JOURNAL OF COMPUTER AND SYSTEM SCIENCES 26, 47–64 (1983)

On the Zero-Inequivalence Problem for Loop Programs*

OSCAR H. IBARRA AND BRIAN S. LEININGER

Department of Computer Science, University of Minnesota, Minneapolis, Minnesota 55455

Received September 17, 1980; revised November 16, 1981

The complexity of the zero-inequivalence problem (deciding if a program outputs a nonzero value for some nonnegative integer input) for several simple classes of loop programs is studied. In particular, we show that the problem is NP-complete for L_1-programs with only one input variable and two auxiliary variables. These are programs over the instruction set $\{x \leftarrow 0,\ x \leftarrow x+1,\ x \leftarrow y,$ **do** $x \cdots$ **end**$\}$, where do-loops cannot be nested. For K_1-programs, where the instruction set is $\{x \leftarrow x+1,\ x \leftarrow x \dot{-} 1,$ **do** $x \cdots$ **end**$\}$, zero-inequivalence is NP-complete even for programs with only one input variable and one auxiliary variable. These results may be the best possible since there is a class of programs which properly contains two-variable L_1-programs and one-variable K_1-programs with a polynomial time decidable equivalence problem. Addition of other constructs, e.g., allowing K_1-programs to use instruction $x \leftarrow x + y$, makes the zero-inequivalence problem undecidable.

1. INTRODUCTION

For $i \geqslant 0$, let L_i be the class of programs using only instructions of the form $x \leftarrow 0$, $x \leftarrow x+1$, $x \leftarrow y$, and **do** $x \cdots$ **end**, where **do** $x \cdots$ **end** constructs can only be nested to maximum depth i. The construct **do** $x \cdots$ **end** causes the instructions inside the **do** to be executed m times, where m is the value of x just before the loop is entered. Two fixed (not necessarily disjoint) sets of program variables are designated input variables and output variables, respectively. Noninput variables are referred to as auxiliary variables, and they are initially set to 0. Variables can only assume nonnegative integer values.

Let L be the union of the L_i's. In [11] it is shown that L-programs compute exactly the primitive recursive functions, and the sequence $L_0, L_1, L_2, \ldots$ defines a hierarchy of primitive recursive functions. It is also shown in [11] that the equivalence problem for L_2-programs is undecidable. For L_1-programs, equivalence is decidable [13]; in fact, inequivalence is NP-complete [2, 13]. For L_0, equivalence is decidable in polynomial time [13].

In this paper, we consider four classes of programs related to L_i. For $i \geqslant 0$:

* This research was supported in part by National Science Foundation Grant MCS78-01736.

0022-0000/83/010047–18$03.00/0

571/26/1-4

(1) K_i is the class with instruction set $x \leftarrow x+1$, $x \leftarrow x \dot{-} 1$ (where $x \dot{-} y = x - y$ if $x \geqslant y$ and otherwise), and **do** $x \cdots$ **end**. (Thus K_i is L_i with $x \leftarrow 0$ deleted and $x \leftarrow y$ replaced by $x \leftarrow x \dot{-} 1$.)

(2) KL_i is L_i augmented by instruction $x \leftarrow x \dot{-} 1$. The class KL_1 was introduced in [1] and was shown to have complete and consistent Hoare axiomatics.

(3) Q_i is the class with instruction set $x \leftarrow x+1$, $x \leftarrow x+y$, $x \leftarrow x \dot{-} 1$, and **do** $x \cdots$ **end**. (Thus, Q_i is K_i augmented by instruction $x \leftarrow x+y$.)

(4) V_i is the class with instruction set $x \leftarrow x+1$, $x \leftarrow x+y$, $x \leftarrow x \dot{-} y$, and **do** $x \cdots$ **end.** (Thus, V_i is Q_i with $x \leftarrow x \dot{-} 1$ replaced by $x \leftarrow x \dot{-} y$.)

We show the following:

(1) The zero-inequivalence problem (given a program, does it output a positive value for some input?) for K_1-programs with only *one* input–output variable (i.e., the input variable is also the output variable) and *one* auxiliary variable is NP-complete. This strengthens a result in [6] which shows the NP-completeness of zero-inequivalence for three-variable KL_1-programs (called CL-programs in [6]).

(2) The zero-inequivalence problem for L_1-programs with only *one* input–output variable and *two* auxiliary variables is NP-complete. (In [6], NP-completeness is shown for four-variable L_1-programs.)

(3) A large subclass of KL_1-programs which properly contains one-variable K_1-programs and two-variable L_1-programs has a polynomial time decidable equivalence problem, where the (in)equivalence problem is the problem of deciding for two programs whether they are (in)equivalent. Thus (1) and (2) may be the best possible results. (KL_1 has an NP-complete inequivalence problem. In fact, the NP-completeness holds for KL_1-programs augmented by instructions **goto** l and **if** $x=0$ **then goto** l, where l is a "forward" label appearing outside do constructs [5]. If forward jumps inside **do** loops are allowed, zero-equivalence is undecidable [6].)

(4) The zero-equivalence problem for L_2-programs (respectively, K_2-programs) with only *one* input–output variable and *three* auxiliary variables is undecidable. This strengthens a result in [11].

(5) The zero-equivalence problem for Q_1-programs with *nine* input variables and *four* auxiliary variables is undecidable. This contrasts the decidability of equivalence for KL_1-programs.

(6) The zero-equivalence problem for V_1-programs with *one* input–output variable and *three* auxiliary variables is undecidable.

Using the techniques in [11], it can be shown that for $i \geqslant 1$, K_{i+1}, L_{i+1}, KL_{i+1}, and V_i are equivalent languages.

2. Two-Variable K_1-Programs

The proof of the main result of this section uses the following theorem in [8] (the theorem without the "exactly three literals per clause" requirement follows directly from results of Cook [3] and Gold [4]):

THEOREM 1. *The satisfiability problem for Boolean formulas in* CNF *with exactly three literals per clause, where each clause contains either all negated variables or all unnegated variables, is* NP-*hard.*

To simplify our discussion, we introduce an intermediate language which limits our programs to straight-line codes. This new language has instruction set $x \leftarrow 1$, $x \leftarrow 2x$, $x \leftarrow x + y$, $x \leftarrow x \dot{-} y$, and $x \leftarrow \text{norm}(x)$, where $\text{norm}(x) = 1$ if $x > 0$ and 0 otherwise. Norm(x) is also called signum(x) in the literature.

The inequivalence (and, hence, the zero-inequivalence) problem for KL_1-programs is NP-complete [5]. The results in this section and in Section 3 concern programs that can easily be transformed in polynomial time to equivalent KL_1-programs. It follows that the inequivalence (and, hence, the zero-inequivalence) problem for these programs is in NP. Therefore, in the proofs of the results only NP-hardness will be shown.

THEOREM 2. *The zero-inequivalence problem for* $\{x \leftarrow 1,\ x \leftarrow 2x,\ x \leftarrow x + y,\ x \leftarrow x \dot{-} y,\ x \leftarrow \text{norm}(x)\}$*-programs with one input–output variable and one auxiliary variable is* NP-*complete. The result holds even if x and y are required to be distinct in the construct* $x \leftarrow x \dot{-} y$.

Proof. Let F be a Boolean formula in conjunctive normal form with clauses $C_1, \ldots, C_m$ and variables $x_1, \ldots, x_n$. Assume that each C_k contains exactly 3 negated or 3 unnegated variables. By Theorem 1, deciding if such a formula is satisfiable is NP-hard. We shall construct a program P_F which computes a nonzero function if and only if F is satisfiable. Program P_F has input–output variable x and auxiliary variable y. The bits of x represent the assignment of values to the variables of F, where 1 represents "true" and 0 represents "false." Program P_F has the form

$$
\begin{array}{l}
S \\
P_n \\
P_{n-1} \\
\vdots \\
P_1 \\
Q_m \\
Q_{m-1} \\
\vdots \\
Q_1
\end{array}
$$

Program Segment S. The program segment S first checks if $x \leqslant 2^n \dot{-} 1$. If $x \leqslant 2^n \dot{-} 1$, it leaves this value unchanged. If $x > 2^n \dot{-} 1$, S puts the value $2^n \dot{-} 1$ in x. Then x is replaced by $x2^{2m+1} + 1$.

$$
\begin{array}{ll}
y \leftarrow 1 & \\
x \leftarrow x + y & \\
y \leftarrow 2^n & \text{coded: } \; y \leftarrow 1; \overbrace{y \leftarrow 2y; \ldots; y \leftarrow 2y}^{n} \\
y \leftarrow y \dot{-} x & \\
x \leftarrow 1 & \\
y \leftarrow y + x & \\
x \leftarrow 2^n & \\
x \leftarrow x \dot{-} y & \\
x \leftarrow x2^{2m+1} & \text{coded: } \; x \leftarrow 2x; \ldots; x \leftarrow 2x \quad (2m+1 \text{ times}) \\
y \leftarrow 1 & \\
x \leftarrow x + y &
\end{array}
$$

Program Segment P_i $(i = n, \; n-1, \ldots, 1)$. On entering P_i, $x = x_i \cdots x_1 d_{2m} d_{2m-1} \cdots d_2 d_1 1$. P_i extracts $\bar{x}_i$, adds $\bar{x}_i$ to $d_{2k} d_{2k-1}$ if x_i or $\bar{x}_i$ appears in C_k, and deletes x_i from x. Let x_i or $\bar{x}_i$ appear in clauses $C_{i_1}, \ldots, C_{i_r}$, where $1 \leqslant i_1 < i_2 < \cdots < i_r \leqslant m$.

$$
\begin{array}{ll}
y \leftarrow 2^{2m+i} & \\
y \leftarrow y \dot{-} x & y = 0 \text{ if } x_i = 1; \quad y > 0 \text{ if } x_i = 0 \\
y \leftarrow \mathrm{norm}(y) & y = \bar{x}_i \\
y \leftarrow y2^{2i_1 - 1} & \\
x \leftarrow x + y & \text{Add } \bar{x}_i \text{ to } d_{2i_1} d_{2i_1 - 1} \\
y \leftarrow y2^{2(i_2 - i_1)} & \\
x \leftarrow x + y & \text{Add } \bar{x}_i \text{ to } d_{2i_2} d_{2i_2 - 1} \\
\vdots & \\
y \leftarrow y2^{2(i_r - i_{r-1})} & \\
x \leftarrow x + y & \text{Add } \bar{x}_i \text{ to } d_{2i_r} d_{2i_r - 1} \\
y \leftarrow y2^{2m+i+1-2i_r} & y = \bar{x}_i 2^{2m+i} \\
x \leftarrow x + y & x = 1x_{i-1} \cdots x_1 d_{2m} d_{2m-1} \cdots d_2 d_1 1 \\
y \leftarrow 2^{2m+i} & \\
x \leftarrow x \dot{-} y & x = x_{i-1} \cdots x_1 d_{2m} \cdots d_2 d_1 1.
\end{array}
$$

Program Segment Q_k $(k = m, \; m-1, \ldots, 1)$. After executing code segment P_1, $x = d_{2m} d_{2m-1} \cdots d_2 d_1 1$, where $d_{2k} d_{2k-1} = \bar{x}_{k_1} + \bar{x}_{k_2} + \bar{x}_{k_3}$ if $C_k = x_{k_1} + x_{k_2} + x_{k_3}$ or $C_k = \bar{x}_{k_1} + \bar{x}_{k_2} + \bar{x}_{k_3}$. If $C_k = x_{k_1} + x_{k_2} + x_{k_3}$, then C_k is false if and only if $x_{k_1} = x_{k_2} = x_{k_3} = 0$ and, hence, if and only if $d_{2k} d_{2k-1} = 11$ (in binary). If $C_k = \bar{x}_{k_1} + \bar{x}_{k_2} + \bar{x}_{k_3}$, then C_k is false if and only if $d_{2k} d_{2k-1}$ is 00. Let $x = d_{2k} d_{2k-1} \cdots d_2 d_1 l$ on entering Q_k. If C_k is true, Q_k does not change l but deletes $d_{2k} d_{2k-1}$ from x. If C_k is false, Q_k sets x to zero, and therefore, l to zero. This does not change the truth value of F since it is already zero and once l becomes zero l

cannot become one again. If $C_k = x_{k_1} + x_{k_2} + x_{k_3}$, then Q_k is given by program segments Q_k^1 and Q_k^2 below. If $C_k = \bar{x}_{k_1} + \bar{x}_{k_2} + \bar{x}_{k_3}$, then Q_k is only Q_k^2.

$$
Q_k^1 \left\{ \begin{array}{ll}
y \leftarrow 2^{2k-1} & \\
x \leftarrow x + y & x = pd_{2k}d_{2k-1} \cdots d_2 d_1 l. \;\; d_{2k}d_{2k-1} = 00 \text{ if } C_k = 0; \\
 & d_{2k}d_{2k-1} \neq 00 \text{ if } C_k = 1. \\
y \leftarrow y2^2 & \\
y \leftarrow y \dot{-} x & y = 0 \text{ if } p = 1; \; y > 0 \text{ if } p = 0 \\
y \leftarrow \text{norm}(y) & y = \bar{p} \\
y \leftarrow y2^{2k+1} & \\
x \leftarrow x + y & x = 1d_{2k}d_{2k-1} \cdots d_2 d_1 l \\
y \leftarrow 2^{2k+1} & \\
x \leftarrow x \dot{-} y & x = d_{2k}d_{2k-1} \cdots d_2 d_1 l. \\
 & d_{2k}d_{2k-1} \neq 00 \text{ if } C_k = 1; \;\; d_{2k}d_{2k-1} = 00 \text{ if } C_k = 0
\end{array} \right.
$$

$$
Q_k^2 \left\{ \begin{array}{ll}
y \leftarrow 2^{2k-1} & \\
y \leftarrow y \dot{-} x & y = 0 \text{ if } C_k = 1; \;\; y > 0 \text{ if } C_k = 0 \\
y \leftarrow y2^{2k-1} & y = 0 \text{ if } C_k = 1; \;\; y \geqslant 2^{2k-1} \text{ if } C_k = 0. \\
 & \text{Note that if } C_k = 0, \text{ then } x < 2^{2k-1} \\
x \leftarrow x \dot{-} y & x = 0, \text{ i.e., } l \text{ becomes } 0 \text{ if } C_k = 0 \\
y \leftarrow 2^{2k} & \\
y \leftarrow y \dot{-} x & \\
y \leftarrow \text{norm}(y) & y = \bar{d}_{2k} \\
y \leftarrow y2^{2k} & \\
x \leftarrow x + y & x = 1d_{2k-1} \cdots d_1 l \\
y \leftarrow 2^{2k} & \\
x \leftarrow x \dot{-} y & x = d_{2k-1} \cdots d_1 l \\
y \leftarrow 2^{2k-1} & \\
y \leftarrow y \dot{-} x & \\
y \leftarrow \text{norm}(y) & y = \bar{d}_{2k-1} \\
y \leftarrow y2^{2k-1} & \\
x \leftarrow x + y & x = 1d_{2k-2} \cdots d_1 l \\
y \leftarrow 2^{2k-1} & \\
x \leftarrow x \dot{-} y & x = d_{2k-2} \cdots d_1 l.
\end{array} \right.
$$

After executing code Q_1, x has value 1 if F is satisfied for the initial value of x; otherwise, x has value 0. Hence, P_F computes a nonzero function if and only if F is satisfiable. ∎

We can now prove the main result of this section.

THEOREM 3. *The zero-inequivalence problem for K_1-programs with one input–output variable and one auxiliary variable is* NP*-complete.*

Proof. The instructions $x \leftarrow 1$, $x \leftarrow 2x$, $x \leftarrow x + y$, $x \leftarrow x \dot{-} y$ are easily coded in K_1. The instruction $x \leftarrow \text{norm}(x)$ is equivalent to the K_1-code

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

The result now follows from Theorem 2. ∎

We conclude this section with two interesting corollaries to Theorem 2.

COROLLARY 1. *The zero-inequivalence problem for* $\{x \leftarrow 1, x \leftarrow 2x, x \leftarrow x + 1, x \leftarrow x + y, x \leftarrow x \dot{-} y, x \leftarrow y \dot{-} x\}$*-programs with one input–output variable and one auxiliary variable is* NP*-complete.*

Proof. Replace the occurrences of the instruction $y \leftarrow \text{norm}(y)$ in the program P_F of Theorem 2 by the code

$$
\begin{aligned}
&y \leftarrow y 2^{2m+n+1}\\
&y \leftarrow x \dot{-} y\\
&y \leftarrow x \dot{-} y\\
&y \leftarrow y + 1\\
&y \leftarrow y \dot{-} x \quad \blacksquare
\end{aligned}
$$

COROLLARY 2. *The zero-inequivalence problem for* $\{x \leftarrow 1, x \leftarrow x + y, x \leftarrow x \dot{-} y\}$*-programs with one input–output variable and two auxiliary variables is* NP*-complete.*

Proof. By Corollary 1, we need only show how $x \leftarrow x + 1$, $x \leftarrow 2x$, and $x \leftarrow y \dot{-} x$ can be computed using a third variable, z. The code for $x \leftarrow x + 1$ is $z \leftarrow 1$; $x \leftarrow x + z$. The code for $x \leftarrow 2x$ is $z \leftarrow 1$; $z \leftarrow z + x$; $z \leftarrow z + x$; $x \leftarrow 1$; $x \leftarrow x + z$; $z \leftarrow 1$; $x \leftarrow x \dot{-} z$; $x \leftarrow x \dot{-} z$. The instruction $x \leftarrow y \dot{-} x$ can be coded as $z \leftarrow 1$; $x \leftarrow x + z$; $z \leftarrow z + y$; $z \leftarrow z \dot{-} x$; $x \leftarrow 1$; $x \leftarrow x + z$; $z \leftarrow 1$; $x \leftarrow x \dot{-} z$. ∎

3. THREE-VARIABLE L_1-PROGRAMS

In this section we show that the zero-inequivalence problem for L_1-programs with one input–output variable and two auxiliary variables is NP-complete. In Section 4, we shall show that the equivalence problem for two-variable L_1-programs is decidable in polynomial time.

We need the following result which was shown in [7]:

THEOREM 4. *It is an* NP*-hard problem to determine if an arbitrary* $\{x \leftarrow 0, x \leftarrow 1,$

$x \leftarrow 2x$, $x \leftarrow x+y$, $x \leftarrow y/2$, $x \leftarrow \mathrm{rem}(y/2)\}$-program with one input–output variable and one auxiliary variable outputs 0 *for some input.* (*Here $y/2$ is integer division and* $\mathrm{rem}(y/2) =$ *remainder of y divided by* 2.)

THEOREM 5. *The zero-inequivalence problem for L_1-programs with one input–output variable and two auxiliary variables is* NP-*complete.*

Proof. Let P be a two-variable $\{x \leftarrow 0,\ x \leftarrow 1,\ x \leftarrow 2x,\ x \leftarrow x+y,\ x \leftarrow y/2,\ x \leftarrow \mathrm{rem}(y/2)\}$-program. Clearly, the instructions $x \leftarrow 0$, $x \leftarrow 1$, $x \leftarrow 2x$, $x \leftarrow x+y$ are easily coded in L_1. The following code computes $x \leftarrow y/2$ (z is a new variable):

```
x ← 0
z ← 0
do y
   y ← x
   x ← z
   z ← y
   z ← z + 1
end
```

At the end of the **do** loop, $x = y/2$ and $z = (y+1)/2$

```
y ← 0
y ← y + x
y ← y + z
```

restores value of y.

Similary, the code for $x \leftarrow \mathrm{rem}(y/2)$ is

```
z ← y
x ← 0
do z
   y ← x
   x ← z
   z ← y
end
```

Let v be the value of y before the **do** loop. If v is odd, then after the loop, $y = z = 0$ and $x = v$. If v is even, then after the loop $y = z = v$ and $x = 0$.

```
do x
   y ← y + 1
   x ← 0
   x ← x + 1
end
```

After the loop $y = v$ and $x = \mathrm{rem}(v/2)$.

Also, we can construct the following code:

```
y ← 0
y ← y + 1
do x
   y ← 0
end
x ← y
```

If $x > 0$ before the **do** loop, then $x = 0$ at the end of the code. If $x = 0$ before the **do** loop, then $x = 1$ at the end of the code.

If follows that we can construct from P a three-variable program P' such that P' computes a nonzero function if and only if P outputs 0 for some input. The result now follows from Theorem 4. ∎

4. A Class with a Polynomial Time Decidable Equivalence Problem

KL_1 has an NP-complete inequivalence problem. In fact, the NP-completeness holds for KL_1-programs augmented by instructions **goto** l and **if** $x = 0$ **then goto** l, where l is a "forward" label appearing outside **do** constructs [5]. If forward jumps inside **do** loops are allowed, zero-equivalence is undecidable [6].

In this section, we define a subclass of KL_1-programs for which the equivalence problem is decidable in polynomial time. This subclass contains (as special cases) one-variable K_1-programs and two-variable L_1-programs.

Let α be a program code containing only instructions of the form $x \leftarrow 0$, $x \leftarrow x + 1$, $x \leftarrow x \dot{-} 1$, $x \leftarrow y$. Let $x_1, \ldots, x_n$ be the variables in α (i.e., the variables appearing on the left or right sides of instructions in α). In [5] (see also [1]), an algorithm is given which constructs for each variable x_i in α a code $R(x_i)$ which is equivalent to α with respect to variable x_i (i.e., $R(x_i)$ and α have the same effect on x_i), and $R(x_i)$ has the form

$$\begin{array}{l} x_i \leftarrow x_j \\ \beta_i \end{array}$$

where $1 \leqslant j \leqslant n$ (possibly the same as i) and B_i is a (possibly empty) program segment containing only instructions of the form $x_i \leftarrow 0$, $x_i \leftarrow x_i + 1$, $x_i \leftarrow x_1 \dot{-} 1$. Moreover, the algorithm runs in time polynomial in the length of α.

We denote by $F(x_i)$ the instruction $x_i \leftarrow x_j$ and by $T(x_i)$ the code β_i. Clearly, $T(x_i)$ is empty if and only if x_i does not appear on the left side of any instruction in α. Let $g(T(x_i))$ = number of instructions of the form $x_i \leftarrow x_i + 1$ minus the number of instructions of the form $x_i \leftarrow x_i \dot{-} 1$. If $T(x_i)$ is empty, then $g(T(x_i)) = 0$.

We now define a class of programs which is a subset of KL_1-programs. Recall that KL_1-programs can only contain instructions of the form $x \leftarrow 0$, $x \leftarrow x + 1$, $x \leftarrow x \dot{-} 1$, $x \leftarrow y$, and **do** $x\,\alpha$ **end** with no nesting of **do**-loops (i.e., α can only contain instructions of the first four types).

Definition. Let D be the class of KL_1-programs in which each **do** z α **end** construct satisfies the following conditions:

(1) For each variable x in α, if $F(x) = x \leftarrow x$, then one of the following holds:

(i) z is the same variable as x,

(ii) $T(x)$ contains $x \leftarrow 0$,

(iii) $g(T(x)) \geqslant 0$.

(2) For distinct variables x and y in α, if $F(x) = x \leftarrow y$, then $F(y) = y \leftarrow y$. For each positive integer n, let $D(n)$ denote the class of D-programs with n *input* variables. (Note that there is no restriction on the number of output and auxiliary variables.)

The following proposition is obvious:

PROPOSITION 1. *There is a polynomial-time algorithm to determine if an arbitrary KL_1-program is a D-program.*

We shall show that for a fixed n, $D(n)$ has a polynomial-time decidable equivalence problem. Now one-variable K_1-programs and two-variable L_1-programs are clearly contained in $D(2)$. Hence, the equivalence problem for such programs is also polynomial time decidable.

LEMMA 3. *Let P be a program in $D(n)$. (Thus, P has n input variables.) Let r be the number of lines of code of P. Suppose that the set of input variables is partitioned into two sets: $x_1,..., x_m$ and $x_{m+1},..., x_n$ $(0 \leqslant m \leqslant n)$ such that variables $x_1,..., x_m$ are initialized to values $x_1^0 \geqslant 2r,..., x_m^0 \geqslant 2r$ while variables $x_{m+1},..., x_n$ are initialized to fixed values $b_{m+1} < 2r,..., b_n < 2r$. Then for each variable x, we can obtain, in polynomial time, an expression representing the value of x after the execution of P. The expression is of the form k for some nonnegative integer k, or of the form $c + a_1 x_{i_1}^0 + \cdots + a_t x_{i_t}^0$, where $1 \leqslant i_1 < i_2 < \cdots < i_t \leqslant m$, c is a (positive, negative, zero) integer, and $a_1,..., a_t$ are positive integers. Moreover, $c + a_1 x_{i_1}^0 + \cdots + a_t x_{i_t}^0 \geqslant r$.*

Proof. The proof is by induction. At the start of the program, input variables $x_1,..., x_m$ have expressions $x_1^0,..., x_m^0$ while input variables $x_{m+1},..., x_n$ have expressions $b_{m+1},..., b_n$. All other variables have value 0. Proceeding by induction, suppose that we have already obtained expressions for all variables at the end of the $(i-1)$th instruction. Consider the ith instruction.

(1) If the ith instruction is of the form $x \leftarrow 0$, then at the end of the ith instruction, the new expression for x is 0.

(2) If the ith instruction is of the form $x \leftarrow x + 1$ and the expression for x at the beginning of the ith instruction is k or $c + a_1 x_{i_1}^0 + \cdots + a_t x_{i_t}^0$, then the new expression for x is $k + 1$ or $(c+1) + a_1 x_{i_1}^0 + \cdots + a_t x_{i_t}^0$, respectively.

(3) If the ith instruction is of the form $x \leftarrow x \dot{-} 1$ and the expression for x at the beginning of the ith instruction is k or $c + a_1 x_{i_1}^0 + \cdots + a_t x_{i_t}^0$, then the new expression for x is $k \dot{-} 1$ or $(c-1) + a_1 x_{i_1}^0 + \cdots + a_t x_{i_t}^0$, respectively. Note that in the second form, proper subtraction can be replaced by regular subtraction since, as we shall see, the resulting expression is always $\geqslant r$.

(4) If the ith instruction is $x \leftarrow y$, then the new expression for x is the same as that of y.

(5) Suppose the ith instruction is of the form

do z
  α
end

From the expression for z, we can determine if $z = 0$ or $z = 1$. If $z = 0$, there is nothing to do and we can proceed to the $(i+1)$th instruction. If $z = 1$, we can update the expressions for the variables by executing α once. So assume $z \geqslant 2$. We describe how to update the expression for a variable x in α. We consider two cases.

Case 1

Suppose $R(x)$ has the form $\overset{x \leftarrow x}{\beta}$. Then

do z
  α
end

has the same effect on x as

do z
  β
end

If β contains an instruction $x \leftarrow 0$, then the expression for the new value of x at the end of the loop can be uniquely determined independent of the value of z: obtain the expression by simulating one iteration of β. If β does not contain $x \leftarrow 0$, then the net effect of β can be simulated by a single instruction of the form $x \leftarrow x + d_1$ or by a single instruction of the form $x \leftarrow x \dot{-} d_1$ or by two instructions of the form $x \leftarrow x \dot{-} d_1$; $x \leftarrow x + d_2$ (d_1, d_2 are nonegative integer constants). We consider three subcases.

Subcase 1. Suppose $g(T(x)) \geqslant 0$ and before the loop $x = c + a_1 x_{i_1}^0 + \cdots + a_t x_{i_t}^0$. This corresponds to β having a net effect $x \leftarrow x + d_1$ or $x \leftarrow x \dot{-} d_1$; $x \leftarrow x + d_2$ with $d_2 \geqslant d_1$. Then the new expression for x after the loop is $c + a_1 x_{i_1}^0 + \cdots + a_t x_{i_t}^0 + d_1 z$ or $c + a_1 x_{i_1}^0 + \cdots + a_t x_{i_t}^0 + (d_2 - d_1)z$, respectively.

Subcase 2. Suppose $g(T(x)) \geqslant 0$ and before the loop $x = k$. Again this corresponds to β having a net effect $x \leftarrow x + d_1$ or $x \leftarrow x \dot{-} d_1$; $x \leftarrow x + d_2$ with $d_2 \geqslant d_1$. Then the new expression for x is $k + d_1 z$ or $(k \dot{-} d_1) + (z-1)(d_2 - d_1) + d_2$.

Subcase 3. Suppose $g(T(x)) < 0$ and before the loop $x = k$ or $x = c + a_1 x_{i_1}^0 + \cdots + a_t x_{i_t}^0$. This corresponds to β having the net effect $x \leftarrow x \dot{-} d_1$ or $x \leftarrow x \dot{-} d_1$; $x \leftarrow x + d_2$ with $d_1 > d_2$. By the definition of class D, z must be the same variable as x. Hence, the new value of x after the loop is 0 or d_2, respectively.

Case 2

Suppose $R(x)$ is of the form ${}^{x\leftarrow y}_{\beta}$. Then by the definition of class D, it must be the case that $R(y)$ is of the form ${}^{y\leftarrow y}_{\beta'}$. Hence

$$\begin{array}{l}\textbf{do } z\\ \quad \alpha\\ \textbf{end}\end{array}$$

has the same effect on x as the code

$$\left.\begin{array}{l} z \leftarrow z \dot{-} 1\\ \textbf{do } z\\ \quad \beta'\\ \textbf{end}\\ x \leftarrow y\\ \quad \beta \end{array}\right\} \quad (*).$$

The new expression for x as the result of executing (*) can be found as in (1)–(4) and case 1 of (5). ∎

We are now ready to prove the following result:

THEOREM 6. *Let n be a fixed positive integer. Then the equivalence problem for $D(n)$-programs is decidable in polynomial time.*

Proof. Let P be a program in $D(n)$. (Thus, P has n input variables.) Let r be the number of lines of code of P. Referring to Lemma 3, there are at most $K = O((2r+1)^n)$ different ways to partition the input variables of P into two sets and to assign fixed values to the b_i's. By Lemma 3, for each choice we can obtain (in polynomial time) unique arithmetic expressions representing the final values of all variables in P. Since n is fixed, it follows that we can decide equivalence of two programs P_1 and P_2 in polynomial time. ∎

Now one-variable K_1-programs and two-variable L_1-programs are clearly $D(2)$-programs. Hence, we have

COROLLARY 3. *The equivalence problem for one-variable K_1-programs is decidable in polynomial time.*

COROLLARY 4. *The equivalence problem for two-variable L_1-programs is decidable in polynomial time.*

Let L_1' be L_1 with the instruction $x \leftarrow y$ deleted. Clearly, $L_1' \subseteq D$. Then by Theorem 6, we have the following corollary which was first shown in [6]:

COROLLARY 5. *Let n be a fixed positive integer n. Then the equivalence problem for L_1'-programs with at most n input variables is decidable in polynomial time.*

When the number of input variables is not fixed, the zero-inequivalence problem for L_1'-programs is NP-complete [2].

The next result shows that equivalence is decidable in polynomial time for L_1-programs which use only instructions of the form $x \leftarrow x + 1$ and **do** $x \cdots$ **end**.

THEOREM 8. *Let L_1'' be L_1 with the instructions $x \leftarrow 0$ and $x \leftarrow y$ deleted. (Thus, L_1'' has the instruction set: $x \leftarrow x + 1$ and* **do** $x \cdots$ **end**). *Then L_1'' has a polynomial time decidable equivalence problem.*

Proof. If P is an L_1''-program with input variables $x_1, \ldots, x_n$, then the value of any variable y at the end of the program can be written uniquely as a linear combination $y = a_1 x_1 + \cdots + a_n x_n + b$, where $a_1, \ldots, a_n, b$ are nonnegative integers. The expression can be obtained in polynomial time. ∎

We conclude this section with Theorem 9 which is easily verified.

THEOREM 9. *The equivalence problem for KL_0-programs is decidable in polynomial time.*

5. FOUR-VARIABLE L_2-PROGRAMS AND K_2-PROGRAMS

It is known that the equivalence problem for L_2-programs is undecidable [11]. Here we strengthen this result by showing that the zero-equivalence problem for four-variable L_2-programs is undecidable. A similar result holds for K_2-programs. We shall need the following well-known result [12]:

THEOREM 10. *Let M be the class of programs containing only instructions of the form $x \leftarrow x + 1$, $x \leftarrow x \dot{-} 1$,* **goto** *l*, **if** $x = 0$ **then goto** *l, and* **halt**. *(Any statement may be labeled.) The halting problem for two-variable M-programs (i.e., does a program halt when the inputs are initially set to zero?) is undecidable.*

THEOREM 11. *The zero-equivalence problem for L_2-programs with one input–output variable and three auxiliary variables is undecidable.*

Proof. We use Theorem 10. Let P be a two-variable M-program. We may assume without loss of generality that P has exactly one **halt** instruction which appears at the end of the program, and this instruction is executed if and only if P halts. We may also assume that every instruction in P is labeled and the instructions are sequentially labeled. Thus P has the form

$$\begin{array}{c}\beta_1\\ \vdots\\ \beta_n\end{array}$$

where β_n is n: **halt** and for $1 \leqslant i \leqslant n-1$, β_i is of the form i: $x \leftarrow x+1$; i: $x \leftarrow x \dot{-} 1$; i: **goto** l; or i: **if** $x=0$ **then goto** l. Let x and y be the variables of P.

We shall construct an L_2-program P' with input–output variable w and auxiliary variables x, y, and z such that P' computes the zero-function if and only if P does not halt with x and y intially set to 0. Program P' has the form

```
w ← w + 1
z ← z + 1                z = 1
do w
  α_1
  ⋮
  α_n
end
z ← z ∸ (n − 1)          coded   z ← z ∸ 1;...; z ← z ∸ 1   (n − 1 times)
w ← 0         ⎫
do z          ⎪
  w ← w + 1   ⎬   w = z
end           ⎭
```

In program P', the variable z is used as a counter to keep track of which statement in P we are simulating. Each α_i starts with the instruction $z \leftarrow z \dot{-} 1$. The statement β_i is then simulated if and only if $z=0$ after this $z \leftarrow z \dot{-} 1$ instruction. Now, by assumption β_n is executed if and only if P halts. The section of code α_n will put n in z if and only if β_n was to be executed. This will result in z having a value n if and only if P halted after making at most w backward jumps, where w is the input. Thus, after

```
do w
  α_1
  ⋮
  α_n
end
```

$z < n$ if and only if P did not halt after making w backward jumps. Thus, at the end of P', $w=z=0$ if and only if P did not halt. It follows that P' computes the zero-function if and only if P does not halt. We now describe the construction to the α_i's.

For $1 \leqslant i \leqslant n-1$, α_i is defined as follows:

(1) If β_i is i: $x \leftarrow x+1$, then α_i is

```
w ← 0         ⎫
do z          ⎪
  z ← w       ⎬   z ← z ∸ 1
  w ← w + 1   ⎪
end           ⎭
w ← 0     (*)
w ← w + 1
```

```
do z
   w ← 0              if z > 0 after (*), then w ← 0.
end
do w
   x ← x + 1          x ← x + 1 iff z = 0 after (*);
end                   otherwise, x is unchanged.
```

(2) If β_i is i: $x \leftarrow x \dot{-} 1$, then α_i is

```
z ← z ∸ 1
w ← 0          (*)
do z
   w ← 0
   w ← w + 1
end
do w
   x ← x + 1          x ← x + 1 iff z > 0 after (*);
end                   otherwise x is unchanged.
w ← 0
do x
   x ← w              x ← x ∸ 1.
   w ← w + 1
end
```

The net result of α_i is $x \leftarrow x \dot{-} 1$ iff $z = 0$ after (*); otherwise x is unchanged.

(3) If β_i is i: **goto** l, and $l > i$, then α_i is

```
z ← z ∸ 1
w ← l − i      (*)
do z
   w ← 0
end                   z ← l − i iff z = 0 after (*);
do w                  otherwise z is unchanged.
   z ← z + 1
end
```

(4) If β_i is i: **goto** l, and $l \leqslant i$, then α_i is the same as in (3) except that the instruction $w \leftarrow l - i$ is replaced by $w \leftarrow l + (n - i)$.

(5) If β_i is i: **if** $x = 0$ **then goto** l, and $l > i$, then α_i is

```
z ← z ∸ 1
w ← l − i      (*)
```

do z
  $w \leftarrow 0$
end
do x
  $w \leftarrow 0$
end
do w
  $z \leftarrow z + 1$
end
	$z \leftarrow l - i$ iff $z = 0$ after (*) and $x = 0$; otherwise z is unchanged.

(6) If β_i is i: **if** $x = 0$ **then goto** l, and $l \leqslant i$, then α_i is the same as in (5) except that the instruction $w \leftarrow l - i$ is replaced by $w \leftarrow l + (n - i)$.

Finally, the code for α_n is

```
z ← z ∸ 1
w ← n
do z
  w ← 0
end
do w
  z ← z + 1
end
```
∎

Similarly, we have

THEOREM 12. *The zero-equivalence problem for K_2-programs with one input/output variable and three auxiliary variables is undecidable.*

Proof. The proof is similar to that of Theorem 11. The only difference is in the construction of the α_i's which now have to be coded as K_1-programs (i.e., using only the constructs $x \leftarrow x + 1$, $x \leftarrow x \dot{-} 1$, **do** $x \cdots$ **end** without nesting of loops). To illustrate

(1) If β_i is i: $x \leftarrow x + 1$, then α_i is

```
z ← z ∸ 1
do w
  w ← w ∸ 1
end
w ← w + 1
do z
  w ← w ∸ 1
end
do w
  x ← x + 1
end
```

(2) If β_i is i: **goto** l, and $l > i$, then α_i is

$$
\begin{array}{l}
z \leftarrow z \dot{-} 1 \\
\textbf{do } w \\
\quad w \leftarrow w \dot{-} 1 \\
\textbf{end} \\
\left.\begin{array}{l} w \leftarrow w + 1 \\ \quad\vdots \\ w \leftarrow w + 1 \end{array}\right\} \; l - i \text{ times} \\
\textbf{do } z \\
\quad \left.\begin{array}{l} w \leftarrow w \dot{-} 1 \\ \quad\vdots \\ w \leftarrow w \dot{-} 1 \end{array}\right\} \; l - i \text{ times} \\
\textbf{end} \\
\textbf{do } w \\
\quad z \leftarrow z + 1 \\
\textbf{end}
\end{array}
$$

(3) If β_i is i: **if** $x = 0$ **then goto** l, and $l > i$, then α_i is the same as in (2) except that the following code is inserted between the last two **do** loops.

$$
\begin{array}{l}
\textbf{do } x \\
\quad \left.\begin{array}{l} w \leftarrow w \dot{-} 1 \\ \quad\vdots \\ w \leftarrow w \dot{-} 1 \end{array}\right\} \; l - i \text{ times} \\
\textbf{end} \quad \blacksquare
\end{array}
$$

6. The Undecidability of the Zero-Equivalence Problem for Q_1-Programs and V_1-Programs

We define Q_1-programs as K_1-programs which can use instructions of the form $x \leftarrow x + y$. While K_1-programs have an NP-complete inequivalence problem [5], we have, in contrast, the following negative result for Q_1-programs:

THEOREM 13. *The zero-equivalence problem for Q_1-programs with nine input variables and four auxiliary variables is undecidable. (Recall that Q_1-programs have instruction set $x \leftarrow x + 1$, $x \leftarrow x + y$, $x \leftarrow x \dot{-} 1$, and* **do** *$x \cdots$* **end**.)

Proof. We use the undecidability of Hilbert's tenth problem [10]. Let $F(x_1,..., x_n)$ be a Diophantine polynomial with $n = 9$ variables. Determining if such a polynomial has a nonnegative integer solution is undecidable. (The proof for $n = 13$ is contained in [10]. The reduction to $n = 9$ was reported in [9].) We can effectively construct for a given $F(x_1,..., x_n)$ a Q_1-program P_F with input variables $x_1,..., x_n$ and output variable y such that P_F outputs 0 for all inputs if and only if F has no solution. In addition to variables $x_1, \ldots, x_n$, y, program P_F need only use three other variables. We omit the construction which is straightforward. ∎

We do not know whether the number of input variables in Theorem 13 can be reduced. In the construction of P_F the 9 input variables represent the variables in the Diophantine polynomial, and it is an open problem whether Hilbert's tenth problem is undecidable for polynomials with fewer than 9 variables. For V_1-programs, however, we can prove

THEOREM 14. *The zero-equivalence problem for V_1-programs with one input–output variable and three auxiliary variables is undecidable. (Recall that V_1-programs have instruction set $x \leftarrow x + 1$, $x \leftarrow x + y$, $x \leftarrow x \dot{-} y$,* **do** *$x \cdots$* **end**.)

Proof. The program P' in the proof of Theorem 12 (see also the proof of Theorem 11) can easily be converted to a V_1-program using such transformations as

$$
\begin{array}{llll}
(1) & z \leftarrow z \dot{-} k & \text{translates to} & \begin{array}{l} w \leftarrow w \dot{-} w \\ \left.\begin{array}{l} w \leftarrow w + 1 \\ \quad\vdots \\ w \leftarrow w + 1 \end{array}\right\} k \\ z \leftarrow z \dot{-} w \end{array} \\[2ex]
(2) & \begin{array}{l} \textbf{do } w \\ \quad w \leftarrow w \dot{-} 1 \\ \textbf{end} \end{array} & \text{translates to} & w \leftarrow w \dot{-} w \\[2ex]
(3) & \begin{array}{l} \textbf{do } z \\ \quad \left.\begin{array}{l} w \leftarrow w \dot{-} 1 \\ \quad\vdots \\ w \leftarrow w \dot{-} 1 \end{array}\right\} k \\ \textbf{end} \end{array} & \text{translates to} & \left.\begin{array}{l} w \leftarrow w \dot{-} z \\ \quad\vdots \\ w \leftarrow w \dot{-} z \end{array}\right\} k \\[2ex]
(4) & \begin{array}{l} \textbf{do } w \\ \quad x \leftarrow x + 1 \\ \textbf{end} \end{array} & \text{translates to} & x \leftarrow x + w
\end{array}
$$

∎

REFERENCES

1. J. CHERNIAVSKY AND S. KAMIN, A complete and consistent Hoare axiomatics for a simple programming language, *J. Assoc. Comput. Mach.* **26** (1979), 119–128.
2. R. CONSTABLE, H. HUNT III, AND S. SAHNI, On the computational complexity of scheme equivalence, *in* "Proceedings of the 8th Annual Princeton Conference on Information Science Systems," Princeton, N.J., 1974.
3. S. COOK, The complexity of theorem proving procedures, *in* "Conference Record of the Third ACM Symposium on Theory of Computing," pp. 151–158, ACM, New York, N.Y., 1971.
4. E. GOLD, Complexity of automaton identification from given data, *Inform. and Control* **37** (1978), 302–320.

5. E. Gurari and O. Ibarra, The complexity of the equivalence problem for simple programs, *J. Assoc. Comput. Mach.* **28** (1981), 535–560.
6. E. Gurari and O. Ibarra, Some simplified undecidable and *NP*-hard problems for simple programs, *Theoret. Comput. Sci.* **17** (1982), 55–73.
7. O. Ibarra and B. Leininger, "On the Simplification and Equivalence Problems for Straight-Line Programs," University of Minnesota Computer Science Department Tech Report 79-21, September 1979.
8. O. Ibarra, B. Leininger, and S. Moran, On the complexity of simple arithmetic expressions, *Theoret. Comput. Sci.* **19** (1982), 17–28.
9. K. Manders and L. Adleman, NP-complete decision problems for quadratic polynomials, *in* "Eighth Annual ACM Symposium on Theory of Computing," pp. 23–29, Hershey, Pa., May 1976.
10. Y. Matijasevič and J. Robinson, Reduction of an arbitrary Diophantine equation to one in 13 unknowns, *Acta Arith.* **27** (1975), 521–553.
11. A. Meyer and D. Ritchie, The complexity of loop programs, *in* "Proceedings of the Twenty-Second National Conference of the ACM," pp. 465–469, Thompson, Washington, D.C., 1967.
12. M. Minsky, "Computation: Finite and Infinite Machines," Prentice–Hall, Englewood Cliffs, N.J., 1967.
13. D. Tsichritzis, The equivalence problem of simple programs, *J. Assoc. Comput. Mach.* **17** (1970), 729–738.