ON SIMPLE PROGRAMS WITH PRIMITIVE CONDITIONAL STATEMENTS

Oscar H. Ibarra and Louis E. Rosier

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

TR-232 September 1983

ON SIMPLE PROGRAMS WITH PRIMITIVE CONDITIONAL STATEMENTS[1]

Oscar H. Ibarra
Department of Computer Science
University of Minnesota
Minneapolis, MN 55455

and

Louis E. Rosier
Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712

[1]This research was supported in part by NSF Grants MCS 81-02853 and MCS 83-04756, The University Research Institute, The University of Texas at Austin and the IBM Corporation.

Table of Contents

ABSTRACT

This paper is concerned with the semantics (or computational power) of very simple loop programs over different sets of primitive instructions. In addition to new results, the paper clarifies and corrects some of the claims made in [10]. The new results improve earlier results in the literature. In particular, it is shown that an {x←0, x←x+1, x←y, do x ... end, if x=0 then y←z}-program which contains no nested loops can be transformed into an equivalent {x←0, x←x+1, x←y, do x ... end}-program (also without nested loops) in exponential time and space. This translation was earlier claimed to be doable in polynomial time, but this was subsequently shown to imply that PSPACE=PTIME. Consequently, the question of translatability was left unanswered. Also it is shown that the class of functions computable by {x←0, x←x+1, x←y, x←x÷1, do x ... end, if x=0 then x←c}-programs is exactly the class of Presburger functions. When the conditional instruction is changed to "if x=0 then x←y+1", then the class of computable functions is significantly enlarged, enough so, in fact, as to render many decision problems (e.g. equivalence) undecidable.

1. INTRODUCTION

An important area in computer science concerns itself with the semantics of programs. This topic covers the development of semantics for programs of varying complexity. In this paper we concern ourselves with the semantics of very simple loop programs over different sets of primitive instructions. The computational power of simple loop programs has been studied before as have some of the related decision problems (e.g. the equivalence problem) [1-3, 7-8, 10-12]. Let BB denote a set of primitive (non-looping) instructions, e.g. {x←0, x←x+1, x←y}. An L_i(BB)-program is a program of the form:

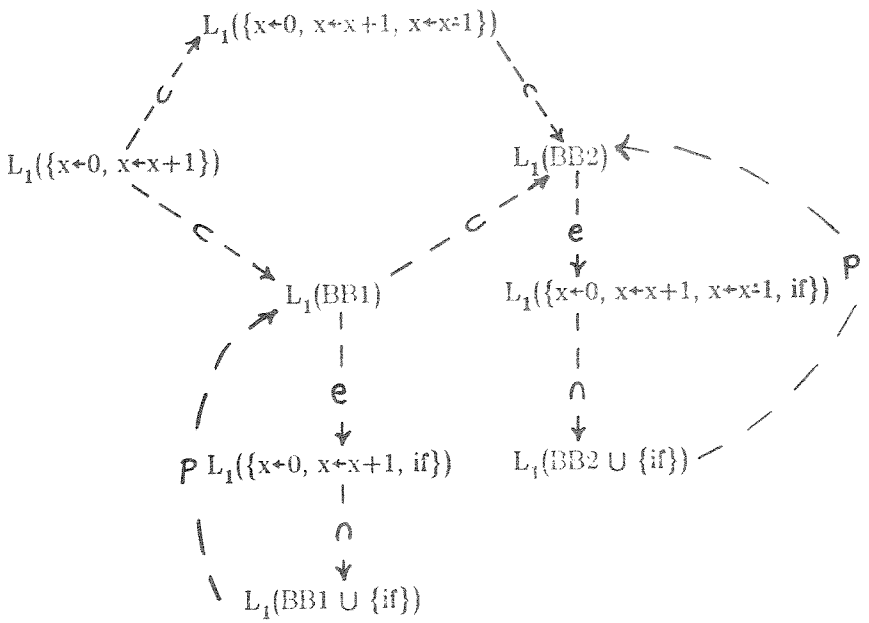
input(x_1,...,x_k);
A;
output(z_1,...,z_l);

where A is a block of instructions using only the constructs in BB and the construct "loop x ... end;". Furthermore the level of nesting allowed in the nested loop structures of A is at most i. The interpretation given to the "loop x ... end;" construct is that altering the contents of the loop control variable x inside the loop does not change the number of iterations executed. Let BB1={x←0, x←x+1, x←y}. The hierarchy of L_i(BB1)-programs is the subrecursive hierarchy of Meyer and Richie [11]. In [11] it was shown that L_2(BB1)-programs compute exactly the class of elementary recursive functions. L_1(BB1)-programs define the class of simple functions [12], a proper subclass of the Presburger functions [1,3]. Certain decision problems, e.g. the equivalence problem, were first shown to be decidable for the class of L_1(BB1)-programs in [12]. Subsequently, it was shown that L_1(BB1 ∪ {x←x÷1})-programs compute exactly the functions definable in Presburger arithmetic (and therefore the equivalence problem for this extension of L_1(BB1) remains decidable) [1,3]. Later the validity of simple correctness formulas of the form {p}S{q} were also shown to be decidable [2], where S is an L_1(BB1 ∪ {x←x÷1})-program and p and q are logical (Presburger) assertions about the variables in S. The formula {p}S{q} is said to be valid if for every input to S which satisfies p, it is the case that q is true following the execution of S. Other extensions of L_1(BB1) have also been studied [3, 10]. For example, Kfoury [10] considered variations of L_1(BB1), where the primitive instructions were to be drawn from the set {x←0, x←x+1, x←x÷1, x←y, if x=0 then A else B}, where A and B are (finite blocks) of the other primitive instructions in the set.

In this paper we investigate claims made (without proof) in [10], concerning L_1(BB)-programs where BB ⊆ {x←0, x←x+1, x←x÷1, x←y, if x=0 then A else B}. We paraphrase the following definitions from [10]. Let L and L' be classes of programs, and $\mathcal{L}$ and $\mathcal{L}'$ the corresponding classes of functions they compute. L is effectively translatable into L' if for every program P in L there is a constructive way to obtain a program P' in L' such that P and P' compute the same function. If there is such a translation we write L --*--> L', where the "*" may be replaced by "⊂", "l", "p" or "e", according to whether the translation is the trivial inclusion map or produces a program P' in L' which is of length at most "linear", "polynomial", or "exponential", in the length of P. Also for our results as well for the claims made in [10], whenever the translation procedure given is "l", "p", or "e", it is also the case that it will take at most "linear", "polynomial", or "exponential" time, respectively (as a function of the size of the source program P).

Let "if" denote the instruction "if x=0 then A else B". Let BB2 denote the set BB1 ∪ {x←x÷1}. The following theorem was claimed without proof in [10].

Theorem. Let γ_1 and γ_2 be subsets of {x←y, x←x÷1, if}. All possible translations from L_1({x←0, x←x+1} ∪ γ_1) to L_1({x←0, x←x+1} ∪ γ_2) can be read off the following diagram:



If an omitted arrow in the above diagram cannot be obtained by composition from the arrows already drawn, then it is the case of non-translatability, which also requires some proof. ■

This theorem probably contains an error concerning the translations:

$L_1(BB1)$ --e--> $L_1(\{x\leftarrow 0,\ x\leftarrow x+1,\ \text{if}\})$ --⊂--> $L_1(BB1 \cup \{\text{if}\})$ --p--> $L_1(BB1)$

and definitely contained an error concerning:

$L_1(BB2)$ --e--> $L_1(\{x\leftarrow 0,\ x\leftarrow x+1,\ x\leftarrow x\dot{-}1,\ \text{if}\})$ --⊂--> $L_1(BB2) \cup \{\text{if}\})$ --p--> $L_1(BB2)$

The remaining claims of the theorem are correct. From results in [7], it can be noted that $L_1(BB1 \cup \{\text{if}\})$ --p--> $L_1(BB1)$ implies PSPACE=PTIME. In [9], it was noted that $L_1(\{x\leftarrow 0,\ x\leftarrow x+1,\ x\leftarrow x\dot{-}1,\ \text{if}\})$-programs were computationally more powerful than $L_1(BB2)$-programs. This is an important jump in terms of computational power for two reasons. First the class of functions computable by such programs is no longer Presburger. Secondly, the jump is so great that most decision problems for such programs are now undecidable, e.g. the equivalence problem is undecidable and hence there no longer exists a decision procedure to decide the validity of even very simple correctness formulas. For example, the validity of correctness formulas of the form $\{true\}S\{x=y\}$ is no longer decidable. Left unanswered then is the question of translatability between $L_1(BB1 \cup \{\text{if}\})$ and $L_1(BB1)$.

In this paper we consider this problem as well as examine the computational gap between $L_1(BB2)$-programs and $L_1(BB2 \cup \{\text{if}\})$-programs. We concentrate on allowing the instructions "y←c" and "y←y∸1" to be conditionally executed. That is, we introduce the constructs "<u>if</u> x=0 <u>then</u> y←y+1", "<u>if</u> x=0 <u>then</u> y←y∸1" and "<u>if</u> x=0 <u>then</u> y←c", where c is any nonnegative integer constant. We are then able to show that:

$L_1(BB1 \cup \{\text{if}\})$ --e--> $L_1(BB1)$

which is perhaps not surprising but nevertheless had not been confirmed. This should be contrasted with

the corresponding situation for BB2, where the inclusion of the "if" construct provides an increase in computational power. We also show that:

L_1(BB2 $\cup$ {if x=0 then y←c}) --e--> L_1(BB2).

This should be contrasted with the result in [9], showing that L_1(BB2 $\cup$ {if x=0 then y←y+1}) is strictly more powerful than L_1(BB2). L_1(BB2 $\cup$ {if x=0 then y←y+1})-programs were shown to be computationally equivalent to L_1({x←0, x←x+1, x←x÷1, if})-programs, in [9]. If both constructs, "if x=0 then y←c" and "if x=0 then y←y÷1" are concurrently considered it is easy to show that L_1(BB2 $\cup$ {if x=0 then y←1, if x=0 then y←y÷1}) is computationally equivalent to L_1(BB2 $\cup$ {if x=0 then y←y+1}). Unfortunately, we are unable to resolve the computational power of L_1(BB2 $\cup$ {if x=0 then y←y÷1})-programs. The "if x=0 then y←y÷1" construct seems similar to the "if x=0 then y←y+1" construct, but as pointed out in [7], functions of one variable computed over BB2 $\cup$ {if x=0 then y←y÷1} are monotonic. Thus the proof techniques used in [9] do not seem to work with this language. This same problem was apparent in [7], where the authors were able to show that the 0-evaluation problem for this language is PSPACE-complete. Unfortunately, these techniques do not seem to work either. Lastly we note that the addition of the construct "if x=0 then y←z" to the set BB2 poses a difficult question. (It is easy to show that L_1(BB2 $\cup$ {if x=0 then y←z})-programs are computationally equivalent to L_1(BB2 $\cup$ {if})-programs.) If L_1(BB2 $\cup$ {if x=0 then y←z})-programs are computationally more powerful than L_1(BB2 $\cup$ {if x=0 then y←y+1})-programs, then it would imply that 0(n) space bounded Turing machines are more powerful than Turing machines operating simultaneously in 0(n) space and $0(2^{\lambda n})$ time, $\lambda<1$. This problem seems very difficult. The answer is not known even for the case when the time restriction is reduced to a polynomial. (See [9].) Other corrections to errors in [10] are presented in the last section.

2. L_1(BB)-Programs

In this section we consider the computational power of L_1(BB)-programs over different sets of primitive instructions, BB. Most of our results consider problems considered in [7-10].

Our first result shows that L_1(BB1) and L_1(BB2)-programs can be converted into L_1({x←0, x←x+1, if x=0 then y←y+1})-programs and L_1(x←0, x←x+1, x←x÷1, if x=0 then y←y+1})-programs, respectively, in polynomial time. This is an improvement over the exponential time needed in [10].

Theorem 1. Given a L_1(BB1)-program P, one can construct in polynomial time, a L_1({x←0, x←x+1, if x=0 then y←y+1})-program P' such that P' is equivalent to P.

Proof. Let P be a L_1(BB1)-program. Using techniques in [3,6] one can construct a straight-line program Q, in polynomial time, over the instructions:

```
x←0
x←x+1
x←x+y
x←x÷1
y←x/k
y←remainder(x,k)
x←(1÷y)x,
```

where k represents a positive integer constant expressed in unary, such that Q is equivalent to P. The result now follows since each of the Q instructions can be simulated by L_1({x←0, x←x+1, if x=0 then y←y+1})-programs. Most of the encodings are straightforward. To illustrate the idea we provide the encoding for the y←x/k instruction. The remaining encodings are similar and are left to the reader.

Let $v_1,...,v_k$, u, w, r and s be new variables. Suppose that v_i ($1 \le i \le k$) are 0/1 valued. Now v can be considered to be a 0/1 valued vector of length k. The function SHIFT(v,j), $1 \le |j| \le k$ is defined to be a circular shift of the values in v by j places. For example, let k=3, v_1=1, v_2=1 and v_3=0. Then SHIFT(v,-1) sets v_1=1, v_2=0 and v_3=1. Now if r and s are 0/1 valued variables, then r←s is simulated by:

```
w←0
if s==0 then w←w+1
r←0
if w=0 then r←r+1
```

Now SHIFT(v,-1) can be computed in the usual fashion. Now then y←x/k can be computed by the following segment of code:

```
v1←1;...;vk-1←1; vk←0; y←0
do x
   if v1=0 then y←y+1
   SHIFT(v,-1)
end
```

■

The proof for L_1(BB2)-programs is similar. One merely allows the intermediate program Q the use of the additional instruction x←x÷y. The rest of the theorem is the same.

Our next result considers whether L_1(BB1 ∪ {if})-programs can be converted into equivalent L_1(BB1)-programs. In [10] it was claimed without proof that this could be done in polynomial time.

However in [7], it was shown that this was only possible if PSPACE=PTIME. Thus the question of convertibility seems to be in doubt. Here we provide an exponential algorithm. This result should be contrasted with the corresponding case for the set BB2, where the addition of the "if" construct provided an increase in the computational power of the language.

Theorem 2. Let P be an L_1(BB1 $\cup$ {if})-program. Then an equivalent L_1(BB1)-program P' can be constructed in exponential time (and space).

Proof. Without loss of generality we can consider P to be the program "do t A end", where only instructions of the form "if x=0 then y←z" and "x←x+1" appear in A and the variable t is not referred to in A. Let $v_1,...,v_n$ be the only variables referred to in A and let $A=I_1;...;I_m$, where each I_j $(1\leq j\leq m)$ is an instruction. Let d_i, p_i $(1\leq i\leq n)$ and u be new variables.

First we construct a new segment of code $A'=I'_1;...;I'_m$, where each I'_j depends on the form of I_j.

Case 1. I_j is "$v_i \leftarrow v_i+1$" then I'_j is:

$d_i \leftarrow 1$;
$v_i \leftarrow v_i + 1$;

Case 2. I_j is "if $v_i=0$ then $v_r \leftarrow v_s$" then I'_j is:

$u \leftarrow 1$;
if $d_i=0$ then $u \leftarrow 0$;
if $u=0$ then $v_r \leftarrow v_s$;
if $u=0$ then $d_r \leftarrow d_s$;
if $u=0$ then $p_r \leftarrow p_s$;

If initially $d_i = \begin{cases} 0 \text{ if } v_i = 0 \\ 1 \text{ otherwise} \end{cases}$ and $p_i=i$ $(1\leq i\leq n)$ the execution of A' is equivalent to the execution of A with respect to the outcome of the values of variables $v_1,...,v_n$. The value of d_i merely keeps track of whether v_i is currently zero or positive $(1\leq i\leq n)$. The p_i's keep track of the exchanges made among the variables. The value of p_i is j whenever the value of v_i is derived from the original value of v_j.

Consider the form of A'. If there are k "if" statements in A', then there are 2^k computational paths in A', since each conditional statement can either be executed or not depending on the truth of the condition. We first note that the computational path taken upon the execution of A' is entirely dependent on the initial values of the d_i's and p_i's $(1\leq i\leq n)$. This is the case since variables may only be exchanged and increased. The p_i's keep track of the exchanges while the d_i's keep track of whether a variable is zero or positive. (Note there is no way for a variable to decrease other then through an exchange.)

Let the settings of the d_i's and p_i's $(1\leq i\leq n)$ be called the states of execution. The state at any time then is $<d_1,...,d_n, p_1,...,p_n>$. (Note that there are $2^n * n^n$ or $0(2^{n \log n+n}$ states of execution.) For any initial setting of the d_i's where each $p_i=i$, $1\leq i\leq n$, we can execute A' a number of times in succession until a state is repeated. This must occur before $2^n * n^n$ executions of A'. Let q denote the state which gets repeated. Let l_1 be the number of times A' was executed before q appeared for the first time, and let l_2 be the number of times A' was executed after that until q reappeared. (Hence A' was executed a total of $l_1 + l_2$ times.)

Consider the execution of $(A')^{l_2}(=\overbrace{A';A';...;A'}^{l_2 \text{ times}})$ beginning in state q and ending in state q. The computational path taken upon execution is totally determined by q. Hence we can construct a segment of code B, which contains only instructions of the form "x←y" and "x←x+1", which is equivalent to $(A')^{l_2}$

when executing on any initial values of $v_1,...v_n$ beginning in state q. The length of B is less than or equal to the length of $(A')^{l}_{2}$.

Define d-state $(<d_1,...,d_n, p_1,...,p_n>) = <d_1,...,d_n>$ and p-state $(<d_1,...,d_n, p_1,...,p_n>) = <p_1,...,p_n>$. Now for each possible initial state, q_h where p-state $(q_h) = <1,2,...,n>$ (there are 2^n such states, hence $1 \leq h \leq 2^n$) we can find the respective constants l^h_1 and l^h_2 denoting the number of executions of A' necessary until the first repeating state, say q'_h, occurs and then reoccurs again, respectively. From $(A')^{l^h_2}$ and q'_h, the code segment without "if" statements, B_h is then constructed.

The program P' can now be described. P' is basically divided into three sections which perform the initialization, the state determination and the actual simulation respectively. P' then has the form:

P': Initialization
State determination
Actual simulation

The initialization segment is of length $0(n^2)$ and is composed of n segments of the instructions:

$p_i \leftarrow i$;
$d_i \leftarrow 1$;
if $v_i = 0$ then $d_i \leftarrow 0$;

for $1 \leq i \leq n$.

The state determination segment is of length $0(n*2^n)$ and is composed of 2^n segments of the instruction:

if d-state $(<d_1,...,d_n, p_1,...,p_n>)$ = d-state (q_h) goto label h

for $1 \leq h \leq 2^n$.

The actual simulation portion is composed of the segments labeled "label 1" through "label 2^n". There is also a label "end" at the very end of this section. The form then is:

label 1: _ _ _ _ _ _ _
.
.
.
_ _ _ _ _ _ _
label 2: _ _ _ _ _ _ _
.
.
.
_ _ _ _ _ _ _
.
.
.
label 2^n: _ _ _ _ _ _ _
.
.
.
_ _ _ _ _ _ _
end:

The instructions at "label h" $(1 \leq h \leq 2^n)$ are now described. Let w_0, w_1, w_2 be new variables.

label h: $w_0 \leftarrow s\text{-}\ l_1^h$;
$w_1 \leftarrow w_0/l_2^h$;
$w_2 \leftarrow rem(w_0, l_2^h)$;
$(A')^{l_1^h}$;
do w_1
B_h
end
$(A')^{w_2}$;
goto end

The length of this segment is $0(m * 2^n * n^n)$. There are $0(2^n)$ such segments in P'. The total length of P' then is $0(2^{cl \log l})$, where l=length(P).

Two things still need to be mentioned. First it should be clear from the earlier discussion that P and P' are equivalent. Second it is still perhaps not clear that P' can be represented by a $L_1(\{x\leftarrow 0,\ x\leftarrow x+1,\ x\leftarrow y\})$-program. In order to see that it can, the reader should consult [12], to see that integer division by a constant and the remainder of an integer division by a constant can be computed by such programs. The fact that such programs can simulate forward goto statements which are outside the scope of any do-loop (and whose target is also outside the scope of any do-loop) is straightforward. ■

Next we consider the computational power of L_1(BB2)-programs when allowing the instructions "y←c" and "y←y÷1" to be conditionally executed, where c can be any nonnegative constant. In our next result we show that $L_1(BB2 \cup \{\underline{\text{if}}\ x=0\ \underline{\text{then}}\ y\leftarrow c\})$[2]-programs compute Presburger functions. For such a program we construct a nondeterministic reversal bounded multicounter machine (hereafter CM) to in some sense simulate the programs computation. If the program has m input variables and n output variables then the CM will on input $\#a_1^{i_1}\#\ldots\#a_{m+n}^{i_{m+n}}$, accept if and only if the program on input $i_1,\ldots,i_m$ outputs $i_{m+1},\ldots,i_{m+n}$. The result now follows from the results concerning nondeterministic reversal bounded CM of [5]. See [5] for precise definitions.

Theorem 3. Every $L_1(BB2 \cup \{\underline{\text{if}}\ x=0\ \underline{\text{then}}\ y\leftarrow c\})$-program computes a Presburger function.

Without loss of generality we need only consider functions computed by programs of the form "do z A end", where A is an arbitrary sequence of instructions over $BB2 \cup \{\underline{\text{if}}\ x=0\ \underline{\text{then}}\ y\leftarrow c\}$. Using techniques from [3], it can be shown that one can find a code segment B^A (over the same set of instructions) that is equivalent to A where the following is true. $B^A = B_1^A, B_2^A$, where B_1^A contains only instructions of the form "x←y" and B_2^A contains no instructions of the form "x←y". Furthermore it is the case that no variable appears on the left hand side of a statement in B_1^A more than once. The construction of B^A can be accomplished in polynomial time in a straightforward manner and its details are left to the reader.

Define the function g:{set of variable names} x {set of code segments over $BB2 \cup \{\underline{\text{if}}\ x=0\ \underline{\text{then}}\ y\leftarrow c\}$} → {set of variable names}, as follows. Let B be a (possibly null) segment of code and z a variable name:

$$g(z, B; x\leftarrow y) = \begin{cases} g(y,B) \text{ if } x=z \\ g(z,B) \text{ otherwise} \end{cases}$$

$$g(z,\lambda) = z$$

[2]Any constant can be substituted for c. In fact each instance of such a statement can have a different constant.

Create the directed graph G^A with a node for each variable mentioned in B^A, which contains the edge $u \to v$, if and only if $g(u,B^A)=v$. The reader can verify that G^A is actually a collection of connected subgraphs $G_1^A,...,G_m^A$ where each subgraph G_i^A, $1 \leq i \leq m$, has at most one cycle. The variables in G_i^A can then be partitioned into two sets, those which are contained within the cycle which we denote as $x_{i1},...,x_{in_i}$ and those not contained in the cycle which we shall denote as $y_{i1},...,y_{ik_i}$. For ease of illustration let the statements, in B_2^A of the form "if x=0 then y←c" be labeled 1,...,k. Let N be a constant greater than n_i and k_i for $1 \leq i \leq m$ (e.g. $N = 1 + \max_{1 \leq i \leq m}\{n_i, k_i\}$ will do). Let $D = 2 * N * \max\{|B_2^A|, \max\{\text{the constants used in } B_2^A\}\}$.

We will construct a CM, M, that will (in a way) simulate the program "do z B^A end". If the program has m input variables and n output variables, then M will on input $\#a_{i_1}\#...\#a_{i_{m+n}}\#$, accept if and only if the program on inputs $i_1,...,i_m$ outputs $i_{m+1},...,i_{m+n}$.

M will have the following counters;

$c\langle x,i,j\rangle$ $\quad 1 \leq i \leq m,\ 1 \leq j \leq n_i$

$c\langle y,i,j\rangle$ $\quad 1 \leq i \leq m,\ 1 \leq j \leq k_i$

$c\langle x,i,j,s,t\rangle$ $\quad 1 \leq i \leq m,\ 1 \leq j,t \leq n_i,\ 1 \leq s \leq k$

$c\langle 0\rangle$ (the zero counter which is always set to zero),

each initially set to zero. In the finite state control of M there will be the following buffers;

$b\langle x,i,j\rangle$ $\quad 1 \leq i \leq m,\ 1 \leq j \leq n_i$

$b\langle y,i,j\rangle$ $\quad 1 \leq i \leq m,\ 1 \leq j \leq k_i$

each capable of containing any binary number between zero and a constant d which we will determine later. Also there are the following pointers (or indicators);

$p\langle x,i,j\rangle$ $\quad 1 \leq i \leq m,\ 1 \leq j \leq n_i$

$p\langle y,i,j\rangle$ $\quad 1 \leq i \leq m,\ 1 \leq j \leq k_i$

each capable of having a value denoting any of the aforementioned counters or the special value λ. Thus, it is possible for the value of $p\langle x,i,j\rangle$ (for some i,j) to be $\langle x,i,j,s,t\rangle$ (for some i,j,s,t), for example.

In addition, the finite state control contains the entities;

$root\langle x,i,j\rangle$ $\quad 1 \leq i \leq m,\ 1 \leq j \leq n_i$

$B\langle x,i,j,s,t\rangle$ $\quad 1 \leq i \leq m,\ 1 \leq j,t \leq n_i,\ 1 \leq s \leq k$

Since the variables in a cycle can be switched about by instructions of the form "x←y", it is important to the simulation which variable is operating under what name (or alias). Thus $root\langle x,i,j\rangle = t$ means that x_{ij} is acting for x_{it} on this pass of the loop. Also at this time $p\langle x,i,j\rangle$ must equal $\langle x,i,t\rangle$, unless the execution of a conditional statement has changed the value of a variable acting for i_{it}, earlier in the simulation. In this case the value of $p\langle x,i,j\rangle$ will either be $\langle 0\rangle$ or $\langle x,i,h,s,t\rangle$, where in the latter case the variable acting for x_{it} at the time the aforementioned conditional statement was executed, was x_{ih}. So at that time, $root\langle x,i,h\rangle$ was t.

B<x,i,j,s,t> is a bounded counter, maintained in the finite state control, capable of containing any number between zero and N. Before M simulates the loop of the program, the contents of B<x,i,j,s,t> will be "guessed". Let the statement labeled s be "if u=0 then $x_{ij} \leftarrow c$". If the initial value of B<x,i,j,s,t> is not N, the M has guessed that the statement labeled s will cause the variable x_{ij} to be set to the value c, while root<x,i,j> = t, exactly B<x,i,j,s,t> times. If B<x,i,j,s,t> is initially N, then M has guessed the aforementioned conditional execution will happen at least N times and so M must also "guess" (at a later time) the last N times this action takes place. If we are only concerned with the final values of all the variables then only the last N times the statement labeled s causes x_{ij} to be set to c, for a certain value of root<x,i,j> is of consequence. The other occurrences are important only in determining which conditional statements are actually executed (i.e. cause a variable to be set to a constant). But we shall see that M need not remember the exact value of a variable to determine whether or not it will be zero at any given time.

We will now describe the simulation performed by M, which can be viewed in three stages. In the first stage M reads in the programs inputs and initializes M as follows:

1. p<x,i,j> and p<y,i,j> are set to <x,i,j> and <y,i,j>, respectively (for appropriate values of i and j).
2. b<x,i,j> and c<x,i,j> are set so that the input value of x_{ij} is equal to the value in b<x,i,j> plus the value in c<x,i,j>, in a way such that b<x,i,j> = min{2 * D, the input value of x_{ij}}.
3. b<y,i,j> and c<x,i,j> are initialized similarly.

Intuitively at each instant of the program's execution the current value of variable x_{ij} (y_{ij}) is at the corresponding instant in the simulation, the value contained in b<x,i,j> (b<y,i,j>) plus the value contained in the counter indicated by p<x,i,j> (p<y,i,j>). In reality this is not always the case, but it will be the case that x_{ij} can only have a value of zero at some instant, in the program's execution, if and only if at the corresponding instant of the simulation x_{ij} has the value zero. The second stage will simulate the execution of the loop structure, which is the program. If M has not rejected at the end of this stage, the current values of each variable correspond exactly to their values when the program terminates. Hence the third stage of M merely checks these values against the corresponding output values given for the program as input to M. If they match M accepts otherwise M rejects.

We now describe the simulation performed in stage two. First M copies the value represented by the loop control variable z, into a distinct counter (that is not mentioned above) to be used exclusively for the number of iterations of the loop. The value of root<x,i,j> is set to j, $1 \leq i \leq m$, $1 \leq j \leq n_i$. A value between zero and N, inclusive, is "guessed" for each B<x,i,j,s,t>, $1 \leq i \leq m$, $1 \leq s \leq k$, $1 \leq j,t \leq n_i$.

The simulation is then performed instruction by instruction as follows depending of course on the type of instruction involved.

Instruction to be Simulated	Simulation
$u \leftarrow v$	The contents of the buffer, pointer and root for the variable v are copied into the respective locations for variable u (if they exist).

$u \leftarrow u + 1$	The buffer for variable u is incremented by 1 unless the buffer contains the value d (and the pointer value is λ) in which case do nothing.
$u \leftarrow u \dot{-} 1$	The buffer for variable u is decremented.
s: if $u = 0$ then $x_{ij} \leftarrow c$	The buffer for variable u is checked for zero. If it is not zero then do nothing. If it is zero the following steps are performed. (1) b<x,i,j> is set to c (2) if b<x,i,j,x,root<x,i,j>> = 0 then reject; /* M has previously "guessed" that the conditional statement s will not alter x_{ij} again for this value of root<x,i,j>. However, since the simulation now requires this action, M has "guessed" incorrectly. Hence M must reject. */ (3) if b<x,i,j,s,root<x,i,j>> = N then do either set p<x,i,j> to λ /* Here M "guesses" that during the simulation there will be at least N more times when statement s alters x_{ij} for this value of root<x,i,j> */ or set p<x,i,j> to <x,i,j,s,root<x,i,j>>, decrement B<x,i,j,s,root<x,i,j>> by one and reset c<x,i,j,s,root<x,i,j>> to zero. /* Here M has previously "guessed" that this is the B<i,j,s,root<x,i,j>> -th to the last time statement s will alter x_{ij} for this value of root <x,i,j>. */
s: if $u=0$ then $y_{ij} \leftarrow c$	The buffer for variable u is checked for zero. If it is not zero then do nothing otherwise perform the following. (1) b<y,i,j> is set to c. (2) p<y,i,j> is set to <0>. /* No variable will depend on the exact value of this variable after the next k_i passes, hence the buffer is sufficient to hold this value. */

After M simulates each pass of the loop the following is done, for each variable x_{ij} whose root is j (i.e. root<x,i,j> = j). (This will happen to all variables in a cycle at the same time.) Consider all vari-

ables whose pointer contents are equal to those of $p\langle x,i,j\rangle$. The reader should note that the current value of any two of these variables can differ by at most D, (unless $p\langle x,i,j\rangle$ is λ). The counter pointed to by $p\langle x,i,j\rangle$ and the buffer $b\langle x,i,j\rangle$ are adjusted so that their combined value is the same but that $b\langle x,i,j\rangle = \min\{2*D$, the current value of $x_{ij}\}$. Note that this step appears to require a counter reversal. We will show later that this is not the case. For any other pointer $p\langle y,i,j\rangle$ that points to the same counter (other than λ), changes are made to $b\langle y,i,j\rangle$ reflecting the change. These values are not so important, since within the next N passes of the loop, they will be replaced. Also since they differ in value by at most D from the current value of x_{ij} this just entails updating the correct buffer.

Now we wish to make the following claims:

1. M is reversal bounded for the constant $d = 4 * D$.
2. That upon termination of the simulation of the loop (stage two), whenever M has not already rejected and $B\langle x,i,j,s,t\rangle = 0$, $1 \leq i \leq m$, $1 \leq s \leq k$, $1 \leq j,t \leq n_i$, that the final values of x_{ij} and y_{ij} can be obtained by addition of the respective buffer and the counter indicated by the respective pointer.

We first must consider what transpires during a pass of the loop. The name of a variable is misleading due to the exchanges in B_1^A. The $root\langle x,i,j\rangle$ keeps track of exactly what variable x_{ij} represents without regard to the variable being set to a constant by a conditional statement. Whenever $root\langle x,i,j\rangle = j$ the loop has been executed 0 mod n_i times. The reader can show that from one time this is true to the next the value of x_{ij} will always increase or always not increase. The only exception to this is when a conditional statement affected the value of x_{it} at a point in between at a time when $root\langle x,i,t\rangle = j$. Hence if d is large enough to hold any changes to x_{ij} for n_i passes the only counter reversals necessary will be caused by the simulation of conditional statements. But the execution of each conditional statement causes either the use of a new counter or the reversal of a fixed counter no more than N times. Hence M is N reversal bounded.

It is the case, however, that when $root\langle x,i,j\rangle = j$ that some pointer $p\langle y,r,s\rangle$ might be the same as $p\langle x,i,j\rangle$. (The reader should note that this is not possible for another x variable in the cycle, unless the value of $p\langle x,i,j\rangle$ is λ.) If this is the case the reader can show that the current values of the respective variables can differ by at most $2 * N * |B_2^A|$, which is less than D.

Now the simulation of the program is faithful except when a variable, say x_{ij}, is set to a constant by one of the conditional statements, and $p\langle x,i,j\rangle$ is set to λ. Even in this case the simulation of all variables which utilize this value will be simulated faithfully until the value exceeds 4*D. Now if such a variable belongs to a cycle (which it must), and it is not reset by another conditional statement, then the value must have increased since it was last reset. Thus the value must increase during n_i passes of the loop. The reader should be able to show that any variable (either in or out of the cycle) which utilizes this value can never again be zero (when utilizing this value), unless another conditional statement intervenes. Thus the execution of the conditional statements depending on the value of these variables will be faithfully simulated in spite of perhaps not "knowing" the exact values.

Lastly, the reader should note that the final value of a variable can depend not only on the faithful execution of the conditional statements, but that value itself may actually only depend on the execution of a conditional statement, labeled l, which either:

1. changes the value of a variable y_{ij} and the computation ends after at most k_i passes. Hence the final value of the variable in question is less than d.
   or

2. changes the value of a variable x_{ij}. However, for the final value of any variable to depend on this it must be the case that this change caused by statement I may happen at most N-1 more times for the same value of root<x,i,j>.

But these are exactly the cases where a counter is keeping track of the exact value. Hence the final values are correct, whenever the last N such times are "guessed" correctly.

Now it is the case that if M either does not "guess" the last N (or fewer) times such a conditional statement sets a variable, M rejects. Note that M will terminate the second stage (without rejecting) with only a unique set of final values for the variables, which can be reconstructed from a variables buffer and the counter indicated by the variables pointer. ■

This result should be contrasted with the result in [9], showing that L_1(BB2 ∪ {if x=0 then y←y+1}) is strictly more powerful than L_1(BB2). If both constructs, "if x=0 then y←c" and "if x=0 then y←y÷1" are concurrently considered it is easy to show that L_1(BB2 ∪ {if x=0 then y←c, if x=0 then y←y÷1}) is computationally equivalent to L_1(BB2 ∪ {if x=0 then y←y+1}), since the instruction "if x=0 then y←y+1" can be simulated by the following sequence of instructions from BB2 ∪ {if x=0 then y←c, if x=0 then y←y÷1}:

```
w ← 0
if x = 0 then w ← 1
y ← y + 1
if w = 0 then y ← y ÷ 1,
```

where w is a new variable. (The converse was shown in [9].) Unfortunately we have been unable to resolve the computational power of L_1(BB2 ∪ {if x=0 then y←y÷1}). The "if x=0 then y←y÷1" construct seems similar to the "if x=0 then y←y+1" construct, but as pointed out in [7] functions of one variable computed over BB2 ∪ {if x=0 then y←y÷1} are monotonic. Thus the proof techniques used in [9] (as well as those presented in the last theorem) do not seem to work with this language. This same problem arose in [7], where the authors were able to show that the 0-evaluation problem for this language is PSPACE-complete.

3. L_i(BB)-Programs

In this section we consider other problems that arose with the claims in [10]. In [10] the following eleven primitive instructions were allowed. In what follows, the accompanying abbreviations are used in place of the instructions.

Assignment Statement	Abbreviation
(1) x ← 0	0 (the constant zero)
(2) x ← y	id (the identity operator)
(3) x ← x + 1	succ (successor)
(4) x ← x ∸ 1	pred (predecessor)
(5) x ← $succ^y$(x)	$succ^y$ (the y^{th} successor)
(6) x ← $pred^y$(x)	$pred^y$ (the y^{th} predecessor)
(7) x ← max(y,z)	max
(8) x ← min(y,a)	min
(9) x ← y + z	+
(10) x ← y ∸ z	∸
(11) if x=0 then A else B	if

The following are theorems given in [10] along with our accompanying comments. The numbering of the theorems corresponds to that in [10] (e.g. Theorem III.4 in this paper corresponds to Theorem 4 in Section III of [10]). To make the notation less cumbersome the set brackets have been dropped in expressing the sets BB, of primitive instructions.

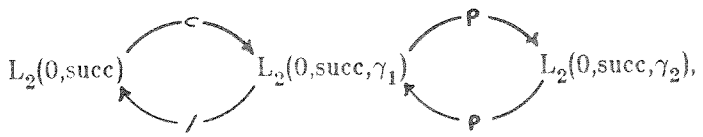
Theorem III.1. Let γ be any subset of {id,pred,$succ^y$,$pred^y$,max,min,+,∸,if}. Then for all $i \geq 3$,

where the degree of the polynomial translation depends on γ and never exceeds 2.

Comments:

This is not true when the operation of "+" is in γ. For a proof see Section 3 of [7].

Theorem III.2. Let γ_1 and γ_2 be non-empty subsets of {id,pred,$succ^y$,$pred^y$,max,min,+,∸,if}. Then

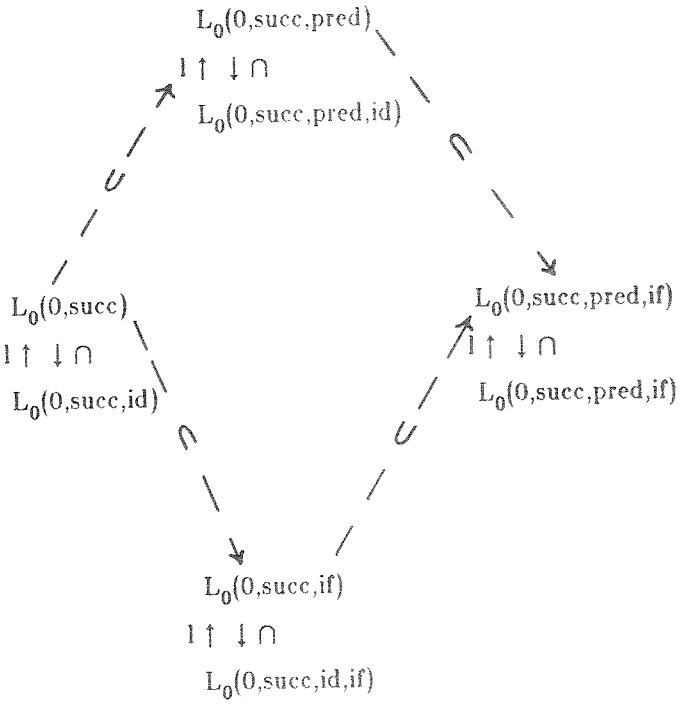


where the degrees of the polynomial translations depend on γ_1 and γ_2, but never exceed 2.

Comments:

This is not true for all possible choices of γ_1 and γ_2. For example it is not true when $\gamma_1 = \{id\}$ and $\gamma_2 = \{id,+\}$. For a proof see Section 3 of [7]. The above theorem also states that $L_2(0,succ) \not\leftrightarrow L_2(0,succ,\gamma_1)$ for any nonempty γ_1. Although the result seems likely we have been unable to prove this result and would like to see more details.

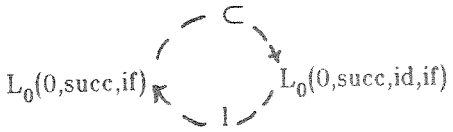
Theorem III.4.



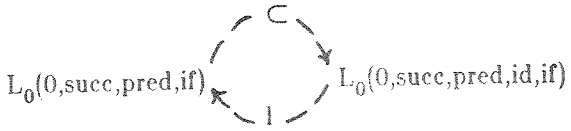
Again here, if an omitted arrow is not obtainable by composition from the arrows in the diagram, then it is a case of non-translatability.

Comments:

This theorem has two basic errors in the diagram. The following two translations are claimed.



and



The first is false since an $L_0(0,succ,if)$-program over inputs $x_1,...,x_k$ computes a function $f(x_1,...,x_k)$

$= cx_i + d$, where c is 0 or 1, $1 \leq i \leq k$ is fixed for a given program, and d belongs to a finite set of integers. It follows that such a program cannot compute the function $f(x,y,z) = \begin{cases} y & x=0 \\ z & x \neq 0 \end{cases}$. The function f can clearly be computed by an L_0(0,succ,id,if)-program, however. The second fails for a similar reason.

To finish the diagram it can be shown that L_0(0,succ,id,if)-programs and L_0(0,succ,pred,id,if)-programs are incomparable and that L_0(0,succ,pred,id,if)-programs are more powerful than L_0(0,succ,id,if)-programs.

Let $f_1,f_2,...,$ be number-theoretic functions. The class of all functions obtained through composition from $f_1,f_2,...,$ will be denoted by: $[f_1,f_2,...]$.

For the next theorem we need to define special functions 0^n, u_i^n, [./k], and w:

$o^n(x_1,...,x_n) = 0;$

$u_i^n(x_1,...,x_n) = x_i$, with $1 \leq i \leq n$;

$[x/k]$ = integer division of x by constant k;

$$w(x_1,x_2) = \begin{cases} x_1, \text{ if } x_2 = 0 \\ 0, \text{ if } x_2 \neq 0. \end{cases}$$

Theorem IV.1. We have the following algebraic characterizations:

(3) $\mathcal{L}_1$(0,succ,pred,if) = $[o^n,u_i^n, succ,w,+,\dot{-},[./k]]$.

Comments:

This was shown to be in error in [9]. If we substitute $\mathcal{L}_1$(0,succ,pred,id) for $\mathcal{L}_1$(0,succ,pred,if) then the theorem is true. In fact this was shown in [3] (see also [1,2,6]).

Theorem V.1. Let γ be any subset of $\{0,id,succ,pred,succ^Y,pred^Y,max,min,+,\dot{-},if\}$. Then for all $i \geq 2$, the equivalence problem of $L_i(0,succ,\gamma)$ is recursively unsolvable.

Comments:

This theorem says that the equivalence problem for L_2(0,succ) is unsolvable. We have been unable to show this result ourselves and would therefore like to inquire about the details.

Theorem V.2. Let P, P'ϵL_1(0,succ) and |P| and |P'| be the respective lengths of P and P'. Then whether P is equivalent to P' is solvable in time proportional to |P| + |P'|; that is, the equivalence problem of L_1(0,succ) is solvable in linear time.

Comments:

For such programs over n-inputs, the problem can easily be seen to be NP-hard (use a reduction from satisfiability).

Theorem V.5. The equivalence problem of L_1(0,succ,pred,id)-and thus, by Theorem III.3, that of L_1(0,succ,pred,if) and that of L_1(0,succ,pred,id,if) too--are each solvable in time exponential in the time required by the decision procedure for Presburger arithmetic. (Proved by reduction to Presburger arithmetic.)

Using the best known bound for a decision procedure of Presburger arithmetic, the time complexity of the procedure in Theorem V.5 is:

$$O(2^{2^{2^{c(|P|+|P'|)}}}).$$

Although this complexity makes the decision procedure impractical, it improves by one exponential level a similar result in Cherniavsky [1]. The next result shows that, if this bound can be improved further, then it cannot be improved by more than one additional exponential level.

Theorem V.6. The problem of deciding the truth of Presburger formulas is polynomially reducible to the equivalence problem of L_1(0,succ,pred,id). (Proved by using the characterization of part (3) in Theorem IV.1.)

Comments:

The discussion between Theorems 5 and 6 is also taken from [10]. Due to the errors in Theorem III.3 this theorem needs to be reconsidered. (The portion of Theorem III.3 referred to here claimed that L_1(0,succ,pred,id)-programs and L_1(0,succ,pred,id,if)-programs computed identical classes of functions.) In [9] it was shown that if a function is computable by an 0(n) space bounded TM that runs in $0(2^{\lambda n})$ time, for some $\lambda<1$, then the function is also computable by an L_1(pred,if x=0 then y←y+1)-program. It follows that the equivalence problem for L_1(0,succ,pred,if) (and L_1(0,succ,pred,id,if)) is undecidable. Also note that L_1(0,succ,pred,if) is equivalent to the language UL^- in [3]. Then Theorem 14 of [3] is exactly this result. The discussion following Theorem 5 also needs attention. The bound mentioned can be improved by 4 levels of exponentiation (see [3]), and thus Theorem 6 is in error.

Other incorrect claims appear in [10], but they precipitate from those already mentioned. A full accounting can be found in [8].

References

[1] Cherniavsky, J., Simple Programs Realize Exactly Presburger Formulas, *Siam J. Comput.*, 5 (1976), pp. 666-677.

[2] Cherniavsky, J. and Kamin, S., A Complete and Consistent Hoare Axiomatics for a Simple Programming Language, *J. ACM*, 26 (1979), pp. 119-128.

[3] Gurari, E. and Ibarra, O., The Complexity of the Equivalence Problem for Simple Programs, *JACM*, Vol. 28, No. 3, July 1981, pp. 535-560.

[4] Gurari, E. and Ibarra, O., The Complexity of the Equivalence Problem for Two Characterizations of Presburger Sets, *Theor. Computer Science*, 13 (1981) pp. 295-314.

[5] Ibarra, O., Reversal-Bounded Multicounter Machines and their Decision Problems, *J. ACM*, Vol. 25, No. 1, January 1978, pp. 116-133.

[6] Ibarra, O., and Leininger, B., Characteristics of Presburger Functions, *SIAM J. Comput.*, Vol. 10, No. 1, February 1981, pp. 22-39.

[7] Ibarra, O., Leininger, B. and Rosier, L., The Complexity of Evaluating Simple Programs Over Different Sets of Primitives, University of Minnesota, Department of Computer Science, Tech. Rep. No. 82-10 (1982).

[8] Ibarra, O. and Rosier, L., Some Comments Concerning the Analysis of Simple Programs over Different Sets of Primitives, University of Minnesota, Department of Computer Science, Tech. Rep. No. 82-10 (1982).

[9] Ibarra, O. and Rosier, L., Simple Programming Languages and Restricted Classes of Turing Machines, to appear in *Theor. Computer Science.*

[10] Kfoury, A., Analysis of Simple Programs Over Different Sets of Primitives, *7th ACM SIGACT-SIGPLAN Conference Record*, 1980, pp. 56-61.

[11] Meyer, A. and Richie, D., The Complexity of Loop Programs, in *Proc. 22nd Nat. Conf. of the ACM*, Thompson Book Co., Washington DC, 1967, pp. 465-469.

[12] Tsichritzis, D., The Equivalence of Simple Programs, *J. ACM* 17, 4 (Oct. 1970), pp. 729-738.