The Complexity of the Equivalence Problem for Simple Programs

EITAN M. GURARI

SUNY at Buffalo, Amherst, New York

AND

OSCAR H. IBARRA

University of Minnesota, Minneapolis, Minnesota

ABSTRACT The complexity of the equivalence problem for several simple programming languages is investigated. In particular, it is shown that a class of programs, called XL, has an NP-complete inequivalence problem; hence its equivalence problem is decidable in deterministic time $2^{p(N)}$, where $p(N)$ is a polynomial in the sum of the sizes of the programs. This bound is a four-level exponential improvement over a previously known result A very simple subset of XL, called SL, is also considered, and it is shown that every XL-program is polynomial-time reducible to an equivalent SL-program. Moreover, SL is minimal in the sense that all its instructions are independent On the other hand, XL is maximal in that a "slight" generalization yields a language with an undecidable equivalence problem. XL-programs realize precisely the relations (functions) definable by Presburger formulas.

KEY WORDS AND PHRASES: simple programming languages, Presburger formulas, counter machines, equivalence problem, computational complexity, polynomial space, polynomial time, NP-complete

CR CATEGORIES: 5 21, 5.22, 5.24, 5 25, 5.26, 5 27

1. *Introduction*

Let XL be the programming language which has the following instruction set:

(1) $x \leftarrow x + 1$
(2) $x \leftarrow x \dot{-} 1$
(3) **if** $x = 0$ **then exit**
(4) **loop**
(5) **do** x
$\vdots$
end
(6) $x \leftarrow 0$
(7) $x \leftarrow y$
(8) **goto** l
(9) **if** $x = 0$ **then goto** l

This research was supported by the National Science Foundation under Grant MCS 78-01736.

Authors' present addresses E M Gurari, Department of Computer Science, SUNY at Buffalo, 4226 Ridge Lea Road, Amherst, NY 14226; O H. Ibarra, Department of Computer Science, University of Minnesota, 136 Lind Hall, Minneapolis, MN 55455.

An XL-program P is any finite nonempty sequence of instructions of the form (1)–(9) satisfying the following conditions:

(R1) **do** ... **end** pairs only enclose instructions of the form (1)–(4) and (6)–(9). Thus, nested **do**'s are not permitted.
(R2) Labels in instructions of the form (8)–(9) are forward labels.
(R3) No instruction in the scope of a **do** ... **end** construct can be labeled. (The **do** itself can be labeled.)

Each program variable can hold any nonnegative integer. The variable x controlling the **do** x ... **end** construct can be modified inside the **do** without changing the number of iterations. The **if** $x = 0$ **then exit** can appear only inside a **do** ... **end** construct, and it causes an exit out of the **do** if $x = 0$. **loop** causes the program to go into an infinite loop. (This instruction is introduced to allow the definition of partial functions.) The program halts after processing the last instruction unless, of course, the last instruction is a **loop**. (It should be noted that a **halt** instruction can be simulated linearly in any language containing instructions of the form (1)–(3) and (5). Thus, for notational convenience, the **halt** instruction will also be used when appropriate.) Two fixed (not necessarily disjoint) sets of program variables are designated input variables and output variables, respectively. There must be at least one input variable. Before the start of program execution, all noninput variables are initialized to zero while the input variables are set to some input values. Assume that the program P has input variables $x_1, \ldots, x_{n_1}$ and output variables $y_1, \ldots, y_{n_2}$.

P can be used to define a relation or a function over the natural numbers. The relation defined by P is the set $S_P = \{(i_1, \ldots, i_{n_1}) \mid \text{each } i_j \text{ is a nonnegative integer, } P \text{ with } x_1, \ldots, x_{n_1} \text{ set to } i_1, \ldots, i_{n_1} \text{ halts}\}$. If $n_2 \geq 1$, then the (possibly partial) function defined by P is given by $f_P(i_1, \ldots, i_{n_1}) = (j_1, \ldots, j_{n_2})$ if P with $x_1, \ldots, x_{n_1}$ set to $i_1, \ldots, i_{n_1}$ halts with $j_1, \ldots, j_{n_2}$ as the values of the output variables; if P does not halt, $f_P(i_1, \ldots, i_{n_1})$ is undefined. A subset of XL consisting only of instructions (1)–(5) is called the SL language.

XL is a subset of the language SR_1 studied in [5] and is very similar to the L_+ language introduced in [3]. It was shown in [3] that L_+ realizes exactly the Presburger arithmetic, both functionally and relationally. As a by-product of this characterization, an upper bound of

$$2^{2^{2^{2^{p(N)}}}}$$

for the deterministic time complexity of the equivalence problem for L_+-programs was obtained in [3] ($p(N)$ is some polynomial in the sum of the sizes of the programs being considered). One of the main results in this paper is the following:

(a) The equivalence problem for XL-programs is decidable in deterministic polynomial space and, therefore, in time $2^{p(N)}$ for some polynomial $p(N)$. (Note that the time bound is a four-level exponential improvement over the result in [3].) In fact, the inequivalence problem for XL-programs is NP-complete. (See [7, 13] for the definition of an NP-complete problem.)

The proof of (a) consists of showing that every XL-program is polynomial-time reducible to an equivalent SL-program. Every SL-program is then shown to be polynomial-time reducible to an equivalent deterministic multicounter machine each of whose counters makes at most a fixed number of reversals. The result then follows from the recently established upper bounds on the complexity of the equivalence problem for counter machines [10]. The other main results of the paper are

(b) XL-programs are equivalent to SL-programs, and they realize exactly the Presburger formulas. However, constructing an XL-program realizing a Presburger formula requires superexponential space (in the size of the formula) for infinitely many formulas.
(c) SL is minimal in the sense that all its instructions are independent.
(d) XL is maximal in that dropping one of the conditions (R1)–(R3) yields a programming language with an undecidable equivalence problem. In particular, dropping (R3) results in a language, called UL, which is strictly stronger than XL, but strictly weaker than the loop language L_2 (=SR_1 [5]). (For $i \geq 0$, L_i is the loop language consisting only of the instructions $x \leftarrow 0$, $x \leftarrow x + 1$, $x \leftarrow y$, **do** $x \ldots$ **end** with at most i levels of **do** nestings [15].)

Notation. Throughout the paper, the abbreviations succ, pred, ifexit, loop, do, $x \leftarrow 0$, $x \leftarrow y$, goto, and ifgoto will also be used to denote XL-instructions of the form (1)–(9), respectively.

Remark. Note that, in general, $x \leftarrow 0$ cannot be simulated by the instructions pred and do. If $x \leftarrow 0$ were inside a **do** construct, then the simulation would result in nested **do**'s, violating restriction (R1).

The remainder of this section is devoted to definitions and notations needed in the paper.

Definition. Presburger formulas are defined inductively as follows (see, e.g., [9]):

(a) $a_0 + \sum_{i=1}^{m} a_i x_i = b_0 + \sum_{i=1}^{m} b_i x_i$ is a Presburger formula for every integer $m \geq 1$, variables $x_1, \ldots, x_m$, and nonnegative integers $a_0, a_1, \ldots, a_m, b_0, b_1, \ldots, b_m$.
(b) If F_1 and F_2 are Presburger formulas, then so are their conjunction $F_1 \wedge F_2$ and their disjunction $F_1 \vee F_2$.
(c) If F is a Presburger formula, then so is its negation $\neg F$.
(d) If x_i is a free variable in a Presburger formula F, then $(\exists x_i)F$ and $(\forall x_i)F$ are Presburger formulas.
(e) Only expressions derivable using rules (a)–(d) are Presburger formulas.

A Presburger formula with $m \geq 1$ free variables is denoted by $F(x_1, \ldots, x_m)$. The *size of a Presburger formula* is the length of its representation. Throughout, $\mathbb{N}$ denotes the set of natural numbers.

Definition

(1) Let $m \geq 1$. A subset $S \subseteq \mathbb{N}^m$ is a *Presburger set or relation* if there is a Presburger formula $F(x_1, \ldots, x_m)$ such that $S = \{(i_1, \ldots, i_m) \mid F(i_1, \ldots, i_m) \text{ is true}\}$.
(2) Let $n_1, n_2 \geq 1$. A (possibly partial) function $f: \mathbb{N}^{n_1} \to \mathbb{N}^{n_2}$ is a *Presburger function* if there is a Presburger formula $F(x_1, \ldots, x_{n_1}, y_1, \ldots, y_{n_2})$ such that $\{(i_1, \ldots, i_{n_1}, j_1, \ldots, j_{n_2}) \mid f(i_1, \ldots, i_{n_1}) = (j_1, \ldots, j_{n_2})\} = \{(i_1, \ldots, i_{n_1}, j_1, \ldots, j_{n_2}) \mid F(i_1, \ldots, i_{n_1}, j_1, \ldots, j_{n_2}) \text{ is true}\}$.

In Sections 2 and 3 we give a simple programming language characterization of Presburger relations (functions). The proof of the characterization uses a known connection between Presburger formulas and multicounter machines [10]. The complexity results in Sections 4 and 5 also rely on established results concerning the complexity of the equivalence problem for these devices.

We consider counter machines with a finite-state control and n counters, each capable of storing any nonnegative integer. At the start of a computation, the counter machine is set to a specified initial state and a subset of the counters, called *input counters*, is initialized to some "input values." The remaining counters are set to zero.

The input values are *accepted* by the device if the device eventually halts. The inputs are said to be *rejected* if the device does not halt. A subset (possible empty) of the counters are called *output counters*. The values in these counters if and when the machine halts are taken to be the output values. An atomic move consists of incrementing exactly one of the counters by +1 or −1 and changing the state of the finite-state control. In every computation a counter can alternately increase and decrease its value at most k times for some integer k. Thus each counter makes at most k reversals. The device is deterministic in that it may have at most one choice of next move on a given configuration. Thus a counter machine M can be described uniquely by a finite set of rules of the form $[q, i_1, \ldots, i_n, d_1, \ldots, d_n, p]$, where q is the current state, $i_1, \ldots, i_n$ are the *counter modes* (0 or $\neq 0$), $d_1, \ldots, d_n$ are the changes in the values of the counters (+1 or −1) satisfying $\mathrm{abs}(d_1) + \cdots + \mathrm{abs}(d_n) = 1$, and p is the next state. Since M is deterministic, no two rules have the same first $n + 1$ coordinates. Let $C(n, n_1, n_2, k)$ denote the class of n-counter machines with n_1 input counters and n_2 output counters such that each counter makes at most k reversals. The other counters are referred to as *auxiliary counters*. Note that some of the input counters can also be output counters. If M is a reversal-bounded counter machine, then the size of M, denoted by SIZE(M), is the length of the representation of M.

Remark. Our multicounter machines are different from the usual language-accepting counter machines (with input tapes) studied in several places in the literature (see, e.g., [2, 12]). Here we are concerned with relations and functions over the natural numbers computable by these devices.

Convention. In this paper a reversal-bounded counter machine means a machine in $C(n, n_1, n_2, k)$ for some n, n_1, n_2, and k.

Definition. Let M be a machine in $C(n, n_1, n_2, k)$, $n_1 \geq 1$, $n_2 \geq 0$, $k \geq 0$. The relation or set accepted by M is the set $S_M = \{(x_1, \ldots, x_{n_1}) \mid \text{each } x_i \text{ in } \mathbb{N}, M \text{ with its counters set to } x_1, \ldots, x_{n_1} \text{ eventually halts}\}$. If $n_2 \geq 1$, then the function defined by M is $f_M : \mathbb{N}^{n_1} \to \mathbb{N}^{n_2}$, where for $x_1, \ldots, x_{n_1}$ in $\mathbb{N}$, $f_M(x_1, \ldots, x_{n_1}) = (y_1, \ldots, y_{n_2})$ if M on input $x_1, \ldots, x_{n_1}$ eventually halts with $y_1, \ldots, y_{n_2}$ on its output counters; if M does not halt, $f_M(x_1, \ldots, x_{n_1})$ is undefined. A set $S \subseteq \mathbb{N}^{n_1}$ (respectively, function $f : \mathbb{N}^{n_1} \to \mathbb{N}^{n_2}$) is a *counter machine set* (respectively, *counter machine function*) if there is a machine M in $C(n, n_1, n_2, k)$ such that $S_M = S$ (respectively, $f_M = f$).

The following theorems proved in [10] show that counter machine sets (functions) are equivalent to Presburger sets (functions).

THEOREM 1. *Let $S \subseteq \mathbb{N}^m$. Then S is a Presburger set if and only if it is a counter machine set.*

THEOREM 2. *Let f be a function from $\mathbb{N}^{n_1}$ to $\mathbb{N}^{n_2}$. Then f is a Presburger function if and only if it is a counter machine function.*

Let P be a random access machine (RAM) program which uses only the arithmetic operations of multiplication, division, addition, and subtraction. Then P can be implemented on a multitape Turing machine whose time complexity (under the logarithmic cost criterion) is polynomial in the time complexity of the RAM program P [1]. In this paper, unless otherwise stated, the time complexities are for RAM programs using the logarithmic cost criterion.

2. *From Counter Machines to SL-Programs*

In this section and the next we show that the class of programs SL = {succ, pred, ifexit, loop, do} realizes precisely the relations and functions defined by Presburger formulas. For other programming language characterizations see [3, 4].

Notation. We denote by $\mathrm{SL}(n, n_1, n_2)$ the set of n-variable programs in SL having n_1 input variables and n_2 output variables. If P is a program, then the size of P, denoted by $\mathrm{SIZE}(P)$, is the length of the representation of P.

The proof that SL is an exact realization of Presburger formulas uses the machine characterization of Presburger relations and functions given in Theorems 1 and 2. We show in this section that every counter machine has an equivalent program in SL. The converse is proved in Section 3.

THEOREM 3. *Let $S \subseteq \mathbb{N}^{n_1}$ (respectively, $f: \mathbb{N}^{n_1} \to \mathbb{N}^{n_2}$) be a counter machine set (respectively, counter machine function). Then S (respectively, f) is computable by an SL-program.*

PROOF. Every counter machine M' whose counters are restricted to make at most a fixed number of reversals can be converted to an equivalent counter machine M whose counters are restricted to make at most one reversal (see, e.g., [2, 12]). (M simulates the computation of M'. However, when a counter of M', say counter j, is about to make a reversal, the value in the corresponding counter of M is copied into a new counter. Then the new counter is used to simulate the further changes in the jth counter of M'.) Thus it is sufficient to consider a counter machine M in $C(n, n_1, n_2, 1)$ for some n. At any given instant of the computation let V_i be the value in the ith counter of M, $1 \le i \le n$, and let the *status of counter i* be defined by the value $A_i + 2B_i$, where

$$A_i = \begin{cases} 0 & \text{if } V_i = 0, \\ 1 & \text{if } V_i \neq 0, \end{cases}$$

$$B_i = \begin{cases} 1 & \text{if } V_i \text{ equals } 0 \text{ after being positive}, \\ 0 & \text{otherwise.} \end{cases}$$

Counter i is in status 0 ($A_i = B_i = 0$) if the counter is 0 from the start of the computation. It is in status 1 ($A_i = 1$, $B_i = 0$) if the counter is positive and in status 2 ($A_i = 0$, $B_i = 1$) if it is 0 after being positive. A change in counter status occurs whenever a counter with 0 value becomes positive (status 0 changed to 1) and whenever a positive counter becomes 0 (status 1 changed to 2). The *counter status vector* (CSV, for short) is defined to be an n-tuple, $\alpha = [A_1 + 2B_1, \ldots, A_n + 2B_n]$. Thus at any given instant of the computation the CSV is given by the statuses of the counters at that instant.

Define the following relation R on the set of all CSVs: $(\alpha', \alpha'') \in R$ if and only if $\sum_{i=1}^{n} (A_i' + 2B_i') = \sum_{i=1}^{n} (A_i'' + 2B_i'')$, where $\alpha' = [A_1' + 2B_1', \ldots, A_n' + 2B_n']$ and $\alpha'' = [A_1'' + 2B_1'', \ldots, A_n'' + 2B_n'']$. Thus two CSVs are related by R if the sums of their counter statuses are equal. Clearly R is an equivalence relation which induces the equivalence partition $E_0, E_1, \ldots, E_{2n}$ on the class of CSVs such that

$$\alpha \in E_j \qquad \text{if and only if} \quad \sum_{i=1}^{n} (A_i + 2B_i) = j,$$

with α being the CSV $[A_1 + 2B_1, \ldots, A_n + 2B_n]$. Thus $E_0 = \{[0, \ldots, 0] \mid n \text{ 0's}\}$, $E_1 = \{[0, \ldots, 0, 1, 0, \ldots, 0] \mid k \text{ 0's followed by 1 followed by } n - k - 1 \text{ 0's}\}, \ldots,$ $E_{2n} = \{[2, \ldots, 2] \mid n \text{ 2's}\}$.

Initially M is in some CSV α_{i_0} contained in some equivalence class E_{j_0}. During the computation the machine goes through a sequence of configurations having the corresponding CSVs $\alpha_{i_0}, \ldots, \alpha_{i_l}$, where α_{i_k} is contained in the equivalence class E_{j_k}, $0 \le k \le l$. Moreover, $j_0 \le j_1 \le \cdots \le j_l$ and:

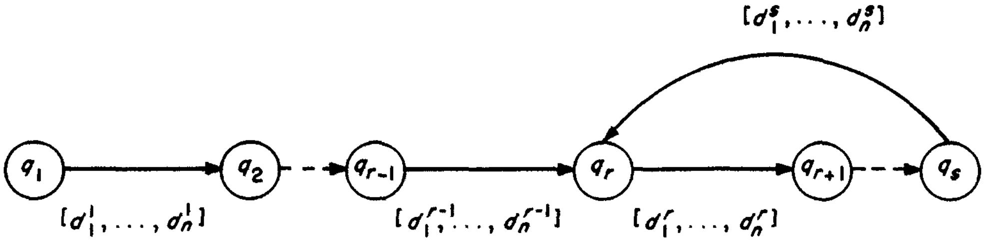

FIG. 1. A transition diagram describing a computation having a fixed CSV that is entered in state q_1. d_u^t is the value by which counter u is changed when an atomic move is made from state q_t.

(a) If $j_k < j_{k+1}$, then $E_{j_{k+1}} = E_{j_k+1}$, $0 \le k < l$. Recall that exactly one counter changes value in an atomic move. Hence $j_k \neq j_{k+1}$ if and only if either a zero counter becomes positive (the corresponding counter status changes from 0 to 1) or a positive counter becomes zero (the corresponding counter status changes from 1 to 2). In both cases the machine goes from a CSV in the equivalence class E_{j_k} to a CSV in the equivalence class $E_{j_{k+1}}$ $(=E_{j_k+1})$.
(b) If $j_k = j_{k+1}$, then $\alpha_k = \alpha_{k+1}$. This is because no change in counter status occurs.

A period of computation having a fixed CSV can be described by a transition diagram showing the transitions between states and the changes in the values of the counters (see Figure 1).

Assume that M has Q states. Now the machine is deterministic. So for any given CSV and state the machine has at most one choice of next move. If such a move exists, then the machine changes the value in exactly one of the counters and enters a new state. Moreover, after at most Q moves the machine enters a cycle of transitions which is repeated as long as the CSV remains unchanged. During a cycle each counter is monotonically nondecreasing or monotonically nonincreasing. This is because the counters are restricted to make at most one reversal. If during a cycle some counters of M are decreasing, then eventually a counter will reach the zero value and cause a change in CSV. If on the other hand M enters a cycle in which all the counters are nondecreasing, then it will never halt. We may assume by the construction below that the machine detects such a behavior, and once it does so, it enters a distinguished state, say q_{loop}. Therefore, in every computation M will eventually halt or enter the state q_{loop}. This is because a computation can be in at most $2n + 1$ distinct CSVs, and while in a given CSV, M eventually halts, enters the state q_{loop}, or enters a new CSV.

Construction. Given an n-counter machine M, modify M to obtain a new machine M' as follows (q_{loop} is a new state). For each rule $[q, i_1, \ldots, i_n, d_1, \ldots, d_n, p]$ of M, determine if M, when started in state q and counter modes $i_1, \ldots, i_n$, goes into an infinite loop; that is, M makes at most Q (=number of states of M) moves without decreasing a counter or changing a counter mode. If so, replace the rule by the rule $[q, i_1, \ldots, i_n, d_1, \ldots, d_n, q_{\text{loop}}]$; otherwise, retain the rule. Clearly, M' has the following properties:

(a) M' is equivalent to M on inputs for which M halts.
(b) M' enters the state q_{loop} if and only if M does not halt.
(c) $O(\text{SIZE}(M')) = O(\text{SIZE}(M))$.

We are now ready to show how an SL-program simulates the computation of M. The program uses variables to hold the values of the counters of M as well as some control variables to monitor the status of M during the computation:

(a) $V_1, \ldots, V_n$—hold the values of the counters of M. The input (respectively, output) variables are those which correspond to the input (respectively, output) counters of M. Initially the input variables contain the input values while the other variables are zero.
(b) $A_1, \ldots, A_n, B_1, \ldots, B_n$—each holding 0 or 1, depending on the current status of the corresponding counter of M.
(c) $\bar{A}_1, \ldots, \bar{A}_n, \bar{B}_1, \ldots, \bar{B}_n$—for $1 \le i \le n$, $\bar{A}_i = 1 - A_i$ and $\bar{B}_i = 1 - B_i$.
(d) $S_1, \ldots, S_m$—each containing 0 or 1, describing the current state of M, as follows. Assume the states of M are indexed from 1 to Q. Then the binary string corresponding to $S_1 \cdots S_m$ is equal to the binary index of the current state, where $m = \lceil \log_2 Q \rceil$.
(e) $\bar{S}_1, \ldots, \bar{S}_m$—for $1 \le i \le m$, $\bar{S}_i = 1 - S_i$.
(f) ONE—a variable holding the value 1.
(g) FIRST, BIG—control variables whose purpose will be explained later.

The program can be viewed as being composed of two main parts: initialization and simulation.

Initialization. The control variables are initialized using the following code segment (recall that initially they are all zero):

```
ONE ← ONE + 1            //ONE ← 1//
//if V_i = 0 then set A_i = 0 and Ā_i = 1, otherwise set A_i = 1 and Ā_i = 0, 1 ≤ i ≤ n//
do ONE
  Ā_1 ← Ā_1 + 1          //Ā_1 ← 1//
  if V_1 = 0 then exit
  A_1 ← A_1 + 1          //A_1 ← 1//
  Ā_1 ← Ā_1 ∸ 1          //Ā_1 ← 0//
end
  ⋮
do ONE
  Ā_n ← Ā_n + 1
  if V_n = 0 then exit
  A_n ← A_n + 1
  Ā_n ← Ā_n ∸ 1
end
//set B̄_1, . . , B̄_n to 1//
B̄_1 ← B̄_1 + 1
      ⋮
B̄_n ← B̄_n + 1
//set S̄_1, . . , S̄_m to 1, assuming the initial state has index 0//
S̄_1 ← S̄_1 + 1
      ⋮
S̄_m ← S̄_m + 1
```

Simulation. Code segments [segment(q, α)] are inserted for all states q and CSVs α. Each [segment(q, α)] simulates the computation of M while in CSV α provided this CSV was entered in state q. The code segments are arranged so that if $j < i$, then segments corresponding to CSVs in equivalence class E_j appear before all the code segments corresponding to CSVs in equivalence class E_i. The order among the code segments corresponding to CSVs in the same equivalence class E_k is arbitrary. Each [segment(q, α)] has the form (see also Figure 1):

```
//set FIRST to zero//
do FIRST
  FIRST ← FIRST ∸ 1
end
//simulate the computation of the "header," that is, the moves from states q (=q1), q2, ..., q(r-1)//
do ONE
  [code(q1)]
     .
     .
     .
  [code(q(r-1))]
  FIRST ← FIRST + 1        //FIRST ← 1//
end
//set BIG to hold a large value, that is, ≥ V1 + ··· + Vn//
do V1
  BIG ← BIG + 1
end
 .
 .
 .
do Vn
  BIG ← BIG + 1
end
//simulate the cycle, that is, iteration of moves from states qr, ..., qs//
do BIG
  if FIRST = 0 then exit
  [code(qr)]
     .
     .
     .
  [code(qs)]
end
```

Each of the code segments [code(p)] does the following:

(1) It checks whether the state is p. If so, then it goes on to (2); otherwise it causes an exit from the corresponding **do** construct. The following code checks for state consistency:

```
if s1 = 0 then exit
     .
     .
     .
if sm = 0 then exit
```

where s_i, $1 \leq i \leq m$, stands for S_i or for $\bar{S}_i$. s_i stands for S_i (respectively, $\bar{S}_i$) in case the ith digit in the binary representation of the index of state p (digit$_i(p)$, for short) is 1 (respectively, 0).

(2) It checks whether the CSV is α. If so, then it continues to (3); otherwise the **do** construct is exited. The code given here is similar to the one used in (1):

```
if a1 = 0 then exit
     .
     .
     .
if an = 0 then exit
if b1 = 0 then exit
     .
     .
     .
if bn = 0 then exit
```

where each a_i stands for A_i or $\bar{A}_i$ and each b_i stands for B_i or $\bar{B}_i$, $1 \le i \le n$, according to the following rules:

status of the ith counter	a_i	b_i
0	$\bar{A}_i$	$\bar{B}_i$
1	A_i	$\bar{B}_i$
2	$\bar{A}_i$	B_i

(3) It simulates the move that M makes in state p while in CSV α, according to the following cases:

(a) M has no next move. Then the **halt** instruction is executed.
(b) p is the state q_{loop}. Then the **loop** instruction is executed.
(c) The move involves adding +1 or −1 to the jth counter. The program executes the instruction $V_j \leftarrow V_j + 1$ or $V_j \leftarrow V_j \dot{-} 1$, respectively.

(4) It modifies the control variables to "remember" the new state and new CSV. Suppose that the move from state p (while in CSV α) involves changing the value in the jth counter and entering state p'. Then the code to modify S_i is determined according to the following cases:

(a) If $\text{digit}_i(p) = 0$ and $\text{digit}_i(p') = 1$, then the code is

$S_i \leftarrow S_i + 1$ //$S_i \leftarrow 1$//
$\bar{S}_i \leftarrow \bar{S}_i \dot{-} 1$ //$\bar{S}_i \leftarrow 0$//

(b) If $\text{digit}_i(p) = 1$ and $\text{digit}_i(p') = 0$, then the code is

$S_i \leftarrow S_i \dot{-} 1$ //$S_i \leftarrow 0$//
$\bar{S}_i \leftarrow \bar{S}_i + 1$ //$\bar{S}_i \leftarrow 1$//

(c) Otherwise (i.e., $\text{digit}_i(p) = \text{digit}_i(p')$) the code is empty.

The changes to the control variables keeping track of the jth counter status are done as follows:

(i) Suppose V_j changes from 0 to 1. Then A_j is set to 1 and $\bar{A}_j$ is set to 0.
(ii) Suppose V_j changes from positive to 0. Then A_j is set to equal 0, $\bar{A}_j$ is set to equal 1, B_j is set to equal 1, and $\bar{B}_j$ is set to 0. The following code will do these modifications:

//set A_j, B_j, $\bar{A}_j$, and $\bar{B}_j$ to correspond to a change of value in V_j from positive to 0//
$A_j \leftarrow A_j \dot{-} 1$ //$A_j \leftarrow 0$//
$B_j \leftarrow B_j + 1$ //$B_j \leftarrow 1$//
$\bar{A}_j \leftarrow \bar{A}_j + 1$ //$\bar{A}_j \leftarrow 1$//
$\bar{B}_j \leftarrow \bar{B}_j \dot{-} 1$ //$\bar{B}_j \leftarrow 0$//
//exit if V_j became zero (after being positive)//
if $V_j = 0$ **then exit**
//if V_j is not 0 then the values in A_j, B_j, $\bar{A}_j$, and $\bar{B}_j$ are restored//
$A_j \leftarrow A_j + 1$
$B_j \leftarrow B_j \dot{-} 1$
$\bar{A}_j \leftarrow \bar{A}_j \dot{-} 1$
$\bar{B}_j \leftarrow \bar{B}_j + 1$

The variable FIRST is used to check that the simulation of a cycle is entered only if its "header" is executed and the execution is completed without a change in the CSV. The variable BIG is set to a value big enough to allow the simulation of the cycle to continue until an **exit** instruction is encountered (which simulates a change in CSV). Note however that an **exit** instruction may not be reached if it has been preceded by a **halt** or **loop** instruction.

The program starts a computation by initializing the control variables and then moving to the code segment corresponding to the initial state and CSV. During the simulation the control variables make sure that only the code segments which simulate the machine computation are used. Recall now that execution of a code segment corresponding to some CSV in some equivalence class E_j eventually causes the program to either halt, enter an infinite loop, or enter a new segment in the equivalence class E_{j+1}. Thus the order among the code segments [segment(q, α)] in any equivalence class can be arbitrary. □

3. *From SL-Programs to Counter Machines*

The construction above takes a counter machine of size N into an equivalent program in SL whose size is exponential in N. We shall prove the converse of Theorem 3. In the converse, a program of size N is converted into an equivalent counter machine of size $O(p(N))$, where $p(N)$ is some polynomial in N. To simplify our discussion, we only consider SL-programs with no **loop** instructions. Generalizations of the results (as well as the proofs) are straightforward.

LEMMA 1. *Every program in SL of size N can be modified in $O(N^3)$ time into an equivalent program which makes references to at most four variables inside each* **do** ... **end** *construct.*

PROOF. Without loss of generality we may assume that execution of every **do** ... **end** construct which references more than four variables is terminated by an ifexit instruction and the variable controlling the iterations does not appear inside the **do** ... **end**. This follows from the observation that the code **do** x α **end** can be simulated as follows:

```
do x
  y ← y + 1
  w ← w + 1
end
w ← w + 1
do w
  if y = 0 then exit
  y ← y ∸ 1
  α
end
```

where y and w are new variables. (Note that the first **do** ... **end** construct references only two variables and hence does not violate our assumption.)

Consider a **do** ... **end** code segment of size s having the form

```
do x
  α1
  if y1 = 0 then exit
  α2
  .
  .
  .
  αr
  if yr = 0 then exit
  αr+1
end
```

Such a segment can be translated into a collection of $O(s^2)$ **do** ... **end** code segments, each of size at most $O(s)$, and allows references to no more than four variables inside every **do** ... **end** construct. $\alpha_1, \ldots, \alpha_{r+1}$ are assumed to consist only of succ and pred instructions. We have three phases.

Phase 1. Generate code segments to determine which of the **if** statements terminates the execution of the **do** ... **end** code segment. The idea is to consider successively pairs of **if** statements and, for each pair considered, to eliminate one of them from further consideration. Assume that at some stage the pth **if** statement is determined to be the one that terminates the execution of the **do** ... **end** code segment, disregarding the presence of the qth, ..., rth **if** statements (initially $p = 1$ and $q = 2$). Then the pair (p, q) is considered next to determine which of these two **if** statements terminates the execution of the **do** ... **end** code segment, disregarding the presence of the $(q + 1)$st, ..., rth **if** statements. (By convention, if none of the two causes the termination, then the pth **if** statement is chosen.)

The code for this phase introduces new control variables $h_1, \ldots, h_r$ with h_1 initialized to 1 and $h_2, \ldots, h_r$ initialized to 0. The code consists of segments corresponding to the pairs (p, q) in the following set and in the given order: (1, 2), (1, 3), (2, 3), (1, 4), (2, 4), (3, 4), (1, 5), (2, 5), (3, 5), (4, 5), ..., $(1, r)$, $(2, r)$, ..., $(r - 1, r)$. Each of these segments has the form

```
//y'_p ← y_p, y'_q ← y_q; y'_p and y'_q are new variables//
do y_p
  y'_p ← y'_p + 1
end
do y_q
  y'_q ← y'_q + 1
end
//At this point h_p is 1 and the other h_i's are 0 if and only if the pth if statement terminates the execution
of the given do ... end code segment, disregarding the presence of the qth, (q + 1)st, .. , rth if statements.
The following code is used only if h_p = 1. It determines which of the pth or qth if statements terminates
the execution of the given do .. end code segment, disregarding the presence of the (q + 1)st, ..., rth if
statements If none of the two, then the pth if statement is chosen to be the one//
do x
  if h_p = 0 then exit
  β_1
  ⋮
  β_p
  if y'_p = 0 then exit
  h_p ← h_p ∸ 1
  h_q ← h_q + 1
  β_{p+1}
  ⋮
  β_q
  if y'_q = 0 then exit
  h_q ← h_q ∸ 1
  h_p ← h_p + 1
  β_{q+1}
  ⋮
  β_{r+1}
end
//β_j is derived from α_j by replacing the occurrences of y_p and y_q with y'_p and y'_q, respectively, and deleting
the instructions that do not refer to these variables //
```

Note that there are exactly $(r - 1)r/2$ such code segments, each of size at most $O(s)$.

Phase 2. Generate code segments to simulate the changes in value of the variables appearing inside the **do** ... **end** code segment. Corresponding to each variable z and each **if** statement that uses a variable other than z, a segment of the following form is introduced:

```
//The code modifies z provided the pth if statement terminates the execution of the do .  end
segment //
do y_p
  if h_p = 0 then exit
  y_p'' <- y_p'' + 1
end
do x
  if h_p = 0 then exit
  γ_1
  .
  .
  .
  γ_p
  if y_p'' = 0 then exit
  γ_{p+1}
  .
  .
  .
  γ_{r+1}
end
```

where γ_j is derived from α_j by deleting the instructions that do not refer to y_p or z and replacing the occurrences of y_p by a new variable y_p''. Note that there are $O(r\cdot(\text{number of variables}))$ code segments having this form, each of size at most $O(s)$.

Phase 3. Generate code segments to set to 0 the variable y_p if y_p corresponds to the **if** statement that terminates the execution of the **do** ... **end** segments. For each y_p we insert the code

```
do y_p
  if h_p = 0 then exit
  y_p <- y_p ∸ 1
end
```

It is easy to verify that the time of translation (described in phases 1–3) is $O(s^3)$. $\square$

We are now ready to prove the converse of Theorem 3.

THEOREM 4. *Every SL-program can be converted into an equivalent machine M in $C(n, n_1, n_2, k)$ for some n, n_1, n_2, and k. Moreover, the machine can be constructed from the program in $p(N)$ time, where $p(N)$ is a polynomial in the size of the program.*

PROOF. By Lemma 1, the program can be translated in $O(N^3)$ time into an equivalent program that makes references to at most four variables inside any **do** ... **end** construct. Assume that the program obtained by such a translation is in SL(m, n_1, n_2), and denote its variables by $x_1, \ldots, x_m$. Clearly, $m \le O(N^3)$. Consider any computation by the program. We describe how a counter machine M realizes the computation. M will have $O(N^{15})$ states and can be constructed to be in $C(n, n_1, n_2, k)$, where $n = m + 2$ and $k \le O(N^3)$.

Denote the counters of M by $C_1, \ldots, C_{m+2}$. M simulates the given computation using counters $C_1, \ldots, C_m$ to hold the values in the variables $x_1, \ldots, x_m$, respectively.

Simulation of succ and pred instructions is straightforward when they are not imbedded in a **do** ... **end** construct. To simulate a **do** ... **end** code segment of the form

do x_i
  I_1
  ⋮
  I_r
end

we need the observation that during a single iteration, each of the four variables referred to in $I_1 \cdots I_r$ (say $x_{j_1}, x_{j_2}, x_{j_3}, x_{j_4}$) is changed by at most the value r. The finite control of M uses four buffers, B_1, B_2, B_3, and B_4, to simulate the changes in these variables during the iterations. Each of the buffers is of length $2r$, and initially they are all zero. The simulation is as follows.

Step 1. Initialize counter C_{m+1} to the value contained in C_i (corresponding to the value in the control variable x_i of the **do** ... **end** construct). This is done by first copying the contents of C_i into C_{m+1} and C_{m+2} simultaneously and then copying the contents of C_{m+2} into C_i

Step 2. If the value in C_{m+1} is 0, then add to counter C_{j_u}, $1 \le u \le 4$, the value in the corresponding buffer B_u and proceed to simulate the code segment following the **do** .. **end** construct.

Step 3 Decrease counter C_{m+1} by 1.

Step 4 Add to buffer B_u and subtract from counter C_{j_u}, $1 \le u \le 4$, the value d_u (negative, zero, positive), where

$$d_u = \begin{cases} r - B_u & \text{if } C_{j_u} + B_u > r, \\ C_{j_u} & \text{otherwise.} \end{cases}$$

Step 5 Simulate the instructions $I_1, \ldots, I_r$ with the buffers $B_1, \ldots, B_4$ standing for the variables $x_{j_1}, \ldots, x_{j_4}$, respectively Note that at the start of each iteration, $B_u = \min\{x_{j_u}, r\}$, and during the iteration exactly r instructions are simulated Thus B_u has the value 0 when an **if** instruction is encountered if and only if x_{j_u} does so, $1 \le u \le 4$ If an **exit** is to be simulated, then add to counter C_{j_u}, $1 \le u \le 4$, the value in B_u and proceed to simulate the code following the **do** ... **end** construct

Step 6. Go to step 2

Note that during the simulation each of the d_u's changes its sign by at most some constant number of times. Thus each of the counters reverses at most some fixed number of times. Each of the buffers can hold at most the value $2r$, and no more than four buffers are simultaneously in use. Also, there are only r instructions inside the **do** ... **end** construct. Therefore the number of states required does not exceed $O((2r)^4 r) = O(r^5)$. It follows that the total number of states of M does not exceed $O((\text{number of instructions in the program})^5) \le O(N^{15})$. The total number of reversals per counter is at most linear in the number of instructions in the program, that is, no greater than $O(N^3)$. □

From Theorems 1–4, we have

THEOREM 5. *A relation (function) is definable by a Presburger formula if and only if it is computable by an SL-program.*

4. *Complexity of the Equivalence Problem for SL-Programs*

In this section we derive some upper bounds on the complexity of the equivalence problem for SL-programs. The bounds are obtained using the results of Section 3 and the known complexity bounds for the equivalence problem of counter machines [10]. In order to make the concept of equivalence precise, we give the following definition.

Definition. Two SL-programs or counter machines are *relationally equivalent* (respectively, *functionally equivalent*) if they define the same *relation* (respectively, *function*).

Convention. If a result holds for both types of equivalence, the word "relationally" or "functionally" is omitted.

In a recent paper [10] the following theorem was shown.

THEOREM 6.

(*a*) *The equivalence problem for counter machines is decidable in deterministic time* $2^{c(N/\log N)^4}$, *where c is some positive constant.*
(*b*) *The inequivalence problem for counter machines is NP-complete.*

In [6] the inequivalence problem for L_1-programs was shown to be NP-complete. (Actually, [6] contains only the proof of NP-hardness [7, 13]; membership in NP was shown in [16].) The construction in [6] can easily be modified to show that the inequivalence problem for SL-programs is NP-hard. Thus, from Theorems 4 and 6(b) and the fact that every language in NP can be accepted by a deterministic polynomial space-bounded Turing machine [1], we have

THEOREM 7.

(*a*) *The inequivalence problem for SL-programs is NP-complete.*
(*b*) *The equivalence problem for SL-programs is decidable in deterministic polynomial space and therefore in time* $2^{p(N)}$ *for some polynomial* $p(N)$.

We will show that the programming language SL is minimal, that is, the instructions in SL are independent. The proof uses the following lemma.

LEMMA 2. *Let P be a program in SL-{*__if__*} defining a total function. Let* $x_1, \ldots, x_n$ *be its input variables, whose initial values are denoted by* $x_1^0, \ldots, x_n^0$, *respectively. Then the final value of any variable v has the form* $a_1x_1^0 + \cdots + a_nx_n^0 + a_{n+1}$, *where* $a_1, \ldots, a_{n+1}$ *(which may depend on* $x_1^0, \ldots, x_n^0$*) come from a finite set of integers (positive, negative, zero) whose cardinality is independent of the magnitudes of* $x_1^0, \ldots, x_n^0$. *Hence the function* $f(x) = \lfloor x/2 \rfloor$ *cannot be computed in SL-{*__if__*}.*

PROOF. The argument is an induction on the number of instructions executed so far. The basis is obvious. So assume that after the ith instruction ($i \geq 0$), the value of the variable v is given by $v_i = a_1x_1^0 + \cdots + a_nx_n^0 + a_{n+1}$. We consider several situations depending on the form of the $(i + 1)$st instruction.

(a) If the $(i + 1)$st instruction does not involve the variable v, then $v_{i+1} = v_i$.
(b) If the $(i + 1)$st instruction is of the form $v \leftarrow v + 1$, then $v_{i+1} = (a_1x_1^0 + \cdots + a_nx_n^0 + a_{n+1}) + 1 = a_1x_1^0 + \cdots + a_nx_n^0 + (a_{n+1} + 1)$. Clearly, v_{i+1} has the right form.
(c) If the $(i + 1)$st instruction is of the form $v \leftarrow v \dot{-} 1$, then either $v_{i+1} = a_1x_1^0 + \cdots + a_nx_n^0 + (a_{n+1} - 1)$ if $v_i \geq 1$ or $v_{i+1} = 0$ if $v_i = 0$. In either case, v_{i+1} has the right form.
(d) The $(i + 1)$st instruction cannot be a **loop** instruction, since by assumption P defines a total function.
(e) The $(i + 1)$st instruction is of the form **do** u α **end** and $v \leftarrow v + 1$ and/or $v \leftarrow v \dot{-} 1$ occurs in α. Let $u = b_1x_1^0 + \cdots + b_nx_n^0 + b_{n+1}$. If $u = 0$, then $v_{i+1} = v_i$. So

assume that $u > 0$. Let the net change in variable v in one iteration of α be c (positive, negative, zero). Since $u > 0$, **loop** is not an instruction in α.

Case 1. v is not zero on entering the **do** instruction (i.e., $v_t > 0$), and v does not become zero during the execution of the **do** instruction. Then $v_{t+1} = (a_1x_1^0 + \cdots + a_nx_n^0 + a_{n+1}) + c(b_1x_1^0 + \cdots + b_nx_n^0 + b_{n+1}) = (a_1 + cb_1)x_1^0 + \cdots + (a_n + cb_n)x_n^0 + (a_{n+1} + cb_{n+1}) > 0$. Hence, v_{t+1} has the right form.

Case 2. v is zero on entering the **do** instruction (i.e., $v_t = 0$) or v becomes zero during the execution of the **do** instruction. Let r be the number of instructions in α. We have three subcases.

Subcase 1. $c < 0$. Thus $v_{t+1} \leq r$, and v_{t+1} has the right form.

Subcase 2. $c \geq 0$ and $u \geq r$.

(i) Suppose v becomes zero during each of the first r iterations of α. Then the process has entered a cycle, and v will become zero during each of the succeeding iterations. Then on the completion of the **do** instruction, $v_{t+1} \leq r$.

(ii) Suppose v does not become zero during the tth iteration and t is the first such iteration. By (i), $t < r$. Then v will not become zero during each of the succeeding iterations. Let v_t' be the value of v after t iterations of α (or equivalently, after executing t copies of α). Clearly, v_t' can be written as $v_t' = a_1'x_1^0 + \cdots + a_n'x_n^0 + a_{n+1}' > 0$. (See (a)–(d).). Let $u' = u - t$. Then $u' \geq 1$ since $u \geq r > t$. Then v_{t+1} can be computed from v_t' as described in case 1.

Subcase 3. $c \geq 0$ and $u < r$. Clearly, the value v_{t+1} of v after executing the construct **do** u α **end** is the same as the value obtained by executing u copies of α. It follows from (a)–(d) that v_{t+1} has the right form. □

THEOREM 8. *The programming language SL is minimal.*

PROOF. We show that the instructions in SL are independent.

(a) **loop** cannot be eliminated, since without it, partial functions cannot be defined.
(b) $x \leftarrow x + 1$ (respectively, $x \leftarrow x \dot{-} 1$) cannot be eliminated, since without it, the function $f(x) = x + 1$ (respectively, $f(x) = x \dot{-} 1$) cannot be computed.
(c) **do** ... **end** cannot be eliminated, since without it, the function $f(x) = 2x$ cannot be computed. (Note that if P is a {succ, pred, ifexit, loop}-program and v is a variable of P, then the final value v_f of v, if defined, must satisfy $v_0 \dot{-} r \leq v_f \leq v_0 + r$, where v_0 is the initial value of v and r is the number of instructions in P.)
(d) **if** $x = 0$ **then exit** cannot be eliminated, since by Lemma 2, $f(x) = \lfloor x/2 \rfloor$ cannot be computed in SL − {**if**}. □

5. *XL and UL*

SL is minimal in the sense that all its instructions are independent. In this section we extend SL to include other programming language constructs without changing its computing capability. The first extension we consider is the programming language XL = {succ, pred, ifexit, loop, do, $x \leftarrow 0$, $x \leftarrow y$, goto, ifgoto}. We shall see that XL is, in a certain sense, maximal in that any further generalization results in a language which has an undecidable equivalence problem.

In Theorem 9 we show how a program in XL can be translated into an equivalent program in SL in polynomial time. Hence the class of XL-programs also characterizes the Presburger arithmetic. The class is similar to the class of L_+-programs studied in [3]. $L_+ = XL - \{\textbf{do}\}$ without restriction (R2). However, the programs must satisfy a certain structural property. Intuitively, the property is that no assignment statement is embedded in more than one loop. (See [3] for the precise definition.) It was shown in [3] that L_+-programs realize exactly the Presburger formulas and that the equivalence problem for L_+-programs is decidable in

$$2^{2^{2^{2^{p(N)}}}}$$

time. As a corollary to Theorem 9 we have that the equivalence problem for programs in XL is decidable in $2^{p(N)}$ time. ($p(N)$ is a polynomial in the sum of the sizes of the programs being considered.)

THEOREM 9. *Every program in XL of size N can be translated into an equivalent program in SL in $O(N^{12})$ time.*

PROOF. In what follows, ZERO and ONE are new variables initialized to 0 and 1, respectively.

Given a program in XL, the instructions not in SL are eliminated according to the following procedure.

Step 1. Replace each **goto** l instruction with an **if** ZERO $= 0$ **then goto** l instruction

Step 2. Replace every instruction of the form **if** $x = 0$ **then goto** l that is not embedded in a **do** $\cdots$ **end** construct by

```
do ONE
  if x = 0 then goto l
end
```

Step 3. Eliminate the **if** $x = 0$ **then goto** l instructions (By steps 1 and 2, they are now all embedded in **do** . . . **end** constructs.) Each such instruction is replaced by a three-instruction code as follows:

```
   do y                          do y
                                   .
     .                             .
     .                             .
     .                            h ← h ∸ 1
    if x = 0 then goto l  ⇒  {   if x = 0 then exit
                                  h ← h + 1
     .                             .
     .                             .
     .                             .
   end                           end
   α                             α
l:                              l
```

where h is a new control variable initialized to 1. Clearly, h is 0 upon leaving the **do** end construct if and only if the execution of the **do** . . . **end** code segment was terminated by the **if** $x = 0$ **then exit** instruction. Then α, the code between the **end** and the instruction labeled l, is modified as follows

(a) An instruction I in α not embedded in a **do** . . . **end** construct is replaced by the code

```
do ONE
  if h = 0 then exit
  I
end
```

(b) A code segment (of α) having the form **do** z β **end** is translated into the form

```
do z
  if h = 0 then exit
  β
end
```

Thus, in simulating the **if** $x = 0$ **then goto** l instruction, the control variable h takes care of "skipping" instructions.

Step 4 Replace each instruction of the form $x \leftarrow 0$ by an instruction of the form $x \leftarrow \text{ZERO}$.

Step 5. Eliminate the instructions of the form $x_i \leftarrow x_j$. The elimination of such an instruction is the most difficult part of the proof and is described in the appendix (Lemma A4)

One easily verifies that steps 1, 2, and 4, can be done in $O(N)$ time. Step 3 can be executed in $O(N^2)$ time. By Lemma A4, step 5 can be done in $O(N^6)$ time. It follows that an overall time complexity of $O(N^{12})$ is sufficient for translation. □

From Theorems 7 and 9 we have the main result of the paper.

THEOREM 10.

(*a*) *The inequivalence problem for XL-programs is NP-complete.*

(*b*) *The equivalence problem for XL-programs is decidable in deterministic polynomial space and, therefore, in time $2^{p(N)}$ for some polynomial $p(N)$.*

Our next theorem shows that XL-programs realize exactly the Presburger formulas. However, constructing an XL-program realizing a Presburger formula requires superexponential space (in the size of the formula) for infinitely many formulas. The proof uses a result of [8]. (It is easy to verify that the theorem also applies to SL-programs.)

THEOREM 11. *A relation (function) is definable by a Presburger formula if and only if it is computable by an XL-program. Moreover, there is a constant $c > 0$ such that if T is a Turing machine that constructs XL-programs realizing the Presburger formulas, then for every integer $k > 0$ there is a formula of size $N \geq k$ for which T requires more than $2^{2^{cN}}$ space in the construction.*

PROOF. The first statement follows from Theorems 5 and 9. Now let T be a Turing machine which constructs XL-program realizations of Presburger formulas. Let $S(N)$ be its space bound. If the second statement of the theorem is false, then for every $c > 0$ there is an integer k_c such that $S(N) \leq 2^{2^{cN}}$ for all formulas of size $N \geq k_c$. We can construct another Turing machine T' which decides equivalence of Presburger formulas as follows.

Given two formulas F_1 and F_2, T' uses T to construct XL-programs P_1 and P_2 realizing F_1 and F_2, respectively. Then T' determines if P_1 and P_2 are equivalent using space polynomial in the sum of the sizes of P_1 and P_2 (Theorem 10(b)). Hence there is a constant $d > 0$ such that if F_1 and F_2 have sizes N_1 and N_2, respectively, then T' can decide their equivalence in space $(S(N_1) + S(N_2))^d$. It follows that T' also has the property that for every constant $c > 0$ there is an integer k_c such that for all $N \geq k_c$, T' uses no more than $2^{2^{cN}}$ space in deciding equivalence of formulas F_1 and F_2, where $N = \text{SIZE}(F_1) + \text{SIZE}(F_2)$. However, by the results in [8] no such Turing machine exists, a contradiction. □

We have seen that the inequivalence problem for SL and XL is NP-complete. We now consider very briefly the complexity of the evaluation problem for these languages. The evaluation problem is the following: Given an arbitrary program P and an arbitrary input $(x_1, \ldots, x_n)$, (i) determine whether or not P halts on $(x_1, \ldots, x_n)$, and (ii) compute the output values (if there is at least one output variable). The evaluation problem for counter machines is defined similarly.

Our next result shows that the evaluation problem for XL is solvable on a multitape Turing machine in time polynomial in $\text{SIZE}(P) + \log(x_1 + \cdots + x_n)$.

The polynomial bound is not at all obvious. If a Turing machine were to do a step-by-step simulation of the computation of an XL-program P, a time bound exponential in SIZE(P) + $\log(x_1 + \cdots + x_n)$ can be achieved in the worst case. For example, consider the program

$$\left.\begin{matrix}\alpha \\ \vdots \\ \alpha\end{matrix}\right\} n \text{ times}$$

where α is the code

```
do x
  x ← x + 1
  x ← x + 1
end
```

Then the final value of x is $2^n v$, where v is its initial value. A step-by-step simulation of P will then take at least time $O(2^n v) \geq O(2^{c(\mathrm{SIZE}(P)+\log v)})$ for some $c > 0$.

THEOREM 12. *We can effectively construct a deterministic multitape Turing machine T which solves the evaluation problem for XL. Moreover, the execution time of T when given the representations of P and input $(x_1, \ldots, x_n)$ is $q(SIZE(P) + log(x_1 + \cdots + x_n))$ for some polynomial $q(\ )$.*

PROOF. By Theorems 9 and 4 we can construct a Turing machine which converts any program P in XL into an equivalent multicounter machine M. Moreover, the translation takes time polynomial in the size of P. The result now follows from the fact that the present theorem is true for multicounter machines [10]. □

In the definition of XL we imposed the following restrictions:

(R1) **do** . . . **end** constructs cannot be nested.
(R2) Only forward **goto** labels are allowed.
(R3) No instruction in the scope of a **do** . . . **end** construct can be labeled.

Relaxing the restrictions by dropping either (R1) or (R2) results in a language at least as powerful as the loop language L_2, and L_2 has an undecidable equivalence problem [15]. If, on the other hand, restriction (R3) is dropped, then we have the following theorem.

THEOREM 13. *Let UL be the language XL with restriction (R3) removed. Let TF(UL) and TF(L_2) be the classes of total functions computable by UL-programs and L_2-programs, respectively. Then TF(UL) $\subsetneq$ TF(L_2).*

PROOF. That TF(UL) $\subseteq$ TF(L_2) follows from a result in [5] which shows that TF(L_2) = TF(SR_1), SR_1 being a language containing UL-{**loop**}. To prove proper containment, we show that $f(x, y) = x * y$ is not a UL-function. Suppose that there is a UL-program P which computes $f(x, y) = x * y$. Let s be the number of instructions in P. (A **do** . . . **end** construct is considered one instruction.) Clearly, if the maximum value of any variable upon entering a **do** . . . **end** construct is t, then the maximum value of any variable upon exit is $O(st)$. Now there are at most $O(s)$ **do** . . . **end** constructs and at most $O(s)$ statements in each **do** . . . **end** construct. It follows that the maximum value of any variable at the end of the computation is $O(s^s(x + y))$. Since s is fixed, $O(s^s(x + y)) < x * y$ for large values of x and y. Therefore, P cannot compute $f(x, y) = x * y$, a contradiction. □

Although multiplication is not UL-program computable, integer division is. In fact, $\lfloor x/y \rfloor$ can be computed by a program using only instructions in SL-{**loop**} plus the instructions **goto** l and **if** $x = 0$ **then goto** l (i.e., the instruction set {succ, pred, do, goto, ifgoto}). The **goto** and **if** instructions can only appear inside **do** ... **end** constructs with l being a forward label in the scope of the **do** ... **end** construct containing the **goto** or **if**. We shall call this last language UL^-.

PROPOSITION 1. *$\lfloor x/y \rfloor$ is UL^--program computable.*

PROOF. The following UL^--program computes $\lfloor x/y \rfloor$ into z (note that t and y' are initially 0):

```
    t ← t + 1
    do x
      if t = 0 then goto l2
      y ← y ∸ 1
      y' ← y' + 1
      if y = 0 then goto l1
      goto l4
l1  //y = 0//
      t ← t ∸ 1
      z ← z + 1
      goto l4
l2: //t = 0//
      y ← y + 1
      y' ← y' ∸ 1
      if y' = 0 then goto l3
      goto l4
l3  //y' = 0//
      t ← t + 1
      z ← z + 1
l4: end
```

□

It is well known that the equivalence problem for L_2-programs is undecidable [15]. The proof of the next theorem follows from Proposition 1 and the undecidability of Hilbert's tenth problem [14]. (Hilbert's tenth problem is the problem of deciding for any given polynomial with integer coefficients whether it has a nonnegative integral solution [11].)

THEOREM 14. *The equivalence problem for UL^--programs is undecidable.*

Appendix

In this appendix we prove that every program in SL ∪ $\{x \leftarrow y\}$ can be converted into an equivalent program in SL in polynomial time.

Notation. Let α be a program segment which contains only succ, pred, and $x \leftarrow y$ instructions. Assume that $x_1, \ldots, x_n$ are the only variables in α. We denote by $x_1^{(m)}, \ldots, x_n^{(m)}$, $m \geq 0$, the values in the variables $x_1, \ldots, x_n$ after m iterations (i.e., executions) of α. We use a pair of brackets [. . .] to enclose tasks that are SL-computable. The proof that such tasks are SL-computable is usually left to the reader.

Four lemmas are needed to prove that every SL ∪ $\{x \leftarrow y\}$ program is polynomial-time reducible to an equivalent SL program. The first two use some ideas previously given in [4].

LEMMA A1. *There is an algorithm which when given input (i, α), $1 \leq i \leq n$, produces output (j, β), where*

(*a*) *β is a program segment which uses only the instructions $x_i \leftarrow x_i + 1$ and $x_i \leftarrow x_i \dot{-} 1$.*
(*b*) *Suppose that $x_1, \ldots, x_n$ contain the values $x_1^{(m)}, \ldots, x_n^{(m)}$, respectively. Then after the execution of the code*

$x_i \leftarrow x_j$
β

x_i contains $x_i^{(m+1)}$.
(*c*) *The algorithm has time complexity $O(r \log n)$, where r is the number of instructions in α.*

PROOF. Let $I_1; I_2; \ldots; I_r$ be the instructions of α. Then j and the instructions $J_u; J_{u-1}; \ldots; J_1$ of β are uniquely determined from α and i in $O(r \log n)$ time:

```
algorithm
  j ← i
  u ← 0
  for k ← r, r − 1, ..., 1 do
    case I_k is the instruction
    x_j ← x_p : j ← p
    x_j ← x_j + 1  u ← u + 1;
                   set J_u to be the instruction x_i ← x_i + 1
    x_j ← x_j ∸ 1 : u ← u + 1;
                   set J_u to be the instruction x_i ← x_i ∸ 1
    x_p ← x_q or      }
    x_p ← x_p + 1 or  }  do nothing
    x_p ← x_p ∸ 1,    }
    where p ≠ j       }
    end
  end
end
```

□

LEMMA A2. *For each variable x_i, we can construct an SL-program segment P_i with the following properties*:

(*a*) *Suppose $x_1^{(0)}, \ldots, x_n^{(0)}$ are the values in $x_1, \ldots, x_n$, respectively. There is a new variable x_i' such that after the execution of P_i the variable x_i' contains the value $x_i^{(m)}$. (P_i references a variable w which is assumed to contain m before P_i is executed.)*
(*b*) *The values of the variables $x_1, \ldots, x_n$ are not modified by P_i.*
(*c*) *P_i can be obtained from α and i in $O(rn \log n)$ time, where r is the number of instructions in α.*

PROOF. Given x_i, we first find the (unique) sequence of integers $j_1, \ldots, j_s, \ldots, j_{s+q}$ and the corresponding sequence of program segments $\beta_1, \ldots, \beta_s, \ldots, \beta_{s+q}$ such that

(a) $\{j_1, \ldots, j_{s+q}\} \subseteq \{1, \ldots, n\}$ is a set of distinct integers.
(b) $j_{s+q} = i$.
(c) For input (j_k, α) the algorithm of Lemma A1 produces output (j_{k-1}, β_k), $k = s+q, s+q-1, \ldots, 2$.
(d) For input (j_1, α) the algorithm of Lemma A1 produces output (j_s, β_1).

Clearly, the sequence of integers $j_1, \ldots, j_{s+q}$ and the sequence of program segments $\beta_1, \ldots, \beta_{s+q}$ can be obtained in $O(nr \log n)$ time.

Next, we construct an intermediate SL-program segment $\hat{P}_i$ computing $x_i^{(m)}$. In the program segment the code [initialize $h_2, \ldots, h_{s+q}, w'$] initializes the new variables $h_2, \ldots, h_{s+q}$ and w' so that

(a) $1 \geq h_{s+q} \geq h_{s+q-1} \geq \cdots \geq h_2 \geq 0$.
(b) $w' \geq 0$ and, if $w' > 0$, then $h_{s+1} = 1$.
(c) $sw' + \sum_{k=2}^{s+q} h_k = m$.

The SL-program $\hat{P}_i$ that computes $x_i^{(m)}$ is

```
[initialize h_2, ..., h_{s+q} w']
[if h_{k+1} ≠ 0 then x_{j_{k+1}} ← x_{j_k}]  ⎫
do h_{k+1}                                  ⎬ repeat for
  β_{k+1}                                   ⎪ k = 1, 2,     , s − 1
end                                         ⎭
do w'
  x_{j_1} ← x_{j_s}
  β_1
  x_{j_2} ← x_{j_1}
    .
    .
    .
  x_{j_s} ← x_{j_{s-1}}
  β_s
end
[if h_{k+1} ≠ 0 then x_{j_{k+1}} ← x_{j_k}]  ⎫
do h_{k+1}                                  ⎬ repeat for
  β_{k+1}                                   ⎪ k = s, s + 1,  . , s + q − 1
end                                         ⎭
```

The program segment [initialize $h_2, \ldots, h_{s+q}, w'$] has the following form:

```
[w' ← ⌊(w ∸ q)/s⌋]
w'' ← w ∸ w'
do ONE                                      ⎫
  if w'' = 0 then exit                      ⎪ repeat for
  h_k ← h_k + 1      //h_k ← 1//            ⎬ k = s + q, s + q − 1,   ., 2
  w'' ← w'' ∸ 1                             ⎪
end                                         ⎭
```

ONE and w'' are new variables, where ONE is initialized to 1.

Finally, P_i is obtained from $\hat{P}_i$ as follows:

(a) The variables $x_1, \ldots, x_n$ are replaced by new variables $x_1', \ldots, x_n'$, respectively.
(b) The variables inside the **do** $w' \ldots$ **end** construct are replaced by the variable x_{j_s}'. Then the instructions of the form $x_{j_s}' \leftarrow x_{j_s}'$ are deleted. (Since the only variable in each β_k is x_{j_k}, it is straightforward to verify that the code inside the **do** is equivalent to the resulting code.)
(c) The following code is inserted as the first instruction in P_i

$$[x_1' \leftarrow x_1; \ldots; x_n' \leftarrow x_n]$$ □

LEMMA A3. *Consider a computation by a program segment of the form*

```
do x_i
  α_1
  if x_i = 0 then exit
  α_2
end
```

where α_1 and α_2 contain only instructions of the form $x \leftarrow x + 1$, $x \leftarrow x \dot{-} 1$, and $x \leftarrow y$. Assume that the only variables used in $\alpha_1\alpha_2$ are $x_1, \ldots, x_n$ and there are r instructions in $\alpha_1\alpha_2$. Then we can construct an SL-program segment P in $O(r^2n^3 \log n)$ time which has the following properties:

(*a*) *P has a new variable t. Execution of P gives in t the number of times the* **if** *instruction is encountered during the computation being considered.*
(*b*) *P leaves $x_1, \ldots, x_n$ unchanged.*

PROOF. Let α_1' and α_2' be the program segments obtained from α_1 and α_2 by replacing the occurrences of $x_1, \ldots, x_n$ by new variables $x_1', \ldots, x_n'$, respectively. Then the following program segment F_1 will produce in t the desired value:

$$\begin{bmatrix} x_1' \leftarrow x_1, x_2' \leftarrow x_2; \ldots, x_n' \leftarrow x_n \\ t \leftarrow 0; z \leftarrow x_l \\ \alpha_1' \end{bmatrix}$$
do z
  $t \leftarrow t + 1$
  if $x_\iota' = 0$ **then exit**
  α_2'
  α_1'
end

Then using the technique in the proof of Lemma A2 we can rewrite F_1 by program F_2 below. For now we assume that the program segments [initialize $h_2, \ldots, h_{s+q}$] and [initialize w] initialize the new variables $h_2, \ldots, h_{s+q}$ and w so that

(a) $1 \geq h_{s+q} \geq h_{s+q-1} \geq \cdots \geq h_2 \geq 0$.
(b) $w \geq 0$ and, if $w > 0$, then $h_{s+1} = 1$.
(c) $sw + \sum_{k=2}^{s+k} h_k = \min(\{z\} \cup \{z_0 \mid$ execution of F_1 with z set to z_0 causes x_ι' to have value 0 on encountering the **if** statement in the z_0th time, but not at previous times the **if** statement was encountered$\})$.

Program segment F_2 has the following form:

$$\begin{bmatrix} x_1' \leftarrow x_1; \ldots, x_n' \leftarrow x_n \\ t \leftarrow 0; z \leftarrow x_l \\ \alpha_1' \end{bmatrix}$$
[initialize $h_2, \ldots, h_{s+q}$]
$\left.\begin{array}{l} [\textbf{if } h_{k+1} \neq 0 \textbf{ then } x_{j_{k+1}}' \leftarrow x_{j_k}'] \\ \textbf{do } h_{k+1} \\ \quad t \leftarrow t + 1 \\ \quad \beta_{k+1} \\ \textbf{end} \end{array}\right\}$ repeat for $k = 1, 2, \ldots, s - 1$
[initialize w]
do w
  $x_{j_1}' \leftarrow x_{j_s}'$
  $t \leftarrow t + 1$
  β_1
  ⋮
  $x_{j_s}' \leftarrow x_{j_{s-1}}'$
  $t \leftarrow t + 1$
  β_s
end
$\left.\begin{array}{l} [\textbf{if } h_{k+1} \neq 0 \textbf{ then } x_{j_{k+1}}' \leftarrow x_{j_k}'] \\ \textbf{do } h_{k+1} \\ \quad t \leftarrow t + 1 \\ \quad \beta_{k+1} \\ \textbf{end} \end{array}\right\}$ repeat for $k = s, s + 1, \ldots, s + q - 1$

where each β_k uses only the instructions $x'_{j_k} \leftarrow x'_{j_k} + 1$ and $x'_{j_k} \leftarrow x'_{j_k} \dot{-} 1$ and $x'_{j_{s+q}}$ is the variable x_i'. Finally, F_2 can be modified to produce program F_3 having the following form:

$$\left[\begin{array}{l} x_1' \leftarrow x_1, \ldots, x_n' \leftarrow x_n \\ t \leftarrow 0;\ z \leftarrow x_l \\ \alpha_1' \end{array}\right]$$

[initialize $h_2, \ldots, h_{s+q}$]

$$\left.\begin{array}{l} [\textbf{if } h_{k+1} \neq 0 \textbf{ then } x'_{j_{k+1}} \leftarrow x'_{j_k}] \\ \textbf{do } h_{k+1} \\ \quad t \leftarrow t+1 \\ \quad \beta_{k+1} \\ \textbf{end} \end{array}\right\} \begin{array}{l} \text{repeat for} \\ k = 1, 2, \ldots, s-1 \end{array}$$

[initialize w]

$$\begin{array}{l} \textbf{do } w \\ \quad t \leftarrow t+1 \\ \quad \gamma_1 \\ \quad \vdots \\ \quad t \leftarrow t+1 \\ \quad \gamma_s \\ \textbf{end} \end{array}$$

$$\left.\begin{array}{l} [\textbf{if } h_{k+1} \neq 0 \textbf{ then } x'_{j_{k+1}} \leftarrow x'_{j_k}] \\ \textbf{do } h_{k+1} \\ \quad t \leftarrow t+1 \\ \quad \beta_{k+1} \\ \textbf{end} \end{array}\right\} \begin{array}{l} \text{repeat for} \\ k = s, s+1, \ldots, s+q-1 \end{array}$$

where each γ_k is obtained from β_k by replacing the occurrences of x'_{j_k} by x'_{j_s}. Program segment F_3 works assuming that $h_2, \ldots, h_{s+q}$ and w are properly initialized. Unfortunately, there is no easy way to directly initialize these variables. However, we can still construct a program segment that computes t indirectly. To do this, we need a program F_4 similar to F_3:

$$\left[\begin{array}{l} x_1' \leftarrow x_1, \ldots, x_n' \leftarrow x_n \\ t \leftarrow 0,\ z \leftarrow x_l \\ \alpha_1' \end{array}\right]$$

[initialize $h_2, \ldots, h_{s+q}$ to some fixed choice of values satisfying $1 \geq h_{s+q} \geq h_{s+q-1} \geq \cdots \geq h_2 \geq 0$]

$$\left.\begin{array}{l} [\textbf{if } h_{k+1} \neq 0 \textbf{ then } x'_{j_{k+1}} \leftarrow x'_{j_k}] \\ \textbf{do } h_{k+1} \\ \quad t \leftarrow t+1 \\ \quad \beta_{k+1} \\ \textbf{end} \end{array}\right\} \begin{array}{l} \text{repeat for} \\ k = 1, 2, \ldots, s-1 \end{array}$$

[initialize w to a value as described in cases (a)–(c) below]

$$\begin{array}{l} \textbf{do } w \\ \quad t \leftarrow t+1 \\ \quad \gamma_1 \\ \quad \vdots \\ \quad t \leftarrow t+1 \\ \quad \gamma_s \\ \textbf{end} \end{array}$$

$$\left.\begin{array}{l} [\textbf{if } h_{k+1} \neq 0 \textbf{ then } x'_{j_{k+1}} \leftarrow x'_{j_k}] \\ \textbf{do } h_{k+1} \\ \quad t \leftarrow t+1 \\ \quad \beta_{k+1} \\ \textbf{end} \end{array}\right\} \begin{array}{l} \text{repeat for} \\ k = s, s+1, \ldots, s+q-1 \end{array}$$

[**if** $x_i' \neq 0$ **then** $t \leftarrow z$]

Denote by δ_k the net change (positive, negative, zero) in x'_{j_k} caused by the program segment β_k. Thus the net change in x'_{j_s} caused by $\gamma_1 \cdots \gamma_s$ is $\sum_{k=1}^{s} \delta_k$. Corresponding to a fixed choice of initial values for $h_2, \ldots, h_{s+q}$ such that $1 \geq h_{s+q} \geq h_{s+q-1} \geq \cdots \geq h_2 \geq 0$, three cases arise:

(a) $h_{s+1} = 0$. Then in F_4 w is initialized to 0.

(b) $h_{s+1} = 1$ and $\sum_{k=1}^{s} \delta_k \geq 0$. For x_i' to have value 0 after executing F_4, x_{j_s}' must be no greater than $\text{abs}(\sum_{k=s+1}^{s+q} \delta_k)$ upon exiting the **do** $w \ldots$ **end** code segment. Thus in F_4 w must be initialized to a value no greater than $\text{abs}(\sum_{k=s+1}^{s+q} \delta_k) \leq qr$.

(c) $h_{s+1} = 1$ and $\sum_{k=1}^{s} \delta_k < 0$. Again, for x_i' to have value 0 after executing F_4, x_{j_s}' must be no greater than $\text{abs}(\sum_{k=s+1}^{s+q} \delta_k)$ upon leaving the **do** $w \ldots$ **end** code segment. Thus if x_{j_s}' has value v on entering the **do**, w must be initialized so that

$$\left\lfloor \frac{v \dot{-} sr \dot{-} \text{abs}(\sum_{k=s+1}^{s+q} \delta_k)}{-\sum_{k=1}^{s} \delta_k} \right\rfloor \leq w \leq \left\lceil \frac{v}{-\sum_{k=1}^{s} \delta_k} \right\rceil$$

(sr which is an upper bound on the number of instructions in $\gamma_1 \cdots \gamma_s$ is subtracted from v to avoid "boundary problems"). Note that

$$\left\lfloor \frac{v \dot{-} sr \dot{-} \text{abs}(\sum_{k=s+1}^{s+q} \delta_k)}{-\sum_{k=1}^{s} \delta_k} \right\rfloor \geq \left\lceil \frac{v \dot{-} sr \dot{-} qr}{-\sum_{k=1}^{s} \delta_k} \right\rceil - 1$$
$$\geq \left\lceil \frac{v}{-\sum_{k=1}^{s} \delta_k} \right\rceil - (s+q)r - 2.$$

Moreover, $\lceil x_{j_s}'/(-\sum_{k=1}^{s} \delta_k) \rceil$ is SL-computable.

Now, there are exactly $s + q - 1$ possible choices of values for $h_2, \ldots, h_{s+q}$ satisfying $1 \geq h_{s+q} \geq h_{s+q-1} \geq \cdots \geq h_2 \geq 0$. From the lower and upper bounds for w above, to each of these choices there correspond at most $(s + q)r + 3 \leq nr + 3$ possible segments of the form F_4 which can leave x_i' with 0 value. The segments only differ in the code that initializes w. Each of these program segments is of size $\leq O(rn \log n)$. P starts a computation by simulating these program segments and saving in $t_1', \ldots, t_m'$ the corresponding output values given in t (m is an integer $\leq (s + q - 1)(nr + 3)$ and $t_1', \ldots, t_m'$ are new variables). Then P sets $t = \min\{t_1', \ldots, t_m'\}$ using a program segment of the form

$$\begin{array}{l}
[t \leftarrow 0] \\
\textbf{do } z \\
\quad t \leftarrow t + 1; \\
\quad \left.\begin{array}{l} t_k' \leftarrow t_k' \dot{-} 1 \\ \textbf{if } t_k' = 0 \textbf{ then exit} \end{array}\right\} \begin{array}{l}\text{repeat for} \\ k = 1, \ldots, m\end{array} \\
\textbf{end}
\end{array}$$

The portion of P that computes $t_1', \ldots, t_m'$ is of size $O((s + q - 1)(nr + 3)(rn \log n)) \leq O(r^2n^3 \log n)$, while the portion of P that computes $\min\{t_1', \ldots, t_m'\}$ is of size $\leq O(m \log m) \leq O(r^2n^3 \log n)$. Thus P is of size $\leq O(r^2n^3 \log n)$. It can also be verified that P is effectively constructable in $O(r^2n^3 \log n)$ time. □

We are now ready to prove the main lemma of this appendix.

LEMMA A4. *Every program in SL* $\cup$ $\{x \leftarrow y\}$ *of size N can be translated in* $O(N^6)$ *time into an equivalent SL-program.*

PROOF. Instructions of the form $x \leftarrow y$ that appear outside the **do** ... **end** constructs are replaced by SL-code segments of the form "**do** x; $x \leftarrow x \dot{-} 1$; **end**; **do** y; $x \leftarrow x + 1$; **end**". Now consider a computation by a **do** ... **end** code segment of size N_1 having the form

```
do z
  ρ_1
  if y_1 = 0 then exit
  .
  .
  .
  ρ_s
  if y_s = 0 then exit
  ρ_{s+1}
end
```

where $\rho_1, \ldots, \rho_{s+1}$ do not contain **if** statements. Let $x_1, \ldots, x_n$ be the only variables in $\rho_1, \ldots, \rho_{s+1}$. Without loss of generality we assume that any execution of this program segment is terminated by an **if** instruction. We describe the simulation of this **do** ... **end** code segment by an SL-program segment in four phases.

Phase 1. For each $i = 1, \ldots, s$ introduce a new variable t_i and initialize it to contain the number of times the ith **if** statement is encountered, disregarding the presence of the other **if** statements. By Lemma A3 the code can be constructed in $O(sr^2n^3 \log n)$ time, where r is the number of instructions in $\rho_1 \cdots \rho_{s+1}$.

Phase 2. Set $t = \min\{t_1, \ldots, t_s\}$ and

$$h_i = \begin{cases} 1 & \text{if} \quad t_i = t \quad \text{and} \quad t_i < \min\{t_1, \ldots, t_{i-1}\}, \\ 0 & \text{otherwise,} \end{cases}$$

$1 \le i \le s$, where $h_1, \ldots, h_s$ are new variables. The following code will do this.

```
do z
  t ← t + 1
  h_i ← h_i + 1          ⎫
  t_i ← t_i ∸ 1          ⎬ repeat for
  if t_i = 0 then exit   ⎪ i = 1, . . , s
  h_i ← h_i ∸ 1          ⎭
end
```

Obviously the code for this phase can be constructed in $O(s \log s)$ time.

Phase 3. In this phase the values of $x_1, \ldots, x_n$ are evaluated. Note that h_i contains the value 1 if the execution of the **do** ... **end** program segment being considered is terminated by the ith **if** statement; otherwise it contains 0. t contains the number of times the **do** ... **end** construct is iterated. Thus the values for $x_1, \ldots, x_n$ can be obtained by a code of the form

```
[w ← 0; if t ≠ 0 and h_i ≠ 0 then w ← 1] ⎫
do w                                      ⎪
  ρ_1                                     ⎪
  .                                       ⎪
  .                                       ⎪
  .                                       ⎪
  ρ_i                                     ⎪
end                                       ⎪
[w ← t ∸ 1, if h_i = 0 then w ← 0]        ⎬ repeat for
do w                                      ⎪ i = 1, . . . , s
  ρ_{i+1}                                 ⎪
  .                                       ⎪
  .                                       ⎪
  .                                       ⎪
  ρ_{s+1}                                 ⎪
  ρ_1                                     ⎪
  .                                       ⎪
  .                                       ⎪
  .                                       ⎪
  ρ_i                                     ⎪
end                                       ⎭
```

By Lemma A2, for any j, $1 \le j \le n$, we can use the above code to construct an SL-program segment which leaves a new variable x_j' with the value evaluated for x_j (while $x_1, \ldots, x_n$ remain unchanged). Each such SL-program segment can be constructed in $O(srn \log n)$ time. Hence the desired code for this phase can be constructed in $O(srn^2 \log n)$ time.

Phase 4. Introduce the program segment $[x_1 \leftarrow x_1'; \ldots; x_n \leftarrow x_n']$. This can be done in $O(n \log n)$ time.

Combining the time complexities, we have the bound of $O(sr^2n^3 \log n) \le O(sr^2n^2N_1) \le O(N_1^6)$. The result follows. □

REFERENCES

1. AHO, A., HOPCROFT, J., AND ULLMAN, J. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass., 1974.
2. BAKER, B., AND BOOK, R. Reversal-bounded multipushdown machines. *J. Comput. Syst. Sci. 8* (1974), 315–332.
3 CHERNIAVSKY, J.C. Simple programs realize exactly Presburger formulas. *SIAM J. Comput 5* (1976), 666–677
4 CHERNIAVSKY, J.C., AND KAMIN, S N. A complete and consistent Hoare axiomatics for a simple programming language *J. ACM 26*, 1 (Jan. 1979), 119–128.
5 CONSTABLE, R L , AND BORODIN, A B Subrecursive programming languages, part I· Efficiency and program structure *J ACM 19*, 3 (July 1972), 526–568.
6. CONSTABLE, R.L., HUNT, H III, AND SAHNI, S. On the computational complexity of scheme equivalence. Proc 8th Ann Princeton Conf. on Information Sciences Systems, Princeton, N.J., 1974.
7 COOK, S. The complexity of theorem proving procedures. Conf. Rec. 3rd ACM Symp. on Theory of Computing, Shaker Heights, Ohio, 1971, pp. 151–158.
8. FISCHER, M., AND RABIN, M. Super-exponential complexity of Presburger arithmetic. Project MAC Tech. Memo 43, M I T , Cambridge, Mass., 1974.
9. GINSBURG, S , AND SPANIER, E. Semigroups, Presburger formulas, and languages. *Pacific J. Math. 16* (1966), 285–296
10. GURARI, E M., AND IBARRA, O.H The complexity of the equivalence problem for two characterizations of Presburger sets. *Theor. Comput. Sci. 13* (1981), 295–314
11. HILBERT, D. Mathematische Probleme. Vortrag, gehalten auf dem internationalen Mathematiker-Kongress zu Paris 1900. *Nachr. Akad. Wiss Gottingen Math.-Phys.* (1900), 253–297 [English translation: *Bull. Am Math. Soc. 8* (1901-1902), 437–479]
12 IBARRA, O.H. Reversal-bounded multicounter machines and their decision problems *J. ACM 25*, 1 (Jan. 1978), 116–133
13 KARP, R Reducibility among combinatorial problems In *Complexity of Computer Computations*, R Miller and J. Thatcher, Eds., Plenum Press, New York, 1972, pp. 85–104.
14 MATIJASEVIC, Y. Enumerable sets are Diophantine. *Dodl Akad. Nauk. SSSR 191* (1970), 279–282
15 MEYER, A., AND RITCHIE, D. The complexity of loop programs. In *Proc. 22nd Nat. Conf. of the ACM*, Thompson Book Co., Washington, D.C , 1967, pp 465–469.
16 TSICHRITZIS, D The equivalence problem of simple programs *J ACM 17*, 4 (Oct 1970), 729–738.

RECEIVED AUGUST 1978, REVISED MAY 1980, ACCEPTED MAY 1980