, we may choose to consider items (1) and (2) to be the same tape. Notice, however, that one var declaration defines part of the content of more than one tape.

In spite of the long list of notational conventions, there is a small number of principles which, taken together, describe our notion of an infinite algorthm. These principles can be stated as follows:

An infinite algorithm

(1) has a finite input alphabet, but may receive as its

input, an infinite sequence of input symbols;

(2) has a finite output alphabet, but may write, as its output, an infinite sequence of output symbols;

(3) may perform infinitely many steps in order to complete the computation; and

(4) can be described by writing a finite set of instructions.

As a footnote to item (2), we add that an infinite algorithm is composed of a finite sequence of augmented algorithms, and that each augmented algorithm has a special output symbol "blur", which it may not write during any finite step in the course of its execution.

1.3 Notes on the literature

The literature on finite-state machines contains many examples of attempts to extend the definition of "computation" in an infinite way. One such effort appears in [2]. In that work, Muller defines an infinite sequential machine as a modification of the deterministic finite automaton. Muller's machine is like the finite model in all respects except one; namely, that the usual set, P, of accepting

states is replaced by a collection, F, of sets of states. Muller's machine, M, has finitely many states, so if we give it an infinite sequence of symbols (an infinite input string x), it will enter certain of its states infinitely often. Let Inf(x,M) be the collection of states entered infinitely often upon the input of x to M. The string x is accepted by M iff Inf(x,M) ϵ F.

Muller's infinite sequential machine is strictly weaker than our infinite Turing machine. To see this, consider the following example:

Example 4. Let s be an infinite sequence of positive integers. Following our notational conventions, we let s(1) be the first kntry in s. We want to determine whether s(1) occurs infinitely often in s. We call this the "s(1)-problem".

It is easy to define an infinite Turing machine to solve the s(1)-problem. To do this, modify T_1 of Example 2 by having it compare each integer of s with s(1) rather than 42. No other modification is needed. (Note: We may choose to compare each s(i) to s(1) by having the input tape head of T_1 move back and forth between s(1) and successive entries s(i). Since s(1) is the leftmost integer on the tape, the amount of tape movement increases as i becomes large. To avoid that backtracking on the input tape, we can

begin the computation by writing a copy of s(1) on the output tape. Then we simply compare each s(i) on the input tape to our copy of s(1) on the output tape. Although this new scheme seems more efficient, it will result in no time savings on the infinite scale. The computation will take ω steps either way we do it.).

There is no Muller infinite sequential machine which can solve the s(1)-problem. To see this, let M be an infinite machine in the Muller sense. Let s be the infinite sequence <s(1), s(1), s(1), ...>. If M solves the s(1)-problem, then M accepts sequence s, so Inf(s,M) $\in$ F. Let M be in state q after reading the first s(1). Machine M has finitely many states, but s(1), the first entry in s, can be any one of infinitely many distinct positive integers. Thus there is an integer s(1)'$\neq$s(1) such that M would be in state q after reading s(1)'. Let s' be the infinite sequence <s(1)', s(1), s(1), s(1), ...,>. Since s(1)' occurs only once in s', M should not accept s'. But M is in state q after reading s(1)'. Thereafter, M behaves exactly as if it were reading sequence s. Therefore M accepts s'. This is a contradiction. □

In [3] Buchi defines an infinite sequential machine which is shown by McNaughton C43 to be equivalent to Muller's model. In [5] Rabin uses the Buchi automaton to prove the decidability of the second-order theory of two

successor functions (S2S). Rabin begins by extending Buchi's definition to that of an automaton which accepts an infinite binary tree. (We call this a Rabin automaton.) He then shows that

(1) For the class of Rabin automata, the emptiness problem (Given automaton R, is there a tree that is accepted by R?) is decidable; and

(2) For every formula, F, of S2S, there is a Rabin automaton, R, such that F is true iff there is an infinite tree which is accepted by R.

We proceed to show that (1) does not hold for the class of augmented Turing machines. In the theorem and its proof, "decidable" and "undecidable" have their usual meanings, namely, the existence or non-existence of a finite Turing machine to solve the given problem.

Theorem 1. The emptiness problem for augmented Turing machines is undecidable.

Proof. We define an effective procedure that turns the description of a finite Turing machine, T, into that of an augmented Turing machine H_T. Given T, let H_T begin by writing "does-not-accept" on its output tape. Then for x, a finite string of symbols which we intended to give to T

as input, H_T simulates the action of T on x. If T halts in an accepting state then H_T changes "does-not-accept" to "accepts", and idles until time ω. If T halts in a non-accepting state, then H_T idles without changing the symbol "does-not-accept". If T never halts, then H_T simply simulates T until time ω.

Clearly H_T accepts x if and only if T accepts x. Furthermore H_T is an augmented Turing machine. Even though the action of an augmented machine is infinite, the description of such a machine is always finite. (I.e., an augmented Turing machine has finitely many states; a finite input alphabet, etc. One can write the instructions that describe an augmented algorithm on a finite amount of paper.) It is intuitively clear that the transformation from T to H_T is recursive.

Given T and x as above, let $M_{T,x}$ be the augmented Turing machine which, on any input string w, ignores w and simulates the action of H_T on x. Machine T accepts sequence x if and only if $M_{T,x}$ acoepts any (i.e., all) input sequences. Thus, if the emptiness problem for augmented Turing machines is decidable, we can also effectively decide, for any finite Turing machine T and any finite Sequence x, whether T accepts x, Since this is known to be impossible, we have proved the theorem. □

Thus Rabin's clause (1), given above, does not hold when modified to apply to infinite Turing machines. In fact, the opposite holds:

(1)' For the class of augmented Turing machines, the emptiness problem is undecidable.

Having modified Rabin's clause (1) in such a manner, one would suspect that we can modify his clause (2) to prove that a particular theory is undecidable. Let K be a theory (perhaps a second-order theory). In order to use Rabin's technique to prove that K is undecidable, we need to prove the following "reversed" version of Rabin's clause (2):

(2)' For every augmented Turing machine, H, there is a formula, F, of K such that F is true iff there is an input sequence which is accepted by H.

The difficulty in using this information arises in finding the theory K, Rabin makes infinite tree automata correspond to the theory S2S by demonstrating that acceptance of a tree by an automaton is definable in S2S. The set, over which we quantify in the second order theory S2S, corresponds to a set of nodes on the infinite tree. This is the set of nodes that the automaton may visit while being in a particular state. (By "may visit" we mean the following: Automaton A may visit node n in state q iff, during a run of A in which

A accepts the tree, A is in state q when it visits node n.) . An automaton visits each node of its input tree only once. However, an augmented Turing machine may visit each of its input tape squares more than once. The definition of acceptance by an augmented Turing machine is thus more complicated. This is where Rabin's methods fail to help us find the appropriate theory K.

Recently Bavel [C63] has attempted to expand automata theory to include machines with infinitely many states. Bavel's machine differs from an augmented Turing machine (and from Muller's infinite sequential machine) in several ways, the most important of which is that Bavel's machine does not accept infinite sequences. Bavel's machine accepts only finite sequences, so the definition of acceptance is the usual one. (I.e., there is a set of accepting states, and a machine must land in one of these states.) Since there are infinitely many states, the machine can remember an indefinitely large amount of information. The states perform the same role as the infinite tape of a finite Turing machine. Bavel's most powerful machine is, in fact, equivalent in power to a finite Turing machine.

In this paper we attempt to capture the full notion of "computation on an infinite sequence", so we extend the Turing machine model rather than the finite automaton model. Similar attempts to define computation on infinite sequences

have arisen as extensions of the theory of recursive functions. In Rogers "71 and Kleene E81 we see the extension of recursive function theory to include recursive functionals. An augmented Turing machine is a device whose input and output may both be infinite sequences of integers, An infinite sequence of integers is a function. Thus an augmented Turing machine is a functional (a mapping from functions to functions). In [7] the theory of recursive functions is extended to functionals with the help of oracles. In C81 the theory is extended by defining additional schema, Using either method, the functionals which are considered to be "computable" are all "continuous". I.e., in connection with either definition is a theorem of the following sort: Let F be a computable functional, Then for every function, f, there is a non-negative integer, n, such that

$$(\forall g)[(\forall m<n)(f(m) = g(m)) \Rightarrow F(f) = F(g)].$$

Thus the value of F on f depends only on some finite part of f. (In Rogers, continuity is proved in Corollary XXI on Page 353. The proof is given in terms of the Baire topology, but it is a straightforward consequence of Rogers' definition of "functional". The definition states that, in order to produce a value, functional F can consult the oracle for function f only finitely many times.) This continuity property does not hold for functionals defined by augmented Turing machines, as we have shown in Example 2.

2. The Limit of a Sequence of Rational Numbers

2.1 Definition of the problem

Keeping in mind our definition of an infinite Turing machine, we will now examine an interesting concrete example. The example will give us some sense of the power of an infinite machine. The worst case running time for this machine will be $\omega 2$. So the example will force us to pose the question: Can the same problem be solved by an infinite machine whose worst case running time is ω? At the end of Section 3 we will show that such a machine does not exist. Thus, we will have a counterexample to dispute the hypothesis that "any computation that can be done in $\omega(n+1)$ time can be done in ωn time".

The problem that we have chosen to solve uith an infinite machine is taken :Prom pure and applied mathematics. It lends itself very naturally to solution by an infinite computation.

Problem 1. Given an infinite sequence, **s**, of rational numbers, determine whether **s** converges, and if it does, find the real number to which it converges.

Notice that no assumptions have been made about the regularity of the infinite sequence s, or the regularity of

the decimal expansion of the real number to which s converges. No observable patterns are assumed to occur in either of these sequences. Thus we do not assume the existence of a finite description of sequence s or of its real-number limit. In contrast to this, several authors have attempted to define the notion of a recursive real number. The work in this field includes papers by Rice [9], Mostowski [10], Lachlan [11] and others. Rice describes a recursive real number intuitively as "one for which we can effectively generate as long a decimal expansion as we wish". First choose a recursive function from integers to pairs of integers, so as to obtain a recursive indexing of the rational numbers. A sequence of rationals is recursively enumerable if its corresponding sequence of integers is <g(0), g(1), g(2), ...> for some recursive function g. An r.e. sequence <a(0), a(1), ...> of rational numbers is recursively convergent if there is a recursive function, h, to decide, for each n, how far along on the sequence we must look in order to be assured that

$$|a(i) - a(j)| < 1/n.$$

Rice proves the following: There is no finite Turing machine to determine whether the limit of a recursively enumerable, recursively convergent sequence of rational numbers is equal to zero. We will see in this section that there is an infinite Turing machine to solve this problem for arbi-

trary sequences of rationals, using the conventional analytic definition of convergence. We will see in Section 3 that the problem cannot be solved by an augmented Turing machine.

2.2 Solution of the problem - intuitive..explanation

To simplify the notation we assume that all rationals and reals lie in the cltosed interval [0,1]. Any number in that interval has a decimal expansion in which the most significant digit is to the right of the decimal point. None of the conceptual flavor of the example is lost in making this assumption.

We will construct an infinite algorithm to solve Problem 1. The algorithm will be called "Program PROBLEM1". The "top-down" plan of Program PROBLEM1 is as follows:

(a) The input to Program PROBLEMI is an infinite sequence $\{x_i/y_i\}_{i=1}^{\omega}$ of rational numbers, Each rational number x_i/y_i will be given as a pair $\langle x_i, y_i \rangle$ of non-negative integers, with $y_i > x_i \geq 0$. These integers can be encoded on an input tape using their conventional decimal (base 10) representations. They appear on the tape in the following order:

$$x_1, y_1, x_2, y_2, x_3, y_3, \ldots \text{ etc.}$$

A "comma" symbol acts as separator between two integers,

(b) Let a finite string of symbols be called a finite decimal expansion if its leftmost symbol is a decimal point, and all of its other symbols are decimal digits (0 through 9). Each finite decimal expansion represents a rational number. When we write that "q id a finite decimal expansion" we will often refer also to "the rational number q". We assume that the set of finite decimal expansions is linearly ordered by a recursive indexing (e.g., lexicographical ordering). Thus, for each positive integer i, we can refer to the i^{th} finite decimal expansion.

(c) Here's how Program PROBLEM1 works: Let i be a positive integer. At step i in the computation, find $r_1, r_2, \ldots r_i$, the i most significant digits in the decimal expansion of x_i/y_i. Write these digits on the output tape. Also, for each j between 1 and i, compute the difference between the decimal number $0.r_1r_2\ldots r_i$ and the j^{th} finite decimal expansion q_j. Call this difference $diff_j$. Write these differences on the output tape. Thus at each step, i, the algorithm writes new values of digits $r_1, \ldots, r_i$, printing them Over the old values. The algorithm also writes new values for the decimal differences $diff_1, \ldots, diff_i$, printing them over the values calculated in the previous step.

Consider the digit r_1. It is possible that, at some step in the computation, r_1 takes on a certain digit value, R_1, and never changes again. In this case, the value of r_1 at time ω is the digit R_1. If this happens with each of the digits $r_1, r_2, \ldots$, then at time ω we have an infinite decimal expansion $0.R_1R_2\ldots$ to which $\{x_i/y_i\}_{i=1}^{\omega}$ converges. We might conclude that our algorithm had taken only ω steps to complete its mission. But there is one very important detail remaining: beginning at time ω the algorithm must check each digit $r_1, r_2, r_3, \ldots$ to verify that the value of each did indeed settle, It can take another ω steps to do this checking.

If, for some positive integer i, r_i did not settle, then it will only take finitely many steps to discover this. A good example of this occurs when $\{x_i/y_i\}_{i=1}^{\omega}$ converges to 0.55 by oscillating around it:

$x_1/y_1 = 0.550111\ldots$ (infinitely many 1's)
$x_2/y_2 = 0.549888\ldots$ (infinitely many 8's)
$x_3/y_3 = 0.550011\ldots$
$x_4/y_4 = 0.549988\ldots$

.
.
.

etc.

At time ω, there is a "blur" in the square for r_2. The algorithm finds this "blur" very quickly, i.e., only finitely many steps after time ω. We will give the algorithm all the technical equipment it needs to deduce that, in this case, $\{x_i/y_i\}_{i=1}^{\omega}$ may converge to 0.55. We emphasize the word "may" because the "blur" in place of r_2 is not conclusive evidence that the sequence does or does not converge to 0.55. We will see in the formal treatment that the algorithm can determine that the digit r_2 oscillated between 4 and 5. After making this determination, the algorithm will have to verify that $\{xi/yi\}_{i=1}^{\omega}$ converges, by finding the value of j such that 0.55 is the j^{th} finite decimal expansion, and checking that diffj is equal to zero.

The difficulty which remains is that diffj now has infinitely many digits, since the algorithm computed i digits of diffj at each step i. So checking diffj for equality to zero means checking the infinite number of digits in diffj.

(d) Program PROBLEM? will be presented rigorously as a dovetailed combination of a subprogram called Procedure INFINITE, and several copies of another subprogram called Function FINITE, Each of these subprograms has two main loops. The first loop takes ω time, and the second loop takes up to ω time, In each case, the first loop creates an infinite sequence of ;symbols, and the second loop checks

that infinite sequence for occurrences of the "blur" symbol, Function FINITE answers "true" or "false" to whether $\{x_i/y_i\}_{i=1}^{\omega}$ converges to q, a particular finite decimal expansion. Procedure INFINITE tries to create an infinite decimal expansion to which $\{x_i/y_i\}_{i=1}^{\omega}$ may converge.

The intuitive ideas behind the subprograms FINITE and INFINITE are now given in more detail:

(e) The input to Function FINITE is an infinite sequence $\{x_i/y_i\}_{i=1}^{\omega}$ of rational numbers, and a number, q, with a finite decimal expansion. The output of the algorithm is the value "true" if $\{x_i/y_i\}_{i=1}^{\omega}$ converges to q, and "false" otherwise,

Let i be a positive integer. At step i in the computation, Function FINITE finds the i most significant digits of the difference between x_i/y_i and q. If the function is executed by an infinite Turing machine, the most significant digit of this difference can always be written on the leftmost square of the machine's output tape, Likewise, the k^{th} most significant digit of the difference can always be written on the k^{th} square of the tape. If $\{x_i/y_i\}_{i=1}^{\omega}$ converges to q, the differences thus computed will get smaller and smaller as i becomes :Large. It is intuitively sensible then that each digit on the output tape will settle into being zero, This, sensible claim will be made rigorous by Lemma

2.1. Thus, at step $\omega + i$ in the execution of Function FINITE, the function checks to see if the i^{th} square on its output tape contains a zero. If the function ever finds a non-zero symbol, it concludes that $\{x_i/y_i\}_{i=1}^{w}$ does not converge to q. Otherwise it concludes, after checking each of the ω-many digits, that $\{x_i/y_i\}_{i=1}^{\omega}$ does converge to q.

(f) The input to Procedure INFINITE is an infinite sequence, $\{x_i/y_i\}_{i=1}^{\omega}$, of rational numbers. The output of Procedure INFINITE is in two parts. The first part is an infinite sequence of symbols (which we will call $r_1, r_2, r_3, \ldots$), each of which is either a decimal digit or the "blur" symbol. The second part is the smallest value of k such that r_k = "blur" (if such a value exists).

Procedure INFINITE will attempt to record a succession of better and better approximations (i.e., more and more digits) for the limit; of $\{x_i/y_i\}_{i=1}^{\omega}$, assuming that this limit has an infinite decimal expansion. Beginning at time ω, Procedure INFINITE will scan the infinite expansion that it has created, checking to see if any of its symbols turned out to be "blur". If not, then the program takes this infinite decimal expansion to be the limit of $\{x_i/y_i\}_{i=1}^{\omega}$. If Procedure INFINITE finds a "blur", and if it is determined that the "blur" represents oscillation between two consecutive digits (say 4 and 5), then Program PROBLEM1 consults the output of an appropriate run of Function FINITE-

We give a few examples to illustrate how Procedure INFINITE works:

Example 1. Let $x_1, y_1, x_2, y_2, x_3, \ldots$ be given as input to Procedure INFINITII. Assume that the decimal expansions of $x_1/y_1, x_2/y_2, \ldots$ are

$$x_1/y_1 = 0.5555\ldots \text{ (infinitely many 5's)}$$
$$x_2/y_2 = 0.1555\ldots$$
$$x_3/y_3 = 0.1155\ldots$$

.
.
.

and so on. The procedure begins by writing the following on its output tape:

5	blank	blank	blank	...
square 1	square 2	square 3	square 4	...

This represents the fact that 5 is the most significant digit in the decimal expansion of x_1/y_1. Then procedure INFINITE writes

1	5	blank	blank	...
square 1	square 2	square 3	square 4	...

This represents the two most significant digits in the decimal expansion of x_2/y_2. Next, the procedure writes

1	1	5	blank ...
square 1	square 2	square 3	square 4 ...

And so on. By referring to the definition of the output of an augmented Turing machine, we see the output of Procedure **INFINITE** at time ω will be

1	1	1	1 ...
square 1	square 2	square 3	square 4 ...

From time ω on, the procedure will check each symbol on the output tape to make sure that none of the squares contains the "blur" symbol. This checking can take ω time, so Procedure **INFINITE** has worst case running time $\omega 2$. Notice how the content of the output tape corresponds to the limit of the sequence $\{x_i/y_i\}_{i=1}^{\omega}$, which is 0.1111... .

Example 2. Let

$x_1/y_1 = 0.14888...$ (infinitely many 8's)
$x_2/y_2 = 0.15000...$ (infinitely many 0's)
$x_3/y_3 = 0.14988...$
$x_4/y_4 = 0.15000...$

$x_5/y_5 = 0.14998...$

.
.
.

and so on. Procedure INFINITE begins by writing the following on its output tape:

1	blank	blank	blank	blank ...
square 1	square 2	square 3	square 4	square 5 ...

Then it writes

1	5	blank	blank	blank ...
square 1	square 2	square 3	square 4	square 5 ...

Then

1	4	9	blank	blank ...
square 1	square 2	square 3	square 4	square 5 ...

Then

1	5	0	0	blank ...
square I	square 2	square 3	square 4	square 5 ...

Then

1	4	9	9	8	...
square 1	square 2	square 3	square 4	square 5	...

And so on. Again by referring to the definition of the output of an augmented Turing machine, we see that the output of Procedure INFINITE at time ω will be

1	"blur"	"blur"	"blur"	"blur"	...
square 1	square 2	square 3	square 4	square 5	...

At time ω, the Procedure checks symbols on this output tape until it finds the "blur" symbol in square 2. It records this number 2. In the rigorous definition of Procedure INFINITE, we will denote this number by the letter "$\tilde{k}$". Notice that the output tape of the procedure contains the string <1, "blur", "blur", ...> but the sequence $\{x_i/y_i\}_{i=1}^{\omega}$ converges to 0.15. Procedure INFINITE uses a trick with even and odd digits to determine that the content of the $\tilde{k}^{th}$ square has been oscillating between 4 and 5. In this way INFINITE obtains the digit "5" in 0.15. In the proof of the correctness of Procedure INFINITE it will be shown that the trick always produces the decimal expansion to which $\{x_i/y_i\}_{i=1}^{\omega}$ is "most likely" to converge. I.e., Procedure INFINITE returns the finite decimal expansion 0.15, along with the claim that if $\{x_i/y_i\}_{i=1}^{\omega}$ converges, then it conver-

ges to 0.15. So $\{x_i/y_i\}_{i=1}^{\omega}$ either converges to 0.15, or it does not converge. The convergence of $\{x_i/y_i\}_{i=1}^{\omega}$ is then tested by examining the output of that particular run of Function FINITE whose input is $\{x_i/y_i\}_{i=1}^{\omega}$ along with the finite decimal expansion 0.15, For the sequence given in this example, Function FINITE returns the value "true", indicating that $\{x_i/y_i\}_{i=1}^{\omega}$ converges to 0.15.

Example 3. Let

$x_1/y_1 = 0.14444...$ (infinitely many 4's)
$x_2/y_2 = 0.15555...$ (infinitely many 5's)
$x_3/y_3 = 0.14444...$
$x_4/y_4 = 0.15555...$
$x_5/y_5 = 0.14444...$

.
.
.

and so on. As in Example 2, the output tape of Procedure INFINITE will contain <1,"blur","blur",...> at time ω. Furthermore, Procedure INFINITE will note that the content of square 2 oscillated between 4 and 5. So, just as it did in Example 2, Procedure INFINITE will return the finite decimal expansion 0.15, with the implicit claim that if $\{x_i/y_i\}_{i=1}^{\omega}$ converges, then it converges to 0.15. But unlike

the sequence of Example 2, the sequence of this example does not converge. This demonstrates the need for the part of the algorithm which checks the convergence of $\{x_i/y_i\}_{i=1}^{\omega}$ to 0.15 by examining the output of the appropriate run of Function **FINITE.**

2.3 Solution of the problem - technical details

We may now procede with some of the technical details.

Given real number,, r, and positive integer, i, let i^{th}-digit(r) be the i^{th} digit in the decimal expansion of r. Thus

$$r = 1^{st}\text{-digit}(r)\cdot 10^{-1} + 2^{nd}\text{-digit}(r)\cdot 10^{-2} + \ldots .$$

We use the notation $\text{truncation}_i(r)$ to denote

$$1^{st}\text{-digit}(r)\cdot 10^{-1}+\ldots+i^{th}\text{-digit}(r)\cdot 10^{-i}.$$

Let $\{x_i\}_{i=1}^{\omega}$, $\{y_i\}_{i=1}^{\omega}$ be sequences of integers. Since we want to compute the rational x_i/y_i in [0,1], we insist that $0 \le x_i < y_i$, for all i. We would like a machine whose input tape has the infinite sequence

$x_1, y_1, x_2, y_2, x_3, y_3, \ldots$

and whose output tape has either (1) the symbol "does-not-converge", or (2) the symbol "converges" and the decimal expansion of the number r, where

$$r = \lim_{i \to \omega} x_i/y_i.$$

Note that, for every positive integer i, $\text{truncation}_i(x_i/y_i)$ represents the i most significant digits of the i^{th} member of the sequence $\{x_i/y_i\}_{i=1}^{\omega}$.

Function **FINITE** is the algorithm which determines if the sequence $\{x_i/y_i\}_{i=1}^{\omega}$ of rationals converges to q, a number with a finite decimal expansion. In the algorithm, a variable called diff is assigned a new value each time through a "For" loop. After ω iterations of the loop, we want to be able to identify an outcome (possibly "blur") for each individual digit in the decimal expansion of diff. Thus, for each positive integer k, we must write k^{th}-digit(diff) in the same place (an the same square of the output tape) each time through the loop. We can now present Function **FINITE.**

var $\{x_i\}_{i=1}^{\omega}$, $\{y_i\}_{i=1}^{\omega}$: sequences of integers
Function **FINITE** (q : finite decimal expansion) : boolean;
var diff: infinite decimal expansion {= real number}

```
    i,k : integer
For i:= 1 toward ω
    let diff:= |truncation_i(x_i/y_i) - q|
Let FINITE:= true {a tentative conclusion -
                   that {x_i/y_i}_{i=1}^{ω} converges to q}
For k:= 1 toward ω
    if k^th-digit(diff) ≠ 0 then let FINITE:= false
                                 exit from FINITE
exit from FINITE
```

The execution of Function FINITE has two distinct loops. Each iteration of the first loop computes a new value for diff, writing it over the previous value for diff. This value diff represents the difference between x_i/y_i and q (truncated to i digits). If we narrow our attention to the square on the output tape where the most significant digit of diff is being written, we see a new value on that square for each iteration of the loop. If we knew for a fact that $\{x_i/y_i\}_{i=1}^{\omega}$ converged to q, we would expect the symbol in that square to eventually settle into being zero, and never change again. Stated more precisely, we would expect that

$$\lim_{i \to \omega} 1^{st}\text{-digit}(diff) = 0$$

or

$$\lim_{i \to \omega} 1^{st}\text{-digit}(|truncation_i(x_i/y_i - q)|) = 0 .$$

This expectation is given more generally in the statement of Lemma 2.1. The lemma states that if $\{x_i/y_i\}_{i=1}^{\omega}$ converges to q, then each digit of $|\text{truncation}_i(x_i/y_i) - q|$ will settle into being zero.

Lemma 2.1. Let q be a number with a finite decimal expansion. Let $\lim_{i\to\omega} x_i/y_i = q$. Then for each positive integer k,

$$\lim_{i\to\omega} k^{\text{th}}\text{-digit}(|\text{truncation}_i(x_i/y_i) - q|) = 0.$$

Proof. Assume the contrary - that there is an integer, k, such that $\lim_{i\to\omega} k^{\text{th}}\text{-digit}(|\text{truncation}_i(x_i/y_i) - q|) \neq 0$. For any positive integer i, $\text{truncation}_i(x_i/y_i) - q$ is the difference of two finite decimal expansions, so $k^{\text{th}}\text{-digit}(|\text{truncation}_i(x_i/y_i) - q|)$ is a decimal digit. Thus the claim that $k^{\text{th}}\text{-digit}(|\text{truncation}_i(x_i/y_i) - q|)$ does not converge to 0, for some particular positive integer k, can be written rigorously as

$$(\forall i_0)(\exists i>i_0)[k^{\text{th}}\text{-digit}(|\text{truncation}_i(x_i/y_i) - q|)\geq 1].$$

But $k^{\text{th}}\text{-digit}(|\text{truncation}_i(x_i/y_i) - q|)\geq 1$ implies that

$$|\text{truncation}_i(x_i/y_i) - q|\geq 10^{-k}.$$

Eventually i is greater than the largest power of 10 in the decimal expansion for q, at which point one obtains the same result by either truncating x_i/y_i and subtracting q, or by subtracting q from x_i/y_i and then truncating. Stated formally, for sufficiently large i we have

$$|\text{truncation}_i(x_i/y_i) - q| = |\text{truncation}_i(x_i/y_i - q)|$$

$$= \text{truncation}_i(|x_i/y_i - q|) .$$

Thus $\text{truncation}_i(|x_i/y_i - q|) \geq 10^{-k}$. Since $|x_i/y_i - q|$ is non-negative,

$|x_i/y_i - q| \geq \text{truncation}_i(|x_i/y_i - q|)$, so

$|x_i/y_i - q| \geq 10^{-k}$.

Summarizing, we have found an integer, k, satisfying

$$(\forall i_0)(\exists i > i_0)[|x_i/y_i - q| \geq 10^{-k}].$$

This contradicts the assumption that $\lim_{i \to \omega} x_i/y_i = q$. □

Proposition 2.1. Function FINITE is correct; i.e., for an infinite sequence $\{x_i/y_i\}_{i=1}^{\omega}$ and finite decimal expansion q, the value of FINITE(q) is "true" if and only if $\{x_i/y_i\}_{i=1}^{\omega}$

converges to q.

Proof. If $\{x_i/y_i\}_{i=1}^{\omega}$ converges to q, then, by Lemma 2.1, the limit (as i goes to ω) of k^{th}-digit(|truncation$_i$(x_i/y_i) - q|) is 0, for every positive integer k. Thus, for any positive integer k, after some finite number of iterations of the "For i" loop of Function FINITE, the value of k^{th}-digit(|truncation$_i$(x_i/y_i) - q|) settles into being 0. Thus, at time w, k^{th}-digit(diff) is equal to 0. Notice that the second loop of Function FINITE simply checks each digit of diff, If it finds all digits to be 0, it reports "true". Thus, if $\{x_i/y_i\}_{i=1}^{\omega}$ converges to q, the value of Function FINITE(q) is "true".

Conversely, if Function FINITE reports "true" it is easily seen that (each digit of) diff approaches zero as i approaches w. Therefore, $\{x_i/y_i\}_{i=1}^{\omega}$ converges to q. □

The worst case running time for Function FINITE is clearly $\omega 2$. The action of the function takes place in two Parts. The first part creates an infinite sequence of digits; the second part checks to see if each of the digits in that sequence is zero. We will see in Section 3 that it is impossible to shorten the overall worst case running time of the algorithms given in this section. We will prove that, in order to solve Problem 1, a machine must solve a simpler

problem, and this simpler problem cannot always be solved in ω time. Thus the two parts of Function FINITE cannot be combined into one loop. There is no algorithm which, in one ω-time step, computes the infinite sequence of digits which we call diff and reports how many of these digits are zero.

The intuitive idea behind the next algorithm, Procedure INFINITE, has already been discussed. We now give the algorithm in complete detail.

```
var {x_i}_{i=1}^ω, {y_i}_{i=1}^ω: sequences of integers
    {r_k}_{k=1}^ω: sequence of digits
    k̃: integer
    comment: an element of the set
             {"converges","does-not-converge","don't-know"}

Procedure INFINITE

var i,k: integer
    {e_k}_{i=1}^ω: sequence of digits
    {o_k}_{i=1}^ω: sequence of symbols, each of which
       is either a digit or the symbol "not-defined"

For i:=1 toward ω
    For k:= 1 to i
        Let r_k:= k^th_digit(x_i/y_i)
        Let e_k:= the largest even digit≤r_k
```

If $r_k \neq 0$ then let o_k:= the largest Odd digit$\leq r_k$

else let o_k:= "not-defined"

Let comment:= "converges" {a tentative conclusion}

For $\tilde{k}$:= I toward w

if $r_{\tilde{k}}$="blur"

then cases

$e_{\tilde{k}} \neq$"blur": Let $r_{\tilde{k}}$:= $e_{\tilde{k}}$+1

Let comment:= "don't-know"

$o_{\tilde{k}} \neq$"blur": Let $r_{\tilde{k}}$:= $o_{\tilde{k}}$+1 {to see why $r_{\tilde{k}}$ is a digit, see ease 4.2 of Proposition 2.2 below}

Let comment:= "don't-know"

else : Let comment:= "does-not-converge"

exit from INFINITE

exit from INFINITE

Upon each iteration of the "For i" loop, Procedure INFINITE records the value of truncation$_i(x_i/y_i)$, the i most significant digits of x_i/y_i. In the simplest case, $\{x_i/y_i\}_{i=1}^{\omega}$ converges to a real number, and, at the end of the initial loop, r_k is a digit for eaah value of k. For instance, if $\{x_i/y_i\}_{i=1}^{\omega}$ converges to 0.1, then r_k might be 1 for k = 1, and 0 otherwise. However, the decimal expansions x_i/y_i can also converge to 0.1 by oscillating around it as in the following example:

x_1/y_1 = 0.11111... (infinitely many 1's)

$x_2/y_2 = 0.08888...$ (infinitely many 8's)

$x_3/y_3 = 0.10111...$

$x_4/y_4 = 0.09888...$

$x_5/y_5 = 0.10011...$

$x_6/y_6 = 0.09988...$

$x_7/y_7 = 0.10001...$

.

.

.

In this-case, $\{x_i/y_i\}_{i=1}^{\omega}$ converges to 0.1, but at the end of the first ω steps of Procedure INFINITE, r_k is "blur" for all k. To account for this phenomenon we have Procedure INFINITE keep track not only of r_k, but also of e_k and o_k. In the example above, r_1 jumps between 0 and 1 in successive iterations of the "For i" loop. But el, the largest even digit$\leq r_1$, is always 0. Thus, at time ω, Procedure INFINITE observes that r_1 is "blur", so it examines e_1 and finds the digit 0. It then sets $r_1 := 1$ and reports the comment "don't-know". The main algorithm, (Program **PROBLEM1** given below) will interpret this information a5 a request to examine the output; of Function FINITE with input q = 0.1.

Modifying the example slightly, we get

$x_1/y_1 = 0.21111...$ (infinitely many 1's)

$x_2/y_2 = 0.18888...$ (infinitely many 8's)

$x_3/y_3 = 0.20111...$

$x_4/y_4 = 0.19888...$

$x_5/y_5 = 0.20011...$

$x_6/y_6 = 0.19988...$

$x_7/y_7 = 0.20001...$

.

.

.

In this new example, $\{x_i/y_i\}_{i=1}^{\omega}$ converges to 0.2. Once again r_1 is "blur" after ω iterations of the "For i" loop. But since e_1 oscillates between 0 and 2 during the course of the looping, e_1 is also "blur" at the end of the loop. However o_1 is 1 at the end of the loop and Procedure INFINITE sets r_1:=2 and proceeds as before.

If r_k is zero, then there is no largest odd digit$\leq r_k$, thus the use of the new symbol "not-defined". The role of this value in the test for convergence will be examined in detail later.

We must remember that, although each of these examples has an input sequence that follows a nice "recursive" pattern, no such pattern is assumed in general. Our input se-

quences may be quite disorderly. It is important to note this so that we do not make the mistake of thinking that our algorithms can take shortcuts by observing patterns in the sequences.

If we decide to generalize our problem so that all real numbers may be represented (not only real numbers in the interval [0,1]) then at time ω, Procedure INFINITE will have to know the position of the most significant digit of r that needs to be examined. Each time through the "For i" loop, the position of the most significant digit of r should be encoded on a place reserved for that purpose on the output tape. If, at time ω, the value encoded in that place has not settled (i.e., has "blur" symbols) then the sequence $\{x_i/y_i\}_{i=1}^{\omega}$ does not converge,

Notice that the second infinite loop "For $\tilde{k}$" could have been avoided if we had been able to produce, by time ω, a variable whose value is the smallest k such that r_k = "blur". This task is equivalent to the one we encountered in trying to speed up Function FINITE. As we noted earlier, and will prove in Section 3, the speed up is impossible to achieve.

The main program (Program PROBLEM1) combines Function FINITE and Procedure INFINITE by dovetailing.

We now formalize the main program, Program PROBLEM1. In the program, we assume that the set of finite decimal expansions is linearly ordered by a recursive indexing (e.g., lexicographical ordering).

var $\{x_i\}_{i=1}^{\omega}$, $\{y_i\}_{i=1}^{\omega}$: sequences of integgrs
$\tilde{k}$: integer
$\{r_{\tilde{k}}\}_{\tilde{k}=1}^{\omega}$: sequence of digits
comment: an element of the set
{"converges","does-not-converge","don't-know"}

Program PROBLEM1

var $q_1, q_2, \ldots$: finite decimal expansions
$c_1, c_2, \ldots$: boolean
j: integer
Dovetail

[Call Procedure INFINITE1

[Compute q_1, the first finite decimal expansion
Let c_1:= FINITE(q_1)]

[Compute q_2, the second finite decimal expansion
Let c_2:= FINITE(q_2)]

.

.

.

End-dovetail

If comment= "don't-know"

then find the integer j such that q_j = truncation$_{\tilde{k}}(r)$

{Note: q_j is the j^{th} finite decimal expansion;
the integer $\tilde{k}$, and the 1^{st} through $\tilde{k}^{th}$
digits of r were computed by Procedure
INFINITE!

if c_j="true" then let comment:= "converges"
else let comment: = "does-not-converge"

exit from **PROBLEM1**

After the execution of Program **PROBLEM1** the output tape **has** either the symbol "does-not-converge" or **has** the symbol "converges" and a real number, r, to which $\{x_i/y_i\}_{i=1}^{\omega}$ converges. (The digits of r are actually written on the output tape by Procedure INFINITE,) More precisely, $\{r_i\}_{i=1}^{\omega}$ is an infinite sequence of symbols, each of which is either a digit **or** "blur". The program also creates **a** value $\tilde{k}$, which is either an integer or an infinite sequence of **"blur" symbols.** (The variable $\tilde{k}$ **is** an infinite sequence of "blur" symbols **if** there were ω-many steps executed in the "For $\tilde{k}$" **loop** in the second half of Procedure 'INFINITE.)

Proposition 2.2. Program PROBLEM1 is correct, in that

(1) At the end of Program PROBLEM1, the value of the comment variable is either "converges" or "does-not-converge",

(2) If, at the end of Program PROBLEM1, the value of the comment variable is "converges" and the variable $\tilde{k}$ contains all "blur" symbols, then $\{x_i/y_i\}_{i=1}^{\omega}$ converges to the infinite decimal expansion r,

(3) If, at the end of Program PROBLEM1, the value of the comment variable is "converges" and the variable $\tilde{k}$ contains an integer, then $\{x_i/y_i\}_{i=1}^{\omega}$ converges to the finite decimal expansion $0.r_1r_2\ldots r_{\tilde{k}}$ [which equals $\text{truncation}_{\tilde{k}}(r)$], and

(4) If, at the end of Program PROBLEM1, the value of the comment variable is "does-not-converge", then $\{x_i/y_i\}_{i=1}^{\omega}$ does not converge.

Proof. The proof of correctness is straightforward. An outline of parts (2) through (4) is as follows:

(2) Examining the "For $\tilde{k}$" loop in the second half of Procedure INFINITE, we see that the condition stated in (2) above can only happen if, for every positive integer, k, r_k is not

equal to "blur". (Otherwise, the "For $\tilde{k}$" loop would be exited prematurely, leaving $\tilde{k}$ to be an integer.) The "For i" loop of Procedure INFINITE defined each r_k to be a digit. We must show that $\{x_i/y_i\}_{i=1}^{w}$ converges to the real number r, whose decimal expansion is the infinite sequence $0.r_1r_2r_3\ldots$ of digits, To sea this, let ϵ be 10^{-k} for some positive integer k. Since $r_1,\ldots,r_k$ are not "blur" at time w, we know that there is a positive integer i_0 such that, for all $i \geq i_0$, $j^{th}\text{-digit}(x_i/y_i) = r_j$ for all $j = 1,\ldots,k$. Thus, for $i \geq i_0$, $|x_i/y_i - r| < \epsilon$. So $\{x_i/y_i\}_{i=1}^{w}$ does indeed converge to r.

(3) If the condition given in (3) holds, then Procedure INFINITE returned the inconclusive comment "don't-know" and Program PROBLEM1 changed the comment to "converges" after consulting the run of Function FINITE whose input was $i=1$, $\{y_i\}_{i=1}^{w}$ and $\text{truncation}_{\tilde{k}}(r)$. By Proposition 2.1, the result of Function FINITE is correct,

(4) If the condition given in (4) holds, then either Procedure INFINITE returned the comment "does-not-converge", or Procedure INFINITE returned the comment "don't-know" and Program PROBLEM1 changed the comment to "does-not-converge" after consulting the run of Function FINITE whose input was $\{x_i\}_{i=1}^{w}$, $\{y_i\}_{i=1}^{w}$ and $\text{truncation}_{\tilde{k}}(r)$. In either case, the "For $\tilde{k}$" loop in the second half of Procedure INFINITE was exited prematurely, after having found $\tilde{k}$ to be the smallest

positive integer such that $r_{\tilde{k}}$ = "blur". We divide into cases, according to the action of this iteration of the loop.

Case 4.1, The value of $e_{\tilde{k}}$ is not "blur".

For concreteness assume that $e_{\tilde{k}}$ is 4. Then there is an integer, i_0, such that

(1) j^{th}-digit$(x_i/y_i) = r_j$ for all $j = 1,\ldots,\tilde{k}-1$, and

(2) $\tilde{k}^{th}$-digit(x_i/y_i) is 4 or 5

for all $i \geq i_0$.

Also, for each $i \geq i_0$, there are integers $i_4, i_5 \geq i$ such that $\tilde{k}^{th}$-digit$(x_{i_4}/y_{i_4}) = 4$ and $\tilde{k}^{th}$-digit$(x_{i_5}/y_{i_5}) = 5$. The value assigned to truncation$_{\tilde{k}}(r)$ by the "For $\tilde{k}$" loop in the second half of Procedure INFINITE was $0.r_1r_2\ldots r_{\tilde{k}-1}5$. The existence of the "$i_4$" integers indicates that if $\{x_i/y_i\}_{i=1}^{\omega}$ converges, it aonverges to a number $\leq$ truncation$_{\tilde{k}}(r)$. On the other hand, the existence of the "i_5" integers indicates that this convergence must be to a number $\geq$truncation$_{\tilde{k}}(r)$. Thus, the only number to which $\{x_i/y_i\}_{i=1}^{\omega}$ could possibly converge is truncation$_{\tilde{k}}(r)$. But upon exit from Procedure INFINITE, the Program PROBLEM1 consulted .the run of Function FINITE whose input was

truncation$_{\tilde{k}}(r)$. The function reported "false" on the convergence question, and, by Proposition 2.1, the function was correct.

Case 4.2, The value of $e_{\tilde{k}}$ is "blur" but the value of $o_{\tilde{k}}$ is not "blur". Then $o_{\tilde{k}}$ is a digit, for if $o_{\tilde{k}}$ settled into being the "not-defined" symbol, then $r_{\tilde{k}}$ would have settled into being zero, contradicting the fact that $r_{\tilde{k}}$ is "blur". Case 4.2 is then similar to Case 4.1.

Case 4.3. The value of $e_{\tilde{k}}$ is "blur" and value of $o_{\tilde{k}}$ is "blur". Then, in the "For i" loop of Procedure INFINITE, as i went to infinity, the $\tilde{k}^{th}$ digit of x_i/y_i oscillated among digits that were more than one unit apart. It is then a straightforward matter to show that $\{x_i/y_i\}_{i=1}^{\omega}$ does not converge. □

Each of the ω compound statements enclosed within "dovetail...end-dovetail" takes at most $\omega 2$ time. Thus the "dovetail...end-dovetail" portion of the program takes at most $\omega 2$ time, The remainder of the program takes finite time. So the worst case running time for Program PROBLEM? is $\omega 2$. At the end of Section 3, we will show that no machine that solves Problem 1 has worst case running time ω.

The output of Program PROBLEM? will sometimes be infi-

nite, This is necessary by virtue of the nature of Problem 1, since the answer to the problem can be an infinite decimal expansion.

3. The Limit of a Sequence of Positive Integers

3.1 Definition of the problem

On page 52, and again on page 57, we made reference to a problem which is embedded in Problem 1. We now state that problem explicitly:

Definition. Let $s = \{s_i\}_{i=1}^{\omega}$ be an infinite sequence of positive integers, If n is a positive integer, we say that the ~~limit of s is n~~, if

$$(\exists i_0)(\forall i > i_0)[s_i = n].$$

We express this symbolically as

$$\lim_{i \to \omega} s_i = n .$$

We say that ~~the limit of s is infinity~~ if

$$(\forall n)(\exists i_n)(\forall i > i_n)[s_i > n].$$

Problem 2. Given an infinite sequence, s, of positive integers, determine whether or not the limit of s is infinity.

As we hinted on pages 52 and 57, the existence of a "fast" machine (with worst case running time ω) to solve Problem 2 would imply the existence of a fast machine to solve Problem 1.

Recall that the solution to Problem 1 was, under certain conditions, an output of infinite size, In contrast, Problem 2 clearly requires a finite-size answer (such as "yes" or "no"). In reality, what constitutes an "answer" to a problem must always be determined by the nature of the entity that reads the answer. Consider, for instance, two ways to represent the number Zero: (1) Zero can be represented as an infinite decimal expansion, each digit of which is the symbol "0". If $x_i = 0$, for every positive integer i, then Program PROBLEM1 represents that $\{x_i/y_i\}_{i=1}^{\omega}$ converges to zero by printing out this infinite-decimal-expansion version of zero. Any entity reading the output of Program PROBLEM? must be prepared to read an infinite decimal expansion. (2) The other way to represent zero is with a finite symbol of some sort. Function FINITE does this by printing out "true" (since FINITE = "true" means that $\{x_i/y_i\}_{i=1}^{\omega} - q$ converges to zero). This finite representation is necessar because the entity which reads the information needs to read it in finite form. The reading entity is Program PROBLEM?. This procedure depends on Function FINITE to provide a concise answer to the convergence question for $\{x_i/y_i\}_{i=1}^{\omega}$ and each value. of q.

Let s be an infinite sequence of positive integers. We call s an $\exists$-sequence if at least one integer occurs infinitely many times in s. Otherwise s is a $\sim\exists$-sequence.

Lemma 3.1. Let s be an infinite sequence of positive integers. Then s is a $\sim\exists$-sequence iff the limit of s is infinity.

Proof. A simple pidgeonholing argument. □

Thus Problem 2 has an equivalent formulation which can be stated as follows:

Problem 2. Given an infinite sequence, s, of positive integers, determine whether s is an $\exists$-sequence or a $\sim\exists$-sequence.

We now define an infinite Turing machine T_1, T_2 to solve Problem 2: The first machine, T_1, is very much like the T_1-machine of Example 2 in Section 1.1. For each positive integer, i, machine T_1 keeps track, on $\text{tape-out}^{T_1}(i)$, of the number (mod 2) of occurrences of i in s. At time ω, machine T_2 writes "$\sim\exists$-sequence" on square 1 of its output tape. Then T_2 proceeds to examine each square of tape-out^{T_1}, starting with the leftmost square. Machine T_2 prints nothing unless it reads a "blur" symbol on

tape-out^{T_1}, in which case it replaces the symbol "~∃-sequence" on its output tape with the symbol "∃-sequence", and halts.

If s is an ∃-sequence, the running time of the infinite machine T_1,T_2 is ω. Otherwise the time required to perform the computation is $\omega 2$. Our goal is to prove that this infinite Turing machine is time optimal for Problem 2, i.e., that no infinite Turing machine can solve the problem for every sequence in less than $\omega 2$ time. In order to introduce the main ideas used in reaching that goal, we first consider a restricted form of the given problem, For the restricted form of the problem, the desired result is easier to prove.

3.2 Proof of an easier result

Assume that we have an infinite machine T_1,T_2 that solves the original problem, Problem 2, with worst case running time ω. Then T_2 never executes more than finitely many steps. This being the case, T_2 never examines more than finitely many squares of the output tape of T_1. We want to show that, given any augmented Turing machine T (i.e., any augmented algorithm), any finite portion of the output of T_1 is "ambiguous" with respect to Problem 2. The word "ambiguous" will be defined more carefully later. For now let us simply state that no augmented algorithm, T_1, can

successfully examine an infinite sequence, and, on a finite section of its output tape, correctly encode the information "$\exists$-sequence" or "~$\exists$-sequence" about the input sequence.

This result is easy to prove if we make the restriction that the answer to Problem 2 must be encoded on only one square of tape-out^{T_1}, say the leftmost square, tape-out$^{T_1}(1)$. The argument for this simplified problem contains the sequence splicing idea, which we will use later for the original Problem 2. So we will first give an informal description of the argument for this simplified version of Problem 2. In particular, we will show that

(*) If T_1 is an augmented Turing machine, then there is an infinite sequence, s, such that at time ω, the answer "s is an $\exists$-sequence" *or* "s is a ~$\exists$-sequence" is incorrectly coded on tape-out$^{T_1}(1)$.

The simplified argument given below is meant to aid the reader to understand the more complicated argument for the original, un-restricted version of Problem 2. The more complicated argument will be given in Section 3.3.

Assume, then, that we have an augmented Turing machine, TI, whose input is an infinite sequence of positive integers. Let X be the set of all symbols that may appear on tape-out$^{T_1}(1)$ at time ω. Note that X is a subset of

Output-Alphabet^{T1} u {"blur"}. The set X is a disjoint union of two sets, $X_{\exists}$ and $X_{\sim\exists}$. At time ω, if we find an element of $X_{\exists}$ on tape-out^{T1}(1), we conclude that the input sequence was an $\exists$-sequence. If we find and element of $X_{\sim\exists}$, we conclude that the input sequence was a $\sim\exists$-sequence. More formally,

$$X = \{\text{tape-out}_{\omega}^{T_1(s)}(1) \mid s \text{ is an infinite sequence of positive integers}\},$$

$$X_{\sim\exists} = \{\text{tape-out}_{\omega}^{T_1(s)}(1) \mid s \text{ is a } \sim\exists\text{-sequence}\}, \text{ and}$$

$$X_{\exists} = \{\text{tape-out}_{\omega}^{T_1(s)}(1) \mid s \text{ is an } \exists\text{-sequence}\}.$$

We will divide the proof into cases, depending on which, if either, of the sets $X_{\exists}$ or $X_{\sim\exists}$ contains the "blur" symbol.

First we define a few useful infinite sequences. Let

$$s = \langle 1, 2, 3, 4, \ldots \rangle$$

and, for each positive integer, i, let

$$t^i = \langle i, i, i, i, \ldots \rangle.$$

The result of applying T_1 to s should give us an element of $X_{\sim\exists}$. This is also true for any sequence, s', that differs

from s only in its initial segment, since any such sequence s' is also a ~∃-sequence. (E.g., s' = <5, 5, 5, 5, 10, 11, 12, 13, ...> is a ~∃-sequence.) The result of applying T_1 to any t^i-sequence should give us an element of $X_{\exists}$. The same is true for any sequence, $t^{i'}$, that differs from a ti-sequence only in its initial segment.

Now we divide the proof of (*) into three cases:

Case 1. Neither $X_{\exists}$ nor $X_{\sim\exists}$ contain the "blur" symbol.

Then "blur" cannot appear on tape-out$^{T_1}(1)$ at time ω. If T_1 wants to tell us that the input; sequence is, say, an ∃-sequence, it has to decide on this at some finite time by settling in on a symbol in $X_{\exists}$ and never changing its mind again. Likewise with a -3-sequence. We will compose a sequence, $s_{\exists}$, which tricks T_1 into producing the "blur" symbol on tape-out$^{T_1}(1)$, contrary to our assumptions.

We begin the construction of $s_{\exists}$ with an initial segment of s. How large an initial segment should we choose? Since s is a ~∃-sequence, machine T_1, upon examining s, must at some finite time write an element of $X_{\sim\exists}$ on tape-out$^{T_1}(1)$, and never change that symbol again, Since this happens at a finite time in the computation, T_1 could only have examined a finite initial segment of s by that time. Let f_1 be this initial segment of s. Onto f_1 we ooncatenate f_2, a segment

of t^1. How large a segment should we choose? We should choose f_2 large enough so that T_1 changes its symbol in $X_{\sim\exists}$ to a symbol in $X_{\exists}$. Once again, this has to be done at a finite time in the computation, even though only an initial segment of the sequence

$$f_1 t^1 = \langle 1, 2, 3, 4 \ldots n, 1\ 1, 1, 1, \ldots \rangle$$

has been examined by T_1.

Thus in examining $f_1 f_2$, machine T_1 will, among other things, write an element of $X_{\sim\exists}$ on tape-out$^{T_1}(1)$ and later change that symbol to an element of $X_{\exists}$.

Onto $f_1 f_2$ we concatenate f3, an initial segment of s. Again, in preparation for the examination of the entire infinite sequence $f_1 f_2 s$, T_1 must, at some finite time during; the examination of s, change the answer on its output tape back to an element of $X_{\sim\exists}$.

We continue in this manner, forming $s_{\exists}$ by splicing together segments of $\exists$-sequences and $\sim\exists$-sequences in an alternating fashion. By doing this, we force T_1 to change its mind infinitely often between symbols in $X_{\exists}$ and $X_{\sim\exists}$. Therefore, at time ω, tape-out$^{T_1}(1)$ contains the "blur" symbol, This contradicts the Case 1 hypothesis.

Case 2. The "blur" symbol is an element of $X_{\sim\exists}$.

In order to indicate that the input sequence is a $\sim\exists$-sequence, T_1 may, at some finite time, settle on a non-"blur" element of $X_{\sim\exists}$, or it may change its mind infinitely often about the content of tape-out$^{T_1}(1)$, producing a "blur". In contrast, the elements of $X_{\exists}$ are all (ordinary) elements of the output alphabet of T_1. We must create a contradiction by finding an $\exists$-sequence that forces T_1 to produce a "blur".

Notice that the sequence $s_{\exists}$, created for Case 1, was an $\exists$-sequence. With one slight modification to the argument we can use sequence $s_{\exists}$ to contradict the hypothesis of Case 2. The modification is as follows: When we choose f_{2i+1}, an odd-numbered initial segment of s, we cannot argue as we did in Case 1 that T_1 will, upon examination of some initial segment of $f_1 f_2 \ldots f_{2i} s$, settle on an element of $X_{\sim\exists}$. In Case 2 machine T_1 may never settle on an non-"blur" symbol from $X_{\sim\exists}$, since the machine can indicate a -3-sequence by changing its mind infinitely often, producing a "blur". Instead we argue as follows: Assume that we have constructed $f_1 f_2 \ldots f_{2i}$. Since 2i is an even number, machine T1 will, upon examining $f_1 f_2 \ldots f_{2i}$, write an element of X_3 on tape-out$^{T_1}(1)$. We must choose a finite sequence f_{2i+1} so that, while proceeding on to examine f_{2i+1}, T_1 changes the content of tape-out$^{T_1}(1)$ from that element of $X_{\exists}$ to some

other symbol. To do this, we consider again the infinite sequence $f_1 f_2 \ldots f_{2i} s$. Since this is a $\sim\exists$-sequence, there are two possibilities:

Case 2.1. With $f_1 f_2 \ldots f_{2i} s$ on the input tape of T_1, the content of tape-out$T_1(1)$ at time ω would be a non-"blur" element of $X_{\sim 3}$.

Then, at some finite time during the examination of s, machine T_1 changes the content of 'cape-out$T_1(1)$ to a non-"blur" element of $X_{\sim\exists}$. As in Case 1, T_1 has examined only a finite initial segment of s by the time it changes tape-out$^{T_1}(1)$. We choose f_{2i+1} to be that initial segment.

Case 2.2. With $f_1 f_2 \ldots f_{2i} s$ on the input tape of T_1, the content of tape-out$^{T_1}(1)$ at time ω would be a "blur".

Then, during the examination of s, machine T_1 changes the content of tape-out$^{T_1}(1)$ infinitely many times. Let f_{2i+1} be an initial segment of s which is long enough to insure that, upon examination of f_{2i+1}, machine T_1 changes the content of tape-out$^{T_1}(1)$ at least once.

In either case (2.1 or 2.2), machine T_1 changes the content of tape-out$^{T_1}(1)$ upon examination of f_{2i+1}. This will happen infinitely many times as T_1 examines the infinite sequence $s_3 = f_1 f_2 f_3 \ldots$. Thus, with $s_3 = f_1 f_2 f_3 \ldots$ on the

input tape of T_1, the content of tape-out$T_1(1)$ at time ω will be "blur". As in Case 1, the sequence s_3 that we have constructed is an $\exists$-sequence. Thus, once again, we have tricked T_1 into producing "blur", an element of $X_{\sim\exists}$, upon examination of $s_{\exists}$, an $\exists$-sequence.

Case 3. The "blur" symbol is an element of $X_{\exists}$.

In order to indicate that the input sequence is an $\exists$-sequence, T_1 may, at some finite time, settle on a non-"blur" element of $X_{\exists}$, or it may change its mind infinitely often about the content of tape-out$T_1(1)$, producing a "blur". In contrast, the elements of $X_{\sim 3}$ are all (ordinary) elements of the output alphabet of T_1. We must create a contradiction by finding a $\sim\exists$-sequence that forces T_1 to produce a "blur".

We must modify the construction to produce $s_{\sim\exists}$, a $\sim\exists$-sequence. Choose f_1 as before, and assume that f_1 is <1, 2, 3, ..., n>. Choose f_2 to be a segment of t^{n+1}. This gives us <1, 2, 3, ..., n, n+1, n+1, ..., n+1>. Choose f_3 to be a segment of s that starts with the positive integer n+2. This gives us <1, 2, 3, ... n, n+1, ..., n+1, n+2, n+3, n+4, ..., n+m>. And so on. Notice that we always Choose a new segment which has no integers in common with the previous segments. (We will say that these segments are disjoint).. Using this trick, we are assured that the re-

sulting sequence, $s_{\sim\exists}$, is a $-+$sequence. As in Cases 1 and 2, we construct our sequence by alternating infinitely many times between 3-sequences and $\sim\exists$-sequences, producing a "blur" on tape-out$T^1(1)$. □

This completes the proof of the simplified version of main theorem.

3.3 Proof of the main result

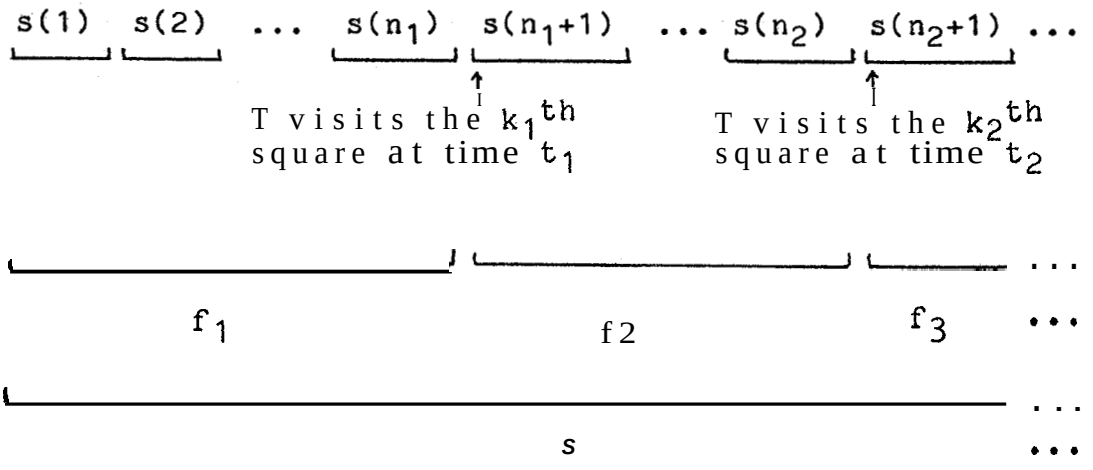
Now we drop the simplifying assumptions, and begin some of the formal details of the proof of the general result.

Notation: In the discussion that follows, s will always stand for an infinite sequence of positive integers and s(i) or s_i will denote the i^{th} member of the sequence; f, f_1, f_2,..., etc. will be finite sequences of positive integers.

Let T be an augmented Turing machine. Assume that s is written on the input tape of T. We consider several finite segments of the sequence s. Let f_1 be $\langle s(1),\ldots,s(n_1)\rangle$ and f_2 be $\langle s(n_1+1),\ldots,s(n_2)\rangle$. The notation f_1f_2 will be used to denote the concatenated sequence $\langle s(1),\ldots,s(n_1),s(n_1+1),\ldots,s(n_2)\rangle$. If p is a positive integer, then f_1p will represent the sequence $\langle s(1),\ldots,s(n_1),p\rangle$. (I.e., f_1p is $f_1\langle p\rangle$.) We say that f_1 and f_2 are <u>disjoint</u> iff

$(\forall i,\ 1 \le i \le n_1)(\forall j,\ n_1+1 \le j \le n_2)[s(i) \neq s(j)]$.

With T, s, f_1 and f_2 defined as above, let the first digit of $s(n_1+1)$ (respectively $s(n_2+1)$) be on the k_1^{th} (respectively k_2^{nd}) square of tape-in T.



Let t_1 (respectively t_2) be the smallest positive integer such that position-in$_{t_1}^{T(s)}$ = k_1 (respectively position-in$_{t_2}^{T(s)}$ = k_2). The integer t_1 (respectively t_2) represents the earliest time when we can be assured that T has read all of f_1 (respectively f_2). Let i be a positive integer. We define tape-out$^{T(f_1)}$(i) to be tape-out$_{t_1-1}^{T(s)}$(i). Intuitively tape-out$^{T(f_1)}$(i) is the symbol written on the i^{th} square of the output tape of T as T begins to move past f_1 for the first time. We say that f_2 changes tape-out$^{T(f_1)}$(i) iff for some j, $t_1 < j \le t_2$, tape-out$_j^{T(s)}$(i) $\neq$ tape-out$^{T(f_1)}$(i).

This means that the symbol written on the i^{th} square of the output tape of T does not remain constant as T proceeds to include f_2 in its knowledge of the contents of tape-in.

We say that f_1 fixes tape-out$_T(i)$ iff for *any* finite sequence f_2, if f_1f_2 is encoded on the leftmost squares of tape-in$_T$, and f_2 is disjoint from f_1, then f_2 does not change $\text{tape-out}^{T(f_1)}(i)$. Intuitively, this means that once T has examined f_1 on its input tape, the i^{th} symbol on the output tape of T will never be changed again (provided T does not find another occurrence of an integer that occured in f_1).

The following lemmas are trivial consequences of the definitions of "changes" and "fixes", and are stated without proof.

Lemma 3.2. Let $s = f_1f_2f_3\ldots$, as above, Let i be a positive integer. If, for infinitely many j, f_{j+1} changes $\text{tape-out}^{T(f_1\ldots f_j)}(i)$ then $\text{tape-out}_\omega^{T(s)}(i) = \text{"blur"}$.

Lemma 3.3. Let $s = f_1f_2f_3\ldots$, as above. Let i be a positive integer. If f_2 changes $\text{tape-out}^{T(f_1)}(i)$, then f_2 is a non-empty sequence.

Lemma 3.4, Let $s = f_1f_2f_3\ldots$, as above. Let i be a positive integer. If f_2 changes $\text{tape-out}^{T(f_1)}(i)$, then f_2f_3 changes

$\text{tape-out}^{T(f_1)}(i)$.

Lemma 3.5. Let $s = f_1 f_2 f_3 \ldots$, with f_j disjoint from f_1, for each $j>1$. Let i be a positive integer. If f_1 fixes $\text{tape-out}_{T(i)}$, then $\text{tape-out}_{\omega}^{T(s)}(i) = \text{tape-out}^{T(f_1)}(i)$.

Lemma 3.6. Let $s = f_1 f_2 f_3 \ldots$, with f_2 disjoint from f_1. Let i be a positive integer. If f_1 fixes $\text{tape-out}_{T(i)}$, then $f_1 f_2$ fixes $\text{tape-out}_{T(i)}$.

We will use the notation and definitions given above to construct sequences of positive integers. These sequences will force our infinite Turing machine to take at least $\omega 2$ time to solve Problem 2.

In solving Problem 2, a machine is given an input tape containing an infinite sequence of positive integers. The machine must examine each member of the sequence (or at least all-but-finitely-many members of the sequence). Thus the action of the machine will take at least ω time. We divide the machine into two parts, T_1 and T_2. The first part, T_1, is an augmented Turing machine which performs the first w steps. The second part, T_2, reads the output of T_1 and performs all other steps necessary to solve Problem 2.

Since T_1 performs ω steps, we can increase its duties by insisting that it make a duplicate copy of its input tape on

alternate squares of its output tape, This work can be intertwined with the machine's other duties, so that the modified machine still has worst case running time w. This added work certainly does not decrease the chance that the combined machine T_1, T_2 can solve Problem 2, In fact, it provides T_2 with a second copy of information that is already available to it by definition of tape-in^{T_2}. (Recall from the definition of an infinite Turing machine that tape-in^{T_i} = tape-in$^{T_{i-1}}$ $\cup$ tape-out$_\omega^{T_{i-1}}$.) Why, then do we insist on having T_1 make this oopy of its input tape? If T_1 does this, then we can simplify the definition of tape-in^{T_2}. We can assume that tape-in^{T_2} = tape-out$_\omega^{T_1}$. This is'because all the information contained on tape-in^{T_1} $\cup$ tape-out$_\omega^{T_1}$ is encoded on tape-out$_\omega^{T_1}$. Given this simplified definition, anything we observe about tape-out$_\omega^{T_1}$ is also true about tape-in^{T_2}. This will be useful in the proofs of several lemmas.

In the above paragraph we argued that we can, without loss of generality, increase the duties of machine T_1 by insisting that it copy the squares of its input tape on alternate squares of its output tape, In this paragraph we increase the machine's duties even further. Recall the mahine of Example 2 in Section 1.1. That machine recorded, on a square of its output tape, the number (modulo 2) of occurrences of 42 that it found on its input tape. We create a similar duty for our Problem 2-machine, T_1. We insist

that machine T_1 record, on the leftmost square of its output tape, the number (mod 2) of squares that it has visited on tape-in^{T_1}, the content of tape-out$^{T_1}(1)$ will oscillate end-

On first sight this added duty seems to be a waste of the machine's effort, The content of tape-out$^{T_1}(1)$ will be of no use in solving Problem 2. Furthermore, the content of this square is completely predictable; i.e., no matter what kind of sequence is encoded on tape-in^{T_1}, the content of tape-out$_{\omega}^{T_1}(1)$ is always "blur". But that predictability is exactly what makes this duty important. When we increase the machine's duties this way, we guarantee that tape-out$^{T_1}(1)$ is not fixed by any finite initial segment of the sequence encoded on tape-in^{T_1}. This will be useful in the proof of the main lemma (Lemma 3.9). Notice also that, as in the above discussion on tape-copying, adding this counting-mod-2 to the list of the machine's duties does not increase the running time of the machine, nor does it decrease the chance that the combined machine T_1, T_2 can solve Problem 2.

The remarks made in the last three paragraphs are summarized in the following definitions and lemma:

Definition, Let T_1 be an augmented Turing machine. For each positive integer i, let t_i be the smallest positive integer such that $\text{position-in}_{t_i}^{T_1} > i$. Then T_1 is called a copying machine' (or, for emphasis, an augmented copying Turing machine) if

$$\text{tape-out}_t^{T_1}(2i) = \text{tape-in}^{T_1}(i)$$

for every i, for every $t \geq t_i$.

Definition. Let T_1 and t_i be defined as above. Then T_1 is called a counting machine (or, for emphasis, an augmented counting Turing machine) if

$$\text{tape-out}_{t_i}^{T_1}(1) = \begin{cases} \text{"even" if } i \text{ is even} \\ \\ \text{"odd" if } i \text{ is odd} \end{cases}$$

for every positive integer i.

Lemma 3.7. If Problem 2 can be solved by an infinite Turing machine whose worst case running time is ωn, then Problem 2 can be solved by an infinite Turing machine whose worst case running time is ωn and whose first augmented Turing machine is a copying and counting machine.

The proof of Lemma 3.7 follows trivially from the re-

marks made above. Henceforth when we refer to an infinite Turing machine T_1, T_2 to solve Problem 2, we will assume that T_1 is a copying and counting machine. Notice that, as a consequence of the definition, a copying machine always performs ω steps.

Returning to our main goal, we are preparing to prove that an infinite Turing machine T_1, T_2 which solves Problem 2 takes at least ω^2 time. In order to show this, we must find sequences that force T_2 to take at least ω time. To do this we will show that T_2 cannot do its job without examining infinitely many squares of the output tape of T_1. Surprisingly, T_2 has very little to do with the proof. What we actually show is that the answer to Problem 2 is not encoded on finitely many squares of the output tape of T_1. We formalize this notion in the following definitions:

Definition. Let n be a positive integer. An augmented Turing machine, TI, is n-ambiguous if there is a $\sim\exists$-sequence, $s_{\sim\exists}$, and an $\exists$-sequence, $s_{\exists}$, such that

$$\text{tape-out}_{\omega}^{T(s_{\sim\exists})}(i) = \text{tape-out}_{\omega}^{T(s_{\exists})}(i)$$

for all $i \leq n$. The $\sim\exists$-sequence is said to n-witness the n-ambiguity of T_1. The $\exists$-sequence is said to n-cowitness the n-ambiguity of T_1.

Definition. An augmented Turing machine, T_1, is uniformly ambiguous, if there is a $\sim\exists$-sequence, $s_{\sim\exists}$, such that, for every positive integer n, $s_{\sim\exists}$ n-witnesses the n-ambiguity of T_1. The $\sim\exists$-sequence is said to witness the uniform ambiguity of T_1.

Notice that if a $\sim\exists$-sequence witnesses the uniform ambiquity of T_1, then for each positive integer n there must be an $\exists$-sequence that n-cowitnesses the n-ambiguity of T_1.

The uniform ambiguity of T_1 is a sufficient condition to force T_1, T_2 to take at least $\omega 2$ time. This is stated formally in the next lemma.

Lemma 3.8. Let T_1 and T_2 be augmented Turing machines which satisfy the following conditions:

(a) T_1 is a copying machine,

(b) T_1 is uniformly ambiguous, and

(c) the infinite Turing machine T_1, T_2 solves Problem 2.

Then the worst case running time of T_1, T_2 is $\omega 2$.

Proof. Since T_1 is a copying machine we can assume that $\text{tape-in}^{T_2} = \text{tape-out}_{\omega}^{T_1}$. We also know that T_1 performs ω

steps, since it copies its entire input tape onto its output tape, We must show that the worst case running time of T_2 is ω.

Let $s_{\sim\exists}$ be the infinite -3-sequence which witnesses the uniform ambiguity of T_1. Assume that $s_{\sim\exists}$ is written on the input tape of T_1. After T_1 has performed its ω-many steps, machine T_2 must examine the output tape of T_1 and draw the conclusion that the sequence given to T_1 was a $\sim\exists$-sequence. We claim that T_2 cannot do this by examining only finitely many squares of tape-outT_1.

Assume the contrary. Let n be a positive integer. Let T_2 halt after examining n squares of tape-outT_1. Since the input to T_1 was a $\sim\exists$-sequence, T_2 writes "$\sim\exists$-sequence", in some form, on its output tape before halting. By the n-ambiguity of TI, there is an n-cowitness sequence, $s_{\exists}$, satisfying tape-out$_{\omega}^{T_1}(s_{\exists})(i)$ = tape-out$_{\omega}^{T_1}(s_{\sim\exists})(i)$ for all $i \leq n$. So if we had written $s_{\exists}$ on the input tape of T_1, machine T_2 would have examined the same symbols on the same n squares of tape-out^{T_1}, and thus would have halted after writing the same message; namely, that T_1 had been given a $\sim\exists$-sequence. Since $s_{\exists}$ is an $\exists$-sequence, this conclusion would be incorrect.

Therefore T_2 must examine infinitely many squares of tape-out^{T_1}. It cannot do this in finitely many steps. So

the worst case running time of T_2 is ω, and the worst case running time of T_1, T_2 is $\omega 2$. □

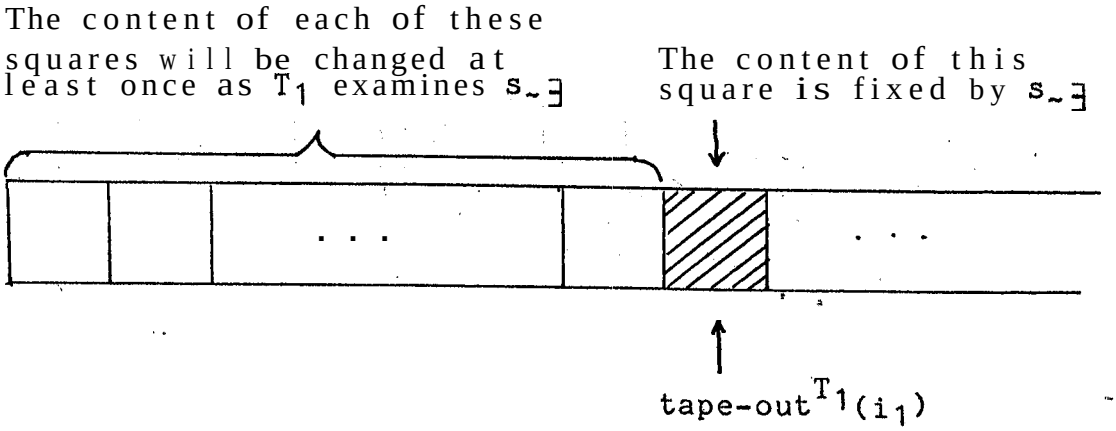
Lemmas 3.7 and 3.8 combined with the main lemma (Lemma 3.9) will yield the desired result. We are now prepared to state and prove the main lemma.

Lemma 3.9. Every augmented copying and counting Turing machine is uniformly ambiguous.

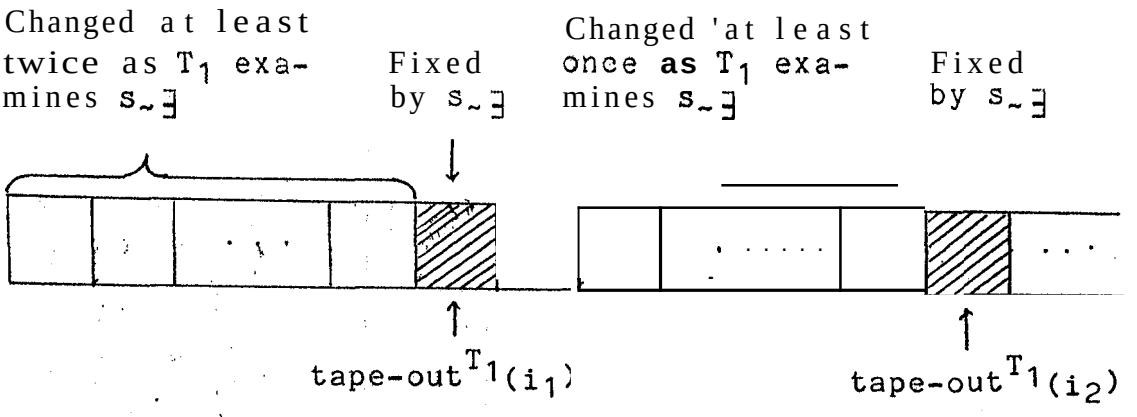
Proof. First we give the intuitive idea. After that we will present the proof in rigorous detail.

We start by constructing a sequence $s_{\sim\exists}$. The idea behind the construction is quite simple: Construct $s_{\sim\exists}$ so as to fix as many squares as possible, and force the rest of the squares to contain a "blur". Begin with $s_{\sim\exists}$ being the empty sequence. Choose the leftmost square of tape-out^{T_1} whose content is fixed by some finite sequence f_1. Let i_1 be the position of this square on the output tape of T_1. (Thus, the square itself is called $\text{tape-out}^{T_1}(i_1)$. We are certain that such a square can be found because of the assumption that T_1 is a copying machine.) Extend $s_{\sim\exists}$, the empty sequence, by concatenating f_1 onto it. Now the square $\text{tape-out}^{T_1}(i_1)$ is fixed by $s_{\sim\exists}$. By the choice of i_1, no square to the left of $\text{tape-out}^{T_1}(i_1)$ can be fixed by any finite extension of $s_{\sim\exists}$. We can force

the **symbol** in each of these squares to change by extending $s_{\sim\exists}$ appropriately. So we have



Let tape-out$^{T_1}(i_2)$ **be** the next **square** (the leftmost **square** that **is** to the right of tape-out$^{T_1}(i_1)$) whose content can **be** fixed by **some** finite extension, $s_{\sim\exists}f_2$, of $s_{\sim\exists}$, where f_2 is disjoint from $s_{\sim\exists}$. **Extend** $s_{\sim\exists}$ by concatenating f_2 onto **it. Now** the squares tape-out$^{T_1}(i_1)$ and tape-out$^{T_1}(i_2)$ **are** fixed by $s_{\sim\exists}$. By the choice of i_2, **no** square between tape-out$^{T_1}(i_1)$ **and** taps-out$^{T_1}(i_2)$ can be fixed **by** any finite extension, $s_{\sim\exists}f$, of $s_{\sim\exists}$, as long as we insist that f be disjoint from $s_{\sim\exists}$. In eaoh square to the left of tape-out$^{T_1}(i_1)$, and **each** square between tape-out$^{T_1}(i_1)$ **and** tape-out$^{T_1}(i_2)$ **we** can force **the symbol** to change **by** extending $s_{\sim\exists}$ appropriately. Now we have

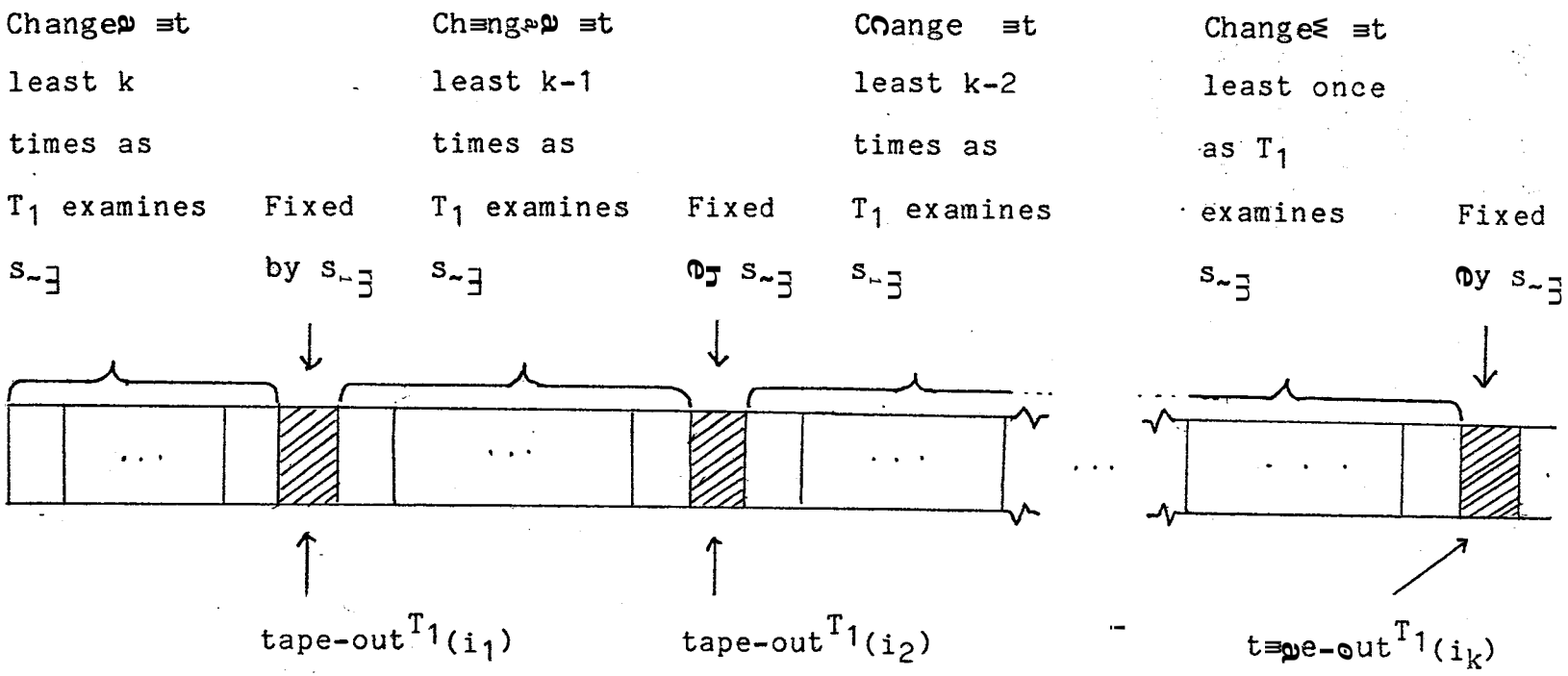


The next square that can be fixed by a finite extension of $s_{\sim\exists}$ will be to the right of tape-out$^{T_1}(i_2)$. Choose the leftmost such square, and repeat the extension process for $s_{\sim\exists}$. Doing this several times, we get the output tape which is illustrated in the figure on the next page.

Extending $s_{\sim\exists}$ infinitely many times, we get a $\sim\exists$-sequence, $s_{\sim\exists}$, which fixes the squares $i_1, i_2, i_3, \ldots$ and forces T_1 to produce "blur" in all the other squares.

We must show that, for each positive integer n, the sequence $s_{\sim\exists}$ n-witnesses the n-ambiguity of T_1. We do this in the following way:

Let tape-out$^{T_1}(1)$ .....tape-out$^{T_1}(n)$ be some finite initial piece of the output tape of TI. Let tape-out$^{T_1}(i_1)$, ...,tape-out$^{T_1}(i_z)$ be the squares in that initial piece that


Change at least k times as T_1 examines s_{-3}
Fixed by s_{-3}
Change at least k-1 times as T_1 examines s_{-3}
Fixed by s_{-3}
Change at least k-2 times as T_1 examines s_{-3}
Change at least once as T_1 examines s_{-3}
Fixed by s_{-3}
tape-out$^{T_1}(i_1)$
tape-out$^{T_1}(i_2)$
tape-out$^{T_1}(i_k)$

were fixed by initial segments of $s_{\sim\exists}$. Let f_n^* be the largest of such initial segments of $s_{\sim\exists}$. Then f_n^* fixes each of tape-out$^{T_1}(i_1),\ldots,$tape-out$^{T_1}(i_z)$. So if $s_\exists$ is an infinite $\exists$-sequence with initial segment f_n^*, and if the rest of $s_\exists$ is disjoint from f_n^*, then $s_\exists$ and $s_{\sim\exists}$ force T_1 to produce exactly the same output on tape-out$_\omega^{T_1}(i_1),\ldots,$ tape-out$_\omega^{T_1}(i_z)$. Thus, in order to show that $s_{\sim\exists}$ n-witnesses the n-ambiguity of T_1, we must construct an n-cowitness $\exists$-sequence, $s_\exists$, by concatenating disjoint segments onto f_n^*. The sequence $s_\exists$ must also be constructed so that it forces the content of the other squares to be "blur". If we do this properly, then the content of tape-out$_\omega^{T_1}(1),\ldots,$ tape-out$_\omega^{T_1}(n)$ is the same whether T_1 receives $s_{\sim\exists}$ or $s_\exists$ as input. So $s_{\sim\exists}$ n-witnesses the n-ambiguity of T_1. We do this for arbitrary n.

We now give the proof in rigorous detail. We are given machine T_1 which is an augmented copying and counting Turing machine. We present an algorithm to construct a sequence, $s_{\sim\exists}$, which witnesses the uniform ambiguity of T_1. Since $s_{\sim\exists}$ is an infinite sequence, the algorithm to construct it is an infinite algorithm. It should be emphasized that, although the sequence is given by an algorithm, we are concerned only with the existence of the sequence, not with the availability of a method for constructing it.

Construction of the witness $s_{\sim\exists}$:

var T_1: augmented copying Turing machine

Program CONSTRUCTION-OF-$s_{\sim\exists}$

var $s_{\sim\exists}$: sequence of positive integers

k, j, i_k: integers

f, f', f_k: finite sequences of positive integers

Fixed, Unfixable: sets of positive integers

{initialize}

Let $s_{\sim\exists}$:= the empty sequence

Let k:= 0

Let i_k:= 0

Let Fixed:= the empty set

Let Unfixable:= the empty set

For k:= 1 toward ω

{Find i_k, the next square that can be fixed}

Let i_k be the smallest positive integer greater than i_{k-1} such that there is a finite sequence, $\tilde{f}$, disjoint from $s_{\sim\exists}$, such that $s_{\sim\exists}\tilde{f}$ fixes tape-out$^{T_1}(i_k)$

Let f be any finite sequence, disjoint from $s_{\sim\exists}$, such that $s_{\sim\exists}f$ fixes tape-out$^{T_1}(i_k)$

{add i_k to the list of Fixed squares}

Let Fixed:= Fixed $\cup$ $\{i_k\}$

{add the squares between i_{k-1} and i_k to the list of Unfixable squares}

For $j := i_{k-1}+1$ to i_k-1

Let Unfixable := Unfixable $\cup$ {j}

{change the content of all the Unfixable squares}

For each $j \in$ Unfixable

Choose f', disjoint from $s_{\sim\exists}f$, such that

f' changes $\text{tape-out}^{T_1}(s_{\sim\exists}f)_{(j)}$

Let f := ff'

{enlarge the sequence $s_{\sim\exists}$}

Let $f_k := f$

Let $s_{\sim\exists} := s_{\sim\exists}f_k$

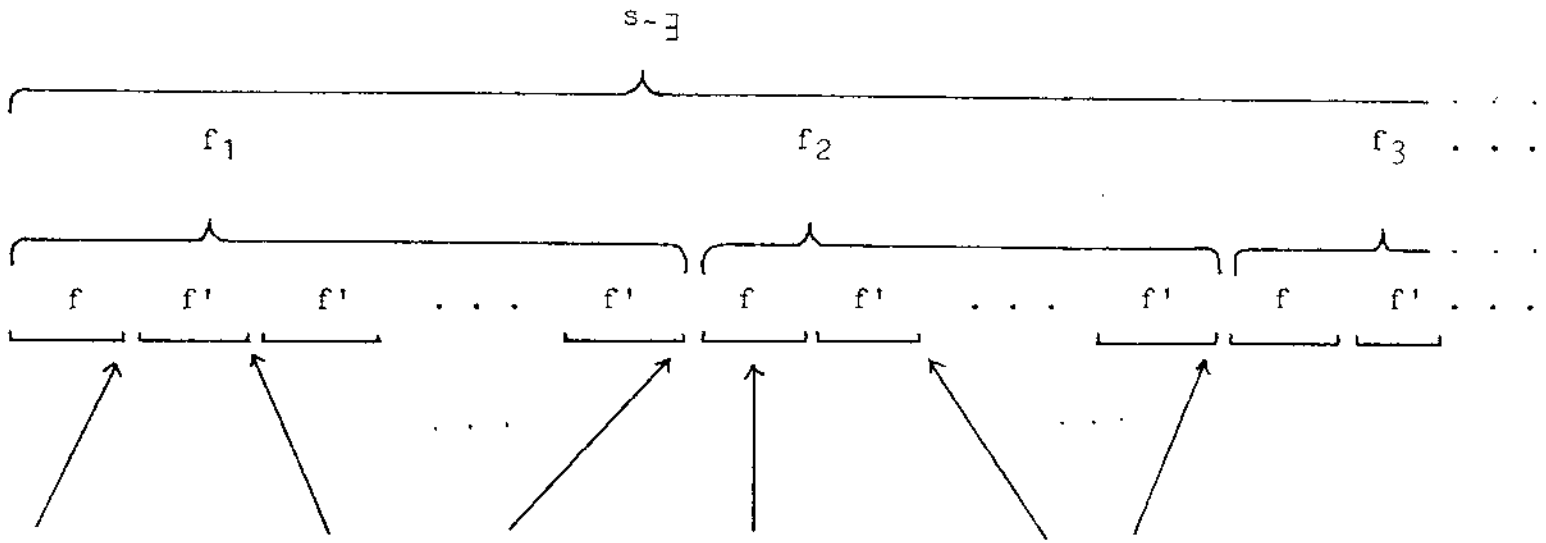
The construction of $s_{\sim\exists}$ partitions the squares of tape-out^{T_1} into two disjoint sets, Fixed and Unfixable. The action of T_1 upon examining $s_{\sim\exists}$ is illustrated in the figure on the next page.

We must show two things about Program CONSTRUCTION-OF-$s_{\sim\exists}$, namely

(1) that the algorithm will not halt unexpectedly, and

(2) that the algorithm constructs an infinite $\sim\exists$-sequence, $s_{\sim\exists}$, which witnesses the uniform ambiguity of T_1.

Proof of (1). We must show that, for every positive integer k_1, we can complete the k_1^{th} iteration of the "For k" loop. To do this we must show that



When T_1 finishes reading f, the content of tape-out$^{T_1}(i_1)$ gets fixed

The content of each square to the left of tape-out$^{T_1}(i_1)$ gets changed

The content of tape-out$^{T_1}(i_2)$ gets fixed

The content of each square to the left of tape-out$^{T_1}(i_1)$ gets changed again; the content of each square between tape-out$^{T_1}(i_1)$ and tape-out$^{T_1}(i_2)$ gets changed

(1.1) there is a positive integer, i_{k_1}, satisfying the required conditions, and

(1.2) in the "For each j ϵ Unfixable" loop there is always a finite sequence f' satisfying the required conditions.

Proof of (1.1). Assume that the k_1-1st iteration of the "For k" loop has just been executed. Then i_{k_1-1} is defined and $s_{\sim\exists}$ is the finite sequence $f_1f_2\ldots f_{k_1-1}$. Let i_{k_1} be any even positive integer greater than i_{k_1-1}. Let f_{k_1} be any finite (possibly empty) sequence such that the sequence $f_1f_2\ldots f_{k_1}$ is long enough to occupy at least $i_{k_1}/2$ squares on the input tape of T_1. Since T_1 is a copying machine, it makes an exact copy of its input tape on alternate squares of its output tape. Therefore, having $f_1f_2\ldots f_{k_1}$ written on tape-in^{T_1} determines completely the content of tape-out$^{T_1}(i_{k_1})$. We cannot change tape-out$^{T_1}(i_{k_1})$ by extending $f_1f_2\ldots f_{k_1}$. Thus tape-out$^{T_1}(i_{k_1})$ is fixed by $f_1f_2\ldots f_{k_1}$.

Proof of (1.2). Assume that the k_1th iteration of the "For k" loop is being executed, and that execution of the "For j:= i_{k-1}+1 to i_k-1" loop has just ended. At this instant, $s_{\sim\exists}$ is the finite sequence $f_1f_2\ldots f_{k_1-1}$, and Unfixable is a finite set. We have also chosen an integer i_{k_1}, and a finite sequence f, disjoint from $s_{\sim\exists}$ such that $s_{\sim\exists}$ f fixes

tape-out$^{T_1}(i_{k_1})$. Let j be an element of Unfixable. We need to show that there is a finite sequence, f', disjoint from $s_{\sim\exists}f$, such that f' changes tape-out$^{T_1}(s_{\sim\exists}f)_{(j)}$.

The positive integer j was made an element of Unfixable during some iteration of the "For k" loop. At the time, k had a certain value; call it k_0. Then

$$s_{\sim\exists} = f_1f_2\cdots f_{k_1-1}$$
$$= f_1f_2\cdots f_{k_0-1}f_{k_0}\cdots f_{k_1-1}.$$

By virtue of the choice of each f and f' in Program CONSTRUCTION-OF-$s_{\sim\exists}$, $f_{k_0}\cdots f_{k_1-1}f$ is disjoint from $f_1f_2\cdots f_{k_0-1}$.

The positive integer j was placed in the set Unfixable during the k_0^{th} iteration of the "For k" loop, so j is greater than i_{k_0-1} and less than i_{k_0}. By virtue of the way in which i_{k_0} was chosen, we know that there is no finite sequence, $\tilde{f}$, disjoint from $f_1f_2\cdots f_{k_0-1}$ such that $f_1f_2\cdots f_{k_0-1}\tilde{f}$ fixes tape-out$^{T_1}(j)$.

Since $f_{k_0}\cdots f_{k_1-1}f$ is disjoint from $f_1f_2\cdots f_{k_0-1}$, $f_1f_2\cdots f_{k_0-1}f_{k_0}\cdots f_{k_1-1}f$ does not fix tape-out$^{T_1}(j)$. Thus, $s_{\sim\exists}f$ does not fix tape-out$^{T_1}(j)$. Therefore, there is a finite sequence, f', disjoint from $s_{\sim\exists}f$, such that f' changes tape-out$^{T_1}(s_{\sim\exists}f)_{(j)}$.

Having concluded the proofs of items (1.1) and (1.2), we procede to the proof of item (2). We divide item (2) into three sub-items. We show that

(2.1) $s_{\sim\exists}$ is an infinite sequence,

(2.2) $s_{\sim\exists}$ is a $\sim\exists$-sequence, and

(2.3) $s_{\sim\exists}$ witnesses the uniform ambiguity of T_1.

Proof of (2.1). Recall, from the discussion preceding the definition of a counting machine, that tape-out$^{T_1}(1)$ is not fixed by any finite initial segment of the sequence encoded on tape-in^{T_1}. Thus, in the construction of $s_{\sim\exists}$, when k is equal to 1, the positive integer value assigned to i_k $(=i_1)$ is sure to be greater than 1. Since $i_0 = 0$ and $i_1 > 1$, the value of j in the first iteration of the "For j" loop is 1. Thus, when k is equal to 1, the set Unfixable becomes a non-empty set. Since elements are never removed from Unfixable, the set Unfixable remains non-empty during the course of the construction.

By item (1) above, the "For k" loop in Program CONSTRUCTION-OF-$s_{\sim\exists}$ undergoes ω iterations. In each iteration, Unfixable is a non-empty set. During each iteration, for each $j \in$ Unfixable, we extend $s_{\sim\exists}$ by adding,

among other things, a finite sequence f' which changes $\text{tape-out}^{T_1(s_{\sim\exists}f)}(j)$. By Lemma 3.3, f' is non-empty. Thus $s_{\sim\exists}$ is a concatenation of infinitely many non-empty sequences. Therefore, $s_{\sim\exists}$ is an infinite sequence.

Proof of (2.2). In the course of the construction, whenever we entend $s_{\sim\exists}$, we do so by concatenating a finite sequence which is disjoint from the initial segment of $s_{\sim\exists}$ that has already been constructed. Clearly this makes $s_{\sim\exists}$ a $\sim\exists$-sequence.

Proof of (2.3). We must show that, for every positive integer n, $s_{\sim\exists}$ n-witnesses the n-ambiguity of T_1. Thus for every positive integer n we must find an n-cowitness sequence; namely, an $\exists$-sequence, $s_{\exists}$, such that

$$\text{tape-out}_{\omega}^{T_1(s_{\exists})}(i) = \text{tape-out}_{\omega}^{T_1(s_{\sim\exists})}(i)$$

for all $i \leq n$. We first determine the value of $\text{tape-out}_{\omega}^{T_1(s_{\sim\exists})}(i)$, for each $i \leq n$.

Let n be a positive integer. Let k_n be the smallest value of k in CONSTRUCTION-OF-$s_{\sim\exists}$ such that $i_k \geq n$. Let $f_n^* = f_1 \ldots f_{k_n}$. Let $\text{Fixed}_{i_{k_n}} = \{i \in \text{Fixed} \mid i \leq i_{k_n}\}$. Let $\text{Unfixable}_{i_{k_n}} = \{j \in \text{Unfixable} \mid j \leq i_{k_n}\}$. Note that the set $\{1,2,\ldots,n\}$ is a subset of $\text{Fixed}_{i_{k_n}} \cup \text{Unfixable}_{i_{k_n}}$ since $i_{k_n} \geq n$. Note also that $\text{Fixed}_{i_{k_n}} \cap \text{Unfixable}_{i_{k_n}}$ = the empty

set. We will determine the value of $\text{tape-out}_{\omega}^{T_1(s_{-\exists})}(i)$ for each $i \in \text{Fixed}_{i_{k_n}} \cup \text{Unfixable}_{i_{k_n}}$.

Let k_0 be a positive integer, less than or equal to k_n. In CONSTRUCTION-OF-$s_{-\exists}$, f_{k_0} was formed by concatenating several finite sequences. The first of these sequences, f, was chosen so that $f_1 \ldots f_{k_0-1} f$ fixes $\text{tape-out}^{T_1}(i_{k_0})$. After that, whenever a finite sequence was concatenated onto $s_{-\exists}$, it was disjoint from $f_1 \ldots f_{k_0-1} f$. So by Lemma 3.6, f_n^* fixes $\text{tape-out}^{T_1}(i_{k_0})$. By Lemma 3.5, $\text{tape-out}_{\omega}^{T_1(s_{-\exists})}(i_{k_0}) = \text{tape-out}^{T_1(f_n^*)}(i_{k_0})$.

Now let j be an element of $\text{Unfixable}_{i_{k_n}}$. For concreteness assume that j was placed in the set Unfixable during the k_0^{th} iteration of the "For k" loop. For every positive integer k, f_k was formed by concatenating several finite sequences. If k is greater than or equal to k_0, then one of these sequences, f', was chosen so that f' changes $\text{tape-out}^{T_1(f_1 \ldots f_{k-1})}(j)$. Thus, by Lemma 3.4, f_k changes $\text{tape-out}^{T_1(f_1 \ldots f_{k-1})}(j)$. This is true for infinitely many values of k, so by Lemma 3.2, $\text{tape-out}_{\omega}^{T_1(s_{-\exists})}(j)$ = "blur".

Once again, let n be a positive integer. In order to show that the sequence $s_{-\exists}$, just constructed, witnesses the uniform ambiguity of T_1, we must provide an n-cowitness sequence, $s_{\exists}$. We must do this for every positive integer n.

Thus, for each n, we must construct an $\exists$-sequence, $s_{\exists}$, such that

$$\text{tape-out}_{\omega}^{T_1(s_{\exists})}(i) = \text{tape-out}^{T_1(f_n^*)}(i) \quad \text{for all } i \in \text{Fixed}_{i_{k_n}}, \text{ and}$$

$$\text{tape-out}_{\omega}^{T_1(s_{\exists})}(j) = \text{"blur"} \quad \text{for all } j \in \text{Unfixable}_{i_{k_n}},$$

where i_{k_n} and f_n^* are related to n as described on page 89.

Now we fix the positive integer n, and present an algorithm to construct the n-cowitness sequence, $s_{\exists}$:

Construction of the n-cowitness $s_{\exists}$:

```
var T1: augmented copying Turing machine
    f_n*: finite sequence of positive integers
    Fixed_{i_{k_n}},
    Unfixable_{i_{k_n}}: finite sets of positive integers
Program CONSTRUCTION-OF-s∃
  var s∃: sequence of positive integers
      m,p: integers
      f_p,f_p': finite sequences of positive integers
  {initialize}
  Let s∃:= f_n*
  Let m:= 1 + the largest integer that occurs in f_n*
```

For p:= 1 toward ω

{change the content of all the Unfixable squares}

For each $j \in \text{Unfixable}_{i_{k_n}}$

Choose f_p', disjoint from $s_{\exists}$, such that

f_p' changes $\text{tape-out}^{T(s_{\exists})}(j)$

{enlarge the sequence $s_{\exists}$}

Let $f_p := f_p'm$

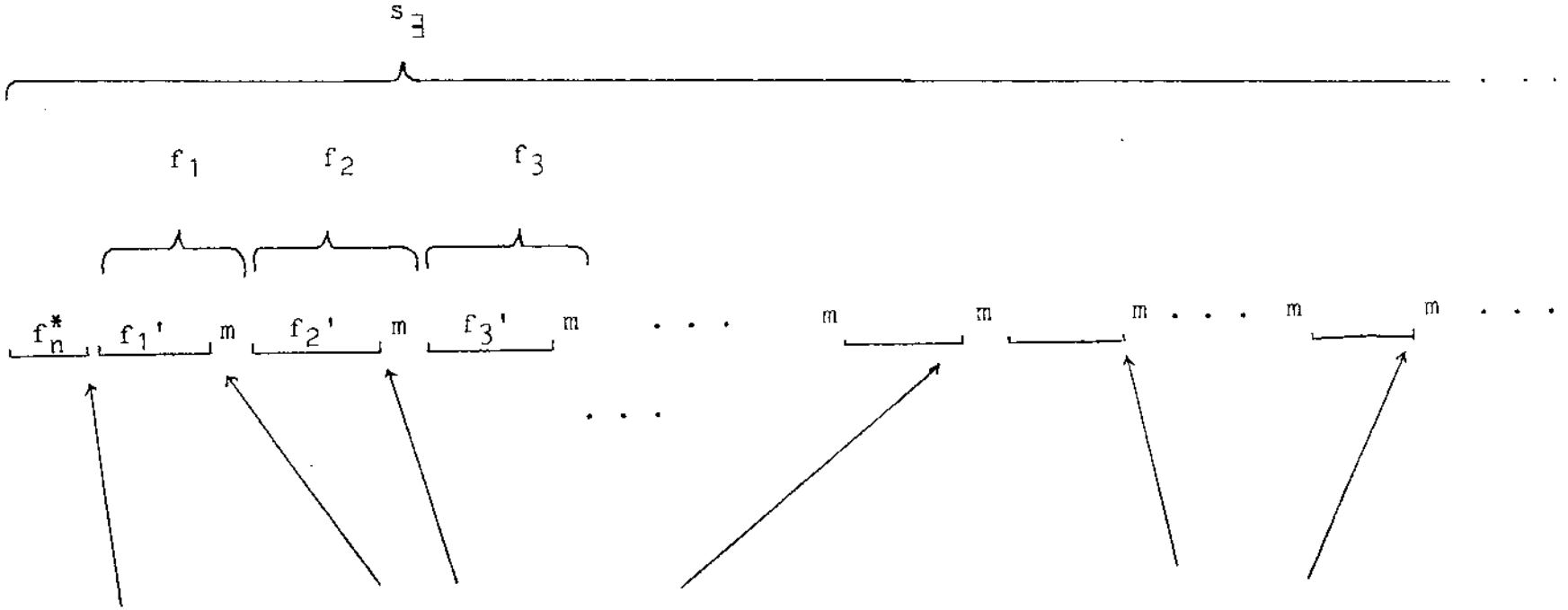
Let $s_{\exists} := s_{\exists}f_p$

The above algorithm is designed to construct a sequence, $s_{\exists}$, which forces $\text{tape-out}_{\omega}^{T_1}(j)$ to be "blur" for every $j \in \text{Unfixable}_{i_{k_n}}$. Sequence $s_{\exists}$ also forces $\text{tape-out}_{\omega}^{T_1}(i)$ to be $\text{tape-out}^{T_1(f_n^*)}(i)$ for each $i \in \text{Fixed}_{i_{k_n}}$. The action of T_1, upon examining $s_{\exists}$, is illustrated in the figure on the next page.

As in the construction of $s_{\sim\exists}$, we must show

(2.3.1) that the algorithm to construct $s_{\exists}$ will not halt unexpectedly, and

(2.3.2) that, for each n, the algorithm constructs an infinite $\exists$-sequence, $s_{\exists}$, which is an n-cowitness, with $s_{\sim\exists}$, to the n-ambiguity of T_1.



When T_1 finishes reading f_n^*, the content of $\text{tape-out}^{T_1}(i)$ is fixed at $\text{tape-out}^{T_1(f_n^*)}(i)$ for each $i \in \text{Fixed}_{i_{k_n}}$

For each $j \in \text{Unfixable}_{i_{k_n}}$ the content of $\text{tape-out}^{T_1}(j)$ gets changed

For each $j \in \text{Unfixable}_{i_{k_n}}$ the content of $\text{tape-out}^{T_1}(j)$ gets changed again

Proof of (2.3.1). We must show that we can always find a finite sequence, f_p', disjoint from $s_\exists$, such that f_p' changes tape-out$^{T_1}(s_\exists)_{(j)}$.

Assume that CONSTRUCTION-OF-$s_\exists$ has progressed to the point where p has the value p_0 and j has the value j_0. A finite sequence, $s_\exists$, has been constructed from the previous iteration of the "For each $j \in$ Unfixable$_{i_{k_n}}$" loop. The sequence f_n^* is an initial segment of $s_\exists$. In fact, $s_\exists$ is of the form $f_n^* f_1 f_2 \ldots f_{p_0-1} = f_n^* f_1' m f_2' m \ldots m f_{p_0-1}'$. Since m does not occur in f_n^*, and f_p' is always chosen to be disjoint from the sequence which has already been constructed, $f_1' m f_2' m \ldots m f_{p_0-1}'$ is disjoint from f_n^*. Thus $f_1 f_2 \ldots f_{p_0-1}$ is disjoint from f_n^*.

The sequence f_n^* was chosen to be a particular initial segment of $s_\exists$. By the construction of $s_\exists$, and the choice of f_n^*, we know that if f^{**} is a finite sequence which is disjoint from f_n^*, then $f_n^* f^{**}$ does not fix tape-out$^{T_1}(j_0)$. But $f_1 f_2 \ldots f_{p_0-1}$ is a finite sequence and is disjoint from f_n^*. So $f_n^* f_1 f_2 \ldots f_{p_0-1}$ does not fix tape-out$^{T_1}(j_0)$. So $s_\exists$ does not fix tape-out$^{T_1}(j_0)$. So there is at least one finite sequence, f_{p_0}', disjoint from $s_\exists$, such that f_{p_0}' changes tape-out$^{T_1}(j_0)$.

Proof of (2.3.2). We want to show that for each positive integer n, the sequence $s_\exists$, constructed by the algorithm, is

an n-cowitness with $s_{\exists}$ to the n-ambiguity of T_1. To do this we must show that

(2.3.2.1) $s_{\exists}$ is an infinite sequence,

(2.3.2.2) $s_{\exists}$ is an $\exists$-sequence, and

(2.3.2.3)

$$\text{tape-out}_{\omega}^{T_1(s_{\exists})}(i) = \text{tape-out}^{T_1(f_n^*)}(i) \quad \text{for all } i \in \text{Fixed}_{i_{k_n}}, \text{ and}$$

$$\text{tape-out}_{\omega}^{T_1(s_{\exists})}(j) = \text{"blur"} \quad \text{for all } j \in \text{Unfixable}_{i_{k_n}}.$$

Proof of (2.3.2.1). Similar to the proof of (2.1).

Proof of (2.3.2.2) Notice that the integer m occurs infinitely many times in $s_{\exists}$.

Proof of (2.3.2.3). Let $i \in \text{Fixed}_{i_{k_n}}$. The sequence $s_{\exists}$ is equal to $f_n^* f_1 f_2 \ldots$ where $f_1, f_2, f_3, \ldots$ are all disjoint from f_n^*. So by Lemma 3.5,

$$\text{tape-out}_{\omega}^{T_1(s_{\exists})}(i) = \text{tape-out}^{T_1(f_n^*)}(i).$$

Let $j_0 \in \text{Unfixable}_{i_{k_n}}$. In the construction of $s_{\exists}$, in the "For each $j \in \text{Unfixable}_{i_{k_n}}$" loop, when j takes on the

value j_0, the segment of $s_\exists$ that has been constructed thus far is finite, and we choose a finite sequence, f_p', which changes tape-out$^{T_1(s_\exists)}(j_0)$. We concatenate f_p' onto $s_\exists$ in constructing $s_\exists$. This happens infinitely often in the construction of $s_\exists$, because of the "For p" loop. Thus, by Lemma 3.2, tape-out$_\omega^{T_1(s_\exists)}(j_0)$ = "blur".

This completes the proof of Lemma 3.9. □

We now present the main result of this paper.

Theorem 2. Let T_1,T_2 be an infinite Turing machine which solves Problem 2. Then the worst case running time of T_1,T_2 is $\omega 2$.

Proof. By Lemma 3.7, it is sufficient to consider the case where T_1 is a copying and counting machine. Then, by Lemma 3.9, T_1 is uniformly ambiguous. By Lemma 3.8, the worst case running time of T_1,T_2 is $\omega 2$. □

This completes the proof of the main result of this paper. The result indicates that the combination of an augmented Turing machine and a finite Turing machine, powerful though it may be, cannot decide whether or not an infinite sequence of positive integers converges. Thus the worst case running time for a machine that solves Problem 2 is at least $\omega 2$. On page 34, and again on page 63, we

stated, but did not prove, that this combination of machines cannot find the limit of an infinite sequence of rationals. We can now prove that result.

Corollary. Let T_1,T_2 be an infinite Turing machine which solves Problem 1. Then the worst case running time of T_1,T_2 is $\omega 2$.

Proof. Assume the contrary - that the worst case running time of T_1,T_2 is ω. Let $s = \{s_i\}_{i=1}^{\omega}$ be a sequence of positive integers. For i even, let $x_i = 1$ and $y_i = s_{(i/2)}$. For i odd, let $x_i = 0$ and $y_i = 1$. Then $\{x_i/y_i\}_{i=1}^{\omega}$ converges iff s goes to infinity. Give $\{x_i\}_{i=1}^{\omega}$, $\{y_i\}_{i=1}^{\omega}$ to T_1,T_2. After ω plus finitely many steps T_1,T_2 reports that $\{x_i/y_i\}_{i=1}^{\omega}$ converges or does not converge. This can be interpreted as an answer about s, contradicting Theorem 2. □

3.4 One further result

As a final result on infinite Turing machines we will show that ωn is "infinite-time constructible" for every positive integer n.

Definition. Let $T_1, \ldots, T_n$ be an infinite Turing machine. We say that $T_1, \ldots, T_n$ halts on input s iff, given input s, the machine $T_1, \ldots, T_n$ performs less than ωn steps. The

ωn-halting problem is then stated as follows: Given an infinite Turing machine T_1, ..., T_n and an input s, does T_1, ..., T_n halt on s?

Note that the $\omega 1$-halting problem is almost the standard halting problem for finite Turing machines. The only difference is that T_1 is an augmented Turing machine, so it can have infinitely many non-blank squares on its input tape.

Theorem 3. Let H_n be an infinite Turing machine which solves the ωn-halting problem. Then H_n has worst case running time ωn.

Proof. Let ωt_n be the worst case running time of H_n.

First we argue that $t_n \leq n$. Construct H_n as in Example 1 of Section 1. I.e., given a machine T_1, ..., T_n and input s, let H_n write "does-not-halt" on its output tape, and then simulate the action of T_1, ..., T_n on s. If T_1, ..., T_n halts before time ωn, then H_n erases "does-not-halt" and writes the "halts" symbol on its output tape. This symbol is the output of H_n at time ωn.

Next we use a variant of the standard halting problem argument to show that $t_n \geq n$. Assume the contrary. Then H_n always halts before performing ωn steps. Define a machine,

H_n', which simulates H_n up to the point when H_n halts. At that point H_n' does the opposite of what is written on its output tape. (I.e., if H_n said "halts", then H_n' continues running until time ωn; if H_n said "does-not-halt", then H_n' halts.) It is easily seen that the act of providing H_n' with itself as input leads to a contradiction. □

References

[1] B. Burd, "Decomposable Collections of Sets", Notre Dame Journal of Formal Logic 25, 1984, pp.17-26.

[2] D. E. Muller, "Infinite Sequences and finite machines", AIEE Proc. Fourth Annual Symp. Switching Circuit Theory and Logical Design, 1963, pp. 3-16.

[3] J. R. Buchi, "On a decision method in restricted second order arithmetic", Proc. Internat. Congr. Logic, Method. and Philos. Sci. 1960, Stanford Univ. Press, Stanford, California, 1962, pp. 1-11.

[4] R. McNaughton, "Testing and generating infinite sequences by a finite automaton", Information and Control 9 1966, pp. 521-530.

[5] M. O. Rabin, "Decidability of second-order theories and automata on infinite trees", Trans. AMS 141, 1969, pp. 1-35.

[6] Z. Bavel, Introduction to the Theory of Automata, 1983, Reston Publishing Co., Reston, Virginia.

[7] H. Rogers, Jr., Theory of Recursive Functions and Effective Computability, 1967, McGraw-Hill, New York, p. 347.

[8] S. C. Kleene, "Recursive functionals and quantifiers of finite types I", Trans. AMS 91, 1959, pp. 1-52; and "... II", Trans. AMS 108, 1963, pp. 106-142.

[9] H. G. Rice, "Recursive real numbers", Proc. Amer. Math. Soc. 5, 1954, pp. 784-791.

[10] A. Mostowski, "Computable sequences", Fundamenta Mathematicae 44, 1957, pp. 37-51.

[11] A. H. Lachlan, "Recursive real numbers", J. Symbolic Logic 28, 1963, pp 1-16.

VITA

Barry Abram Burd

1971	A.B. in Mathematics, Temple University, Philadelphia, Pennsylvania.
1976	Ph.D. in Mathematics, University of Illinois, Urbana, Illinois.
1976-77	Assistant Professor, Department of Mathematics, Syracuse University, Syracuse, New York.
1977-80	Instructor, Department of Mathematical Sciences, Alverno College, Milwaukee, Wisconsin.
1980-83	Director of Academic Computing, Drew University, Madison, New Jersey.
1980-present	Assistant Professor, Department of Mathematics and Computer Science, Drew University, Madison, New Jersey.