The Complexity of Finite Memory Programs with Recursion

NEIL D. JONES

University of Aarhus, Aarhus, Denmark

AND

STEVEN S. MUCHNICK

The University of Kansas, Lawrence, Kansas

ABSTRACT In order to study the effects of recursion on the complexity of program analysis, a finite memory machine with recursive calls is defined, as well as two parameter passing mechanisms which extend the power of the language Close upper and lower bounds on the complexity of determining whether a program accepts the empty language are given for each of the three program models It is shown that such questions as acceptance of the empty set, equivalence, and so on are intractable even for these relatively simple programs

KEY WORDS AND PHRASES computational complexity, program analysis, program equivalence, recursive programs, pushdown automata, call by name, finite memory programs

CR CATEGORIES 5.22, 5.23, 5.24, 5 25, 5 27

1. *Introduction*

In [3] we studied the computational complexity of a number of questions of both programming and theoretical interest (e.g. halting, looping, equivalence) concerning the behavior of programs written in an extremely simple programming language. These finite memory programs or FMPs (pronounced "fumps") model the behavior of Fortran-like programs with a finite memory whose size can be determined by examination of the program itself. The main results of [3] are that determining halting, equivalence, looping, etc., are all of essentially the same complexity and that such analyses generally require nondeterministic algorithms with tape bounds at least proportional to the amount of memory which the program being analyzed can address, as a function of the size of the program. More precisely, if we define ACCEPT to be the set of all finite memory programs which halt and accept at least one input string, then one of our results is that

$$\text{ACCEPT} \in \text{NSPACE}(n) - \bigcup_{\epsilon>0} \text{NSPACE}(n^{1-\epsilon}).$$

Throughout the remainder of this paper we shall assume that the reader is familiar with the notation, terminology, and methods of [3], although any reader generally acquainted with complexity theory should be able to follow this paper with little difficulty.

A preliminary version of this work was presented at the Second ACM SIGACT-SIGPLAN Conference on Principles of Programming Languages, Palo Alto, California, January 1975

The work of N.D. Jones was partially supported by University of Kansas General Research Grant 3802-2038, the work of S S Muchnick was partially supported by University of Kansas General Research Grants 3758-5038 and 3803-x038

Authors' present address Department of Computer Science, The University of Kansas, Lawrence, KS 66045

The present paper is a continuation in which we extend the analysis to include Algol-like programs (called FMP^{REC}s) with the finite memory augmented by an implicit pushdown stack used to support recursion. Three variants are studied: one without formal parameters (equivalent to call-by-value parameters) and two with different varieties of call-by-name parameters. As might be expected, analysis of these programs is somewhat more complex, since the FMP is essentially a programmed generalized sequential machine, while a FMP^{REC} is a programmed deterministic pushdown transducer.

Our major results are the following. First, we show that at least deterministic exponential time is required to determine whether a program in the basic FMP^{REC} model accepts a nonempty set. Then we show that a model with a limited version of call by name requires exponential space to determine acceptance of a nonempty set, and that a more sophisticated model with rewritable conditional formal parameters has an undecidable halting problem. The same lower bounds apply to the equivalence problem, which in contrast to the situation for the basic FMP model is not known to be decidable (since it is not known whether equivalence of deterministic pushdown automata is decidable).

2. *The Basic Model*

A *finite memory program with recursion* (or FMP^{REC}) is defined to be a tuple

$$P = \langle P_0, \mathscr{X}_0, P_1, \mathscr{X}_1, \ldots, P_p, \mathscr{X}_p \rangle,$$

where $p \geq 0$ and

(1) $\mathscr{X}_0, \ldots, \mathscr{X}_p$ are disjoint sets of variable names ($\mathscr{X}_0$ is the set of *global variables* and for $i \geq 1$, $\mathscr{X}_i$ is the set of *local variables* for P_i).

(2) $P_0P_1 \cdot P_p$ is a sequence of labeled instructions,

$$1: I_1; 2: I_2; \ldots; k: I_k,$$

such that

(a) each P_i is a nonempty subsequence of instructions;

(b) each instruction in P_i for $0 \leq i \leq p$ has one of the following forms:

read X	**accept**
write V	**halt**
$X \leftarrow V$	**call** P_j
if $V_1 = V_2$ **goto** r	**return**

where $X \in \mathscr{X}_0 \cup \mathscr{X}_i$; V, V_1, and V_2 denote elements of $\mathscr{X}_0 \cup \mathscr{X}_i \cup \Sigma \cup \{\$\}$; $j \geq 1$; and r labels an instruction in P_i;

(c) the last instruction in any P_i is **accept**, **halt**, or **return**; and

(d) every variable in $\mathscr{X}_0 \cup \cdots \cup \mathscr{X}_p$ occurs at least once in $P_0P_1 \cdots P_p$.

The semantics of a FMP^{REC} is intended to mimic that of an Algol 60-style program of the form

```
begin symbol all variables in X0,
  procedure P1,
    begin symbol all variables in X1,
      body of P1
    end,

  procedure Pp,
    begin symbol all variables in Xp,
      body of Pp
    end,
  body of P0 (the "main program")
end
```

Note that the definition implies that **gotos** cannot cross procedure boundaries; that there are no calls on the outermost block; and that the end of each individual procedure is explicitly marked by a **halt**, **accept**, or **return** statement.

The semantics of the FMP^{REC} is a stack-oriented extension of that of the ordinary FMP. A *configuration* of a FMP^{REC} is a variable-length tuple (whose length is three more than twice the current depth of calls) of the form

$$\alpha = \langle x\$, i_0, y_0, i_1, y_1, \ldots, i_q, y_q \rangle,$$

where, for each j, if i_j labels an instruction in P_r then $y_{i_j} \in (\Sigma \cup \{\$\})^{|\mathscr{X}_r|}$. The *initial configuration* for an input string $x \in \Sigma^*$ is $\langle x\$, 1, \$^{|\mathscr{X}_0|} \rangle$

Intuitively, the currently executing instruction is that numbered i_q, and y_q is a vector containing the current values of the local variables of the procedure containing I_{i_q}. Further, for each j such that $0 \leq j < q$, i_j is a "return address" in some procedure P_r which has executed a call but which has not yet been returned to; the values of the local variables in P_r are concatenated to form y_j.

Now let α be a configuration as above and let $X \in \mathscr{X}_0 \cup \mathscr{X}_1 \cup \cdots \cup \mathscr{X}_p$. Let X be local to procedure P_k (so k is the unique index for which $X \in \mathscr{X}_k$). Further, let $\mathscr{X}_k = \{X_1, \ldots, X_m\}$ and let X be X_j. Then the *memory cell in α denoted by X* is defined to be the jth position in $y_r = a_1 \cdots a_m$ where r is the largest index such that i_q labels an instruction in P_k, and is undefined if no such r exists. We use $\alpha[X/b]$ to denote the configuration in which the memory cell denoted by X has the value b and the remainder of the configuration is identical to α. Similarly, $\alpha[n]$ denotes the configuration identical to α except that i_q is replaced by n. The *content function* con( ) is defined by

$$\text{con}(V, \alpha) = \begin{cases} V \text{ if } V \in \Sigma \cup \{\$\}, \\ a_j \text{ if } V \in \mathscr{X}_0 \cup \cdots \cup \mathscr{X}_p \text{ and symbol } a_j \text{ is in the memory cell in } \alpha \text{ denoted by } V. \end{cases}$$

Let α and β be two configurations, with α as above. We define $\alpha \vdash \beta$ (α *yields* β *directly*) to be true if and only if one of the following is true:

(1) I_{i_q} has the form "**read** X" and $\beta = \alpha[X/b][i_q + 1]$, where $b \in \Sigma \cup \{\$\}$ is such that either $x = zb$ or $x = z = \epsilon$ and $b = \$$,

(2) I_{i_q} has the form "**write** X" and $\beta = \alpha[i_q + 1]$;

(3) I_{i_q} has the form "$X \leftarrow V$" and $\beta = \alpha[X/\text{con}(V, \alpha)][i_q + 1]$;

(4) I_{i_q} has the form "**if** $V_1 = V_2$ **goto** r" and either $\text{con}(V_1, \alpha) = \text{con}(V_2, \alpha)$ and $\beta = \alpha[r]$, or $\text{con}(V_1, \alpha) \neq \text{con}(V_2, \alpha)$ and $\beta = \alpha[i_q + 1]$;

(5) I_{i_q} has the form "**call** P_j" and $\beta = \langle x\$, i_0, y_0, i_1, y_1, \ldots, i_q + 1, y_q, i_{q+1}, y_{q+1} \rangle$ where i_{q+1} labels the first instruction of P_j and $y_{q+1} = \$^{|\mathscr{X}_j|}$;

(6) I_{i_q} is "**return**" and $\beta = \langle x\$, i_0, y_0, i_1, y_1, \ldots, i_{q-1}, y_{q-1} \rangle$.

As usual, $\vdash^*$ denotes the reflexive transitive closure of $\vdash$. We say P *accepts* (*halts* for) an input $x \in \Sigma^*$ if and only if the initial configuration for x yields an accepting (halting) configuration, i.e. a configuration in which I_{i_q} is **accept (halt)**. $L(P)$ denotes the set of all $x \in \Sigma^*$ such that P accepts x.

The *output function* $P: \Sigma^* \to \Sigma^*$ is defined as follows. If $\alpha_1 \vdash \alpha_2 \vdash \cdots \vdash \alpha_p$, where α_1 is the initial configuration of P for x, and α_p is an accepting configuration, then $P(x) = b_1 b_2 \cdots b_p$, where for each i, if $\alpha_i = \langle y\$, i_0, y_0, \ldots, i_q, y_q \rangle$ and the current instruction I_{i_q} has the form "**write** V," then $b_i = \text{con}(V, \alpha_i)$ and $b_i = \epsilon$ otherwise. $P(x)$ is undefined for $x \notin L(P)$.

Finally, we define ACCEPT^{REC} by

$$\text{ACCEPT}^{REC} = \{P \in \text{FMP}^{REC} \mid L(P) \neq \emptyset\}.$$

Note that the representations and lengths of programs will be as for FMPs (see [3]).

3. *Complexity Properties of the* FMP^{REC}

Clearly a FMP^{REC} is computationally more powerful than a FMP, since it can accept a nonregular set. In fact a FMP^{REC} may be viewed as a programmed deterministic pushdown transducer (dpdt), as shown in the following lemma:

LEMMA 1. *For any* $P \in$ FMP^{REC} *there is a deterministic pushdown transducer* M *equivalent to* P *(i.e.* $P(x) = M(x\$)$ *for all* $x \in \Sigma^*$*), and conversely.*

PROOF. Consider a variant of the nonrecursive FMP which is augmented by an explicit stack and the instructions "**push** V" and "**pop** X" with the natural semantics (**push** V adds con(V, α) to the top of the stack and **pop** X removes the topmost stack symbol and places it in the memory cell denoted by X). Such a machine model (which we shall call a FMP^{STACK}) is clearly equivalent in power to a dpdt, since they differ only in how the finite control is specified.

It is, further, quite clear that it is possible to construct for an arbitrary $P \in$ FMP^{REC} an equivalent FMP^{STACK} by standard methods involving storing the "return addresses" on the stack as bit sequences with multiway branches used to simulate the effect of the **return** instruction. Thus constructing an equivalent dpdt, given a FMP^{REC}, is easy.

The other direction of the theorem is a bit more complex.

Suppose we are given a dpdt M presented as a FMP^{STACK}. Let the instructions of M be labeled 1, 2, ... , k in sequence and let the variables in M be named X_1, ... , X_m. We define the FMP^{REC}

$$P = \langle P_0, \{AUX, F_1, \ldots, F_k, X_1, \ldots, X_m\}, P_1, \{TOP\} \rangle,$$

where P_0 and P_1 are given by the following Algol 60-style program[1] with $a \in \Sigma$ chosen arbitrarily and # used as the bottom of stack marker:

```
begin symbol AUX, F1, . . , Fk, X1, . . , Xm,
  procedure P1,
    begin symbol TOP,
      TOP ← AUX,
      do forever
        case
      F1 = a   F1 ← $; J1,
      F2 = a:  F2 ← $, J2,

      Fk = a·  Fk ← $, Jk,
        end case
    end,
F1 ← a, AUX ← #,
call P1, halt
end
```

The instruction segment J_i is obtained from the ith instruction of M as follows:

ith instruction of M	J_i in P_1
read X	**read** X, $F_{i+1} \leftarrow a$
write V	**write** V, $F_{i+1} \leftarrow a$
if $V_1 = V_2$ **goto** r	**if** $V_1 = V_2$ **then** $F_r \leftarrow a$ **else** $F_{i+1} \leftarrow a$
accept	**accept**
halt	**halt**
push V	$AUX \leftarrow V$, $F_{i+1} \leftarrow a$, **call** P_1
pop X	$X \leftarrow TOP$, $F_{i+1} \leftarrow a$, **return**

P simulates the operation of M by interpreting one instruction at a time, keeping track of the current instruction by means of the flags F_1, ... , F_k. Calls and returns simulate **pushes** and **pops** so that the current top of stack symbol is in the cell denoted by TOP in the current invocation of P_1. AUX is a global auxiliary cell used to update TOP when a **push** is performed.

It should be clear that P faithfully simulates M. □

Note that the use of global variables (e.g. AUX) is necessary in this construction, since

[1] In this and succeeding programs we use a number of high level dictions, such as **case** and **do forever**, because of their greater degree of structural clarity All can easily be translated to basic FMP statements.

it is known (Tixier [7] and Lomet [4]) that not all deterministic context-free languages are recognized if information cannot be passed back from a called submachine.

It follows from Lemma 1 that the result in [3], that equivalence of FMPs is computationally no more difficult than nonemptiness, is difficult to extend to FMP^{REC}s (and is perhaps not even true), since the decidability of equivalence for deterministic pushdown automata is an open question of long standing.

THEOREM 1. *There is a constant $c > 0$ such that*

$$\text{ACCEPT}^{REC} \in DTIME(2^{cn}).$$

PROOF. Let P be a FMP^{REC} with k instructions and m variables, and let $s = |\Sigma \cup \{\$\}|$. The natural construction of a dpdt M equivalent to P (as discussed in the proof of the preceding lemma) results in a machine with $k + s$ pushdown stack symbols and at most $k \ s^m$ states. Clearly $k, m \leq \text{len}(P)$, so for some fixed $a > 0$ (independent of P), M will have no more than $2^{a \ \text{len}(P)}$ states. Also, there is a constant $b > 0$ such that a representation of M can be constructed in at most $2^{b \ \text{len}(P)}$ steps.

Now $L(M) \neq \emptyset$ iff $L(P) \neq \emptyset$ and testing whether $L(M) \neq \emptyset$ can be carried out in time polynomial in len(M), e.g. by first building a context-free grammar from M and then testing it as in [2]. Whether $L(P) \neq \emptyset$ (or equivalently $P \in \text{ACCEPT}^{REC}$) can be determined by constructing M and then testing $L(M) \neq \emptyset$, with a total time bound of $2^{c \ \text{len}(P)}$ for some $c > 0$ for all $P \in \text{FMP}^{REC}$. □

THEOREM 2. $\text{ACCEPT}^{REC} \notin DTIME\ (2^{dn/\log n})$ *for some* $d > 0$.

PROOF. Let $L \subseteq \Sigma^*$ be an arbitrary set in DTIME(2^n). Cook [1] has shown that there is a deterministic auxiliary pushdown automaton Z which recognizes L in a linear storage bound (such a machine is merely a linear-space-bounded Turing machine with an auxiliary stack). Without loss of generality we assume Z has $\Sigma \cup \{\$\}$ as both its tape and stack alphabets and operates within tape size gn for some integer $g > 0$. Let Z be presented as a program $I_1; \ldots; I_k$ with instructions of the types **write** s, **right**, **left**, **if** s **goto** l, **push** s, **pop**, **accept**, and **halt** (where $s \in \Sigma \cup \{\$\}$) with the usual interpretations (**pop** moves the topmost symbol on the pushdown stack onto the tape square being scanned).

Given an arbitrary $x = a_1 a_2 \cdots a_n \in \Sigma^*$ we shall construct a FMP^{REC} $f_Z(x)$ such that the computation performed by $f_Z(x)$ will (regardless of its own input) directly simulate the computation of Z on input x. Also, the length of $f_Z(x)$ will be $O(n \log n)$. As in the proof of Lemma 1, flags $F_1, \ldots, F_k$ are used to record which instruction of Z is currently being simulated. The variables $T_1, \ldots, T_{gn}$ are used to contain the symbols on the auxiliary storage tape of Z and flags $POS_1, \ldots, POS_{gn}$ are used to identify which tape square is currently being scanned. The form of the program $f_Z(x)$ (with $\#$ used as a bottom of stack marker) is as follows:

```
begin symbol AUX, F1, ..., Fk, T1, ..., Tgn, POS1, ..., POSgn;
  procedure P1,
    begin symbol TOP;
      TOP ← AUX,
      do forever
        case
      F1 = a   F1 ← $, J1,
        .
      Fk = a   Fk ← $; Jk
        end case
    end,
  F1 ← a; POS1 ← a; AUX ← #,
  T1 ← a1,   Tn ← an,
  call P1,
  halt
end
```

where the correspondence between I_t in Z and J_t in P_1 is given by the following table:

I_i in Z	J_i in P_1
accept	**accept**
halt	**halt**
right	**case** $POS_1 = a$ $POS_1 \leftarrow \$, POS_2 \leftarrow a,$ $POS_{gn-1} = a$ $POS_{gn-1} \leftarrow \$, POS_{gn} \leftarrow a,$ $POS_{gn} = a$ **halt** **end case** $F_{i+1} \leftarrow a$
left	similar to translation of **right**
write s	**case** $POS_1 = a$ $T_1 \leftarrow s,$ $POS_{gn} = a$ $T_{gn} \leftarrow s$ **end case** $F_{i+1} \leftarrow a$
if s **goto** r	**case** $POS_1 = a$ $AUX \leftarrow T_1,$ $POS_{gn} = a$ $AUX \leftarrow T_{gn}$ **end case** **if** $AUX = s$ **then** $F_r \leftarrow a$ **else** $F_{i+1} \leftarrow a$
push s	$AUX \leftarrow s, F_{i+1} \leftarrow a$; **call** P_1
pop	**case** $POS_1 = a$ $T_1 \leftarrow AUX$ $POS_{gn} = a$ $T_{gn} \leftarrow AUX$ **end case** $F_{i+1} \leftarrow a,$ **return**

Now note that there must be constants $b \geq 1$ and $c > 0$ such that for any x,

$$bn \leq \text{len}(f_Z(x)) \leq cn \log n$$

(where $n = |x|$). The $\log n$ factor accounts for representing the subscripts of the F_i, T_i, and POS_i variables. Further, it is clear that $f_Z(x)$ can be constructed in polynomial time, say $p(n)$; that $f_Z(x)$ faithfully simulates Z operating on input x; and that $f_Z(x) \in \text{ACCEPT}^{\text{REC}}$ if and only if Z accepts x.

Thus the question of whether x is a member of L can be solved by first calculating $f_Z(x)$ and then determining whether it is a member of $\text{ACCEPT}^{\text{REC}}$. Now suppose that for all $d > 0$ we have $\text{ACCEPT}^{\text{REC}} \in \text{DTIME}(2^{dn/\log n})$. Then the question "is x in L?" could be answered in time bounded by $p(n) + 2^{dm/\log m}$, where $m = \text{length}(f_Z(x))$, by the method just outlined. By the previous inequality we have

$$\frac{dm}{\log m} \leq \frac{dcn \log n}{\log b + \log n} \leq \frac{dcn}{1 + \log b/\log n} \leq dcn.$$

Thus for any $d > 0$, membership in L could be determined in time bounded by $p(n) + 2^{dcn}$. If we choose $d = 1/2c$, this implies that $\text{DTIME}(2^n) \subseteq \text{DTIME}(p(n) + 2^{n/2})$, which contradicts Theorem 10.11 of Hopcroft and Ullman [2]. Thus there exists $d > 0$ such that $\text{ACCEPT}^{\text{REC}} \notin \text{DTIME}(2^{dn/\log n})$, as required. □

4. *Models with Procedure Parameters*

The effect of call-by-value (and call-by-result) parameters can be achieved in the basic FMP^{REC} model by setting a collection of global variables before executing a **call** and then copying their values into local variables immediately after procedure entry (and similarly

at a **return** for call by result). We shall next consider two versions of call by name and establish their complexity properties.

For the sake of simplicity we avoid giving formal definitions of the call-by-name models, relying instead on the reader's presumed familiarity with the semantics of Algol 60 to verify the validity of our constructions. We first show that a broad interpretation of call by name results in an undecidable halting problem, and consequently in emptiness, equivalence, and similar problems being unsolvable as well. The proof of this fact will again be a simulation argument.

Let QUEUE denote the programming system with a single memory cell X, an implicit queue data structure, and the instruction types

enqueue a—copy the symbol a onto the right end of the queue
dequeue X—remove the leftmost symbol from the queue and copy it into X
if $X = a$ **goto** r where $a \in \Sigma$
halt

We assume that $\Sigma = \{1, 2, \ldots, n\}$ here for some fixed $n \geq 2$. It is known that the problem

given an arbitrary program P in QUEUE,
to determine whether P will eventually halt (by executing a **halt** instruction or emptying its queue)

is recursively undecidable. This can be shown by simulating tag systems [5] in QUEUE, since we know that tag systems in turn can faithfully simulate Turing machines.

THEOREM 3. *The halting problem for finite memory programs with recursion and call by name is recursively undecidable.*

Construction Let $P = 1$: I_1; ... ; k: I_k be any program in QUEUE with variables $X_1, \ldots, X_m$. From P we construct the Algol-like program which follows. The program uses the **integer** data type for simplicity; the same effect could also be achieved by a larger program with only the **symbol** data type, as used by the FMP^{REC}.

```
begin integer F, X,
  procedure P(A, B, C), value C, integer A, B, C,
    begin
      do forever
        case
      F = 1   J1,
        .
      F = k   Jk
        end case
    end
  F ← 1,
  call P(0, −1, −1)
end
```

where the instructions are encoded as follows:

I_i	J_i
halt	**halt**
if $V_1 = V_2$ **goto** r	**if** $V_1 = V_2$ **then** $F \leftarrow r$ **else** $F \leftarrow i + 1$
enqueue a	$F \leftarrow i + 1$; **call** P(**if** $B < 0$ **then** C **else** A, C, a), **halt**
dequeue	**if** $B < 0$ **then halt**, $X \leftarrow A$, $A \leftarrow -A$, $F \leftarrow i + 1$

It is assumed here that a is positive whenever **enqueue** a is performed.

DISCUSSION OF THE CONSTRUCTION. The above construction involves a rather subtle use of call by name, as we shall now illustrate. Suppose the QUEUE program being simulated has performed the sequence of instructions

enqueue X_1,
enqueue X_2,
enqueue X_3

Then our simulator will be in the state diagramed in Figure 1 (recall that call-by-name parameters do not have immediate values, but rather are pointers to expressions which were bound to memory cells or other parameters at the time of their call). The arrows in the figure indicate these parameter bindings.

The successive values of C in the stack will contain all the elements ever added to the queue, with the most recently added ones nearest the top. Those elements which have been dequeued appear as negative numbers. Execution of the statement "$X \leftarrow A$" in the simulator causes evaluation of the expression bound to A. This causes "**if** $B < 0$ **then** C **else** A" to be repeatedly evaluated, with B and C successively bound to elements lower and lower in the stack, until at last a negative value is obtained for B. The value finally returned (and stored in X) will be that positive value of C occurring lowest in the stack, namely X_1 in the given example. Execution of "$A \leftarrow -A$" then causes the same C value to be made negative, effectively removing it from the queue.

PROOF OF THEOREM 3. It should be clear that the constructed program faithfully simulates the actions of P and halts just in case P does. Halting of programs in QUEUE is undecidable, as pointed out above, since they can faithfully simulate tag systems. Consequently any class of finite memory programs which is capable of simulating the type of program we have constructed also has a recursively undecidable halting problem. But the set of values assumed by the variables in the construction is clearly fixed so that the class of FMPs with recursion and call by name is sufficient to carry out the indicated simulation. □

Note now that the program constructed above actually goes somewhat beyond the semantics of Algol 60, in that the statement "$A \leftarrow -A$" has essentially the effect of

$$(\textbf{if } B < 0 \textbf{ then } C \textbf{ else } A) \leftarrow (\textbf{if } B < 0 \textbf{ then } C \textbf{ else } A)$$

Although this is not legal according to the Algol 60 report [6], it is actually allowed in some implementations and should be clear to the reader.

Even if the implicit use of a conditional expression on the left-hand side of an assignment statement is prohibited, the halting problem remains quite intractable, as is shown by the following theorem.

THEOREM 4. *There is a* $d > 0$ *such that the* ACCEPT *problem for* FMPS *with recursion and call by name, but without assignment to conditional expression parameters, is not in* $NSPACE(2^{dn/\log n})$.

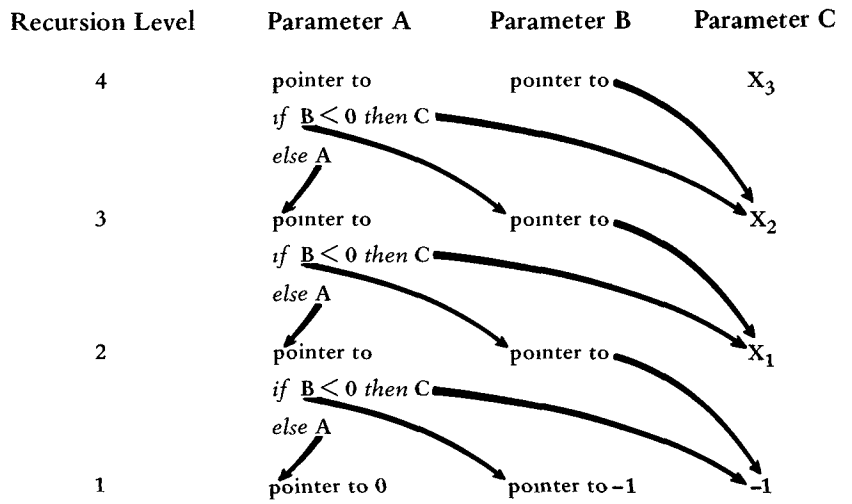

FIG 1 Stack contents during simulation in Theorem 4

PROOF. We omit most of the details since the same principle has been used in several previous theorems (e.g. Theorem 16 of [3]). In essence the technique is to construct, given a Turing machine Z which operates nondeterministically in space 2^n and an arbitrary string $x \in \Sigma^*$, a FMP $f_Z(x)$ of the required variety such that

(i) Z accepts x if and only if $L(f_Z(x)) \neq \emptyset$, and
(ii) $\text{len}(f_Z(x)) = O(|x| \log |x|)$.

The method used in [3] was to build $f_Z(x)$ to accept an input string just in case the input encoded an accepting computation by Z on input x. This required a memory capable of holding 2^n symbols.

Such a memory may be simulated by the use of call by name parameters as follows. Let *INDEX* denote an array of $n + 1$ variables, each containing a 0 or a 1. Such vectors can (and will) be interpreted as binary numbers in the range 0, 1, ... , $2^{n+1} - 1$. Now consider the program

```
begin symbol →INDEX, AUX,
  procedure P(A),
    begin symbol →KEY, DATA,
      if →INDEX < →2^n then
        begin
          →KEY ← →INDEX,
          read DATA,
          →INDEX ← →INDEX + →1,
          P(if →KEY = →INDEX then DATA else A)
        end
      else
        begin
          Q
        end
    end;
  →INDEX ← →0,
  P(0)
end
```

which reads in 2^n data values (hereafter called $X_0, X_1, \ldots, X_{2^n-1}$) while performing recursive calls and then executes the program segment Q. During the execution of Q the local variable stack will contain 2^n $\langle \overrightarrow{KEY}, DATA \rangle$ pairs, namely $\langle \vec{0}, X_0 \rangle$, $\langle \vec{1}, X_1 \rangle$, ... , $\langle \overrightarrow{2^n - 1}, X_{2^n-1} \rangle$, ordered with the last of them at the top of the stack, and the A parameters will be chained together in the same manner as in the proof of Theorem 3. Consequently *any* X_i value for $i = 0, 1, \ldots, 2^n - 1$ can be fetched within the program segment Q by setting $\overrightarrow{INDEX} \leftarrow \vec{i}$ and then fetching A, thus providing effectively random access to a read-only memory of size 2^n. Note also that further recursive calls to P can add to the end of this memory, but that only 2^n distinct data cells may be accessed.

Now the techniques used in the proof of Theorem 16 of [3] can be applied quite directly to yield the desired result. □

5. *Summary of Results*

We have considered a class of programs which bear roughly the same relationship to finite memory programs as Algol 60 does to Fortran, and have shown that analysis of their behavior is of extreme computational difficulty. The following table summarizes both upper and lower bounds for the complexity of the ACCEPT problem for each of the varieties considered in [3] and the present paper.

	Lower bound	Upper bound
ACCEPT	NSPACE($n^{1-\epsilon}$)	NSPACE(n)
ACCEPT^{NUM}	NSPACE($n^{1-\epsilon}$)	NSPACE(n)
ACCEPT^{VWS}	NSPACE($n^{2-\epsilon}$)	NSPACE(n^2)
ACCEPT^{ARRAY}	NSPACE($2^{dn/\log n}$)	NSPACE(2^{cn})
ACCEPT^{RFC}	DTIME($2^{dn/\log n}$)	DTIME(2^{cn})
ACCEPT^{NAMP} (restricted)	NSPACE($2^{dn/\log n}$)	?
ACCEPT^{NAML}	Undecidable	—

REFERENCES

1 COOK, STEPHEN A Characterizations of pushdown machines in terms of time-bounded computers *J ACM 18*, 1 (Jan 1971), 4–18

2 HOPCROFT, JOHN E, AND ULLMAN, JEFFREY D *Formal Languages and Their Relation to Automata* Addison-Wesley, Reading, Mass, 1969

3 JONES, NEIL D, AND MUCHNICK, STEVEN S Even simple programs are hard to analyze *J ACM 24*, 2 (April 1977), 338–350

4 LOMET, DAVID B A formalization of transition diagram systems *J ACM 20*, 2 (April 1973), 235–257

5 MINSKY, MARVIN *Computation Finite and Infinite Machines* Prentice-Hall, Englewood Cliffs, N J, 1967

6 NAUR, PETER, et al. Revised report on the algorithmic language Algol 60. *Comm. ACM 6*, 1 (Jan. 1963), 1–17.

7 TIXIER, V Recursive Functions of Regular Expressions in Language Analysis Tech Rep CS-58, Comptr Sci. Dept, Stanford University, Stanford, Calif, March 1967

RECEIVED NOVEMBER 1976, REVISED SEPTEMBER 1977