THE EQUIVALENCE PROBLEM AND CORRECTNESS FORMULAS FOR A SIMPLE CLASS OF PROGRAMS

Oscar H. Ibarra[1] and Louis E. Rosier[2]

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

TR-83-23 December 1983

[1]Department of Computer Science, University of Minnesota, Minneapolis, MN 55455.
This research was supported in part by NSF Grant MCS 83-04756

[2]This research was supported in part by 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. Recently, a complete and consistent Hoare axiomatics for the class of {x←0, x←y, x←x+1, x←x$\dot{-}$1, *do* x...*end*} programs which contain no nested loops, was given, where the allowable assertions were those formulas in the logic of Presburger arithmetic. The class of functions computable by such programs is exactly the class of Presburger functions. Thus, the resulting class of correctness formulas has a decidable validity problem. In this paper, we present simple loop programming languages which are, computationally, strictly more powerful, i.e. which can compute more than the class of Presburger functions. Furthermore, using a logical assertion language that is also more powerful than the logic of Presburger arithmetic, we present a class of correctness formulas over such programs that also has a decidable validity problem. In related work, we examine the expressive power of loop programs over different sets of primitive instructions. In particular, we show that an {x←0, x←y, x←x+1, *do* x ... *end*, *if* x=0 *then* y←z}-program which contains no nested loops can be transformed into an equivalent {x←0, x←y, x←x+1, *do* x ... *end*}-program (also without nested loops) in exponential time and space. This translation was earlier claimed, in the literature, to be obtainable in polynomial time, but then this was subsequently shown to imply that PSPACE=PTIME. Consequently, the question of translatability was left unanswered. Also, we show that the class of functions computable by {x←0, x←y, x←x+1, x←x$\dot{-}$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

In the previous ten or so years there has been a tremendous interest in the topic of program correctness--both from a theoretical and practical point of view[3,10,17,23]. Work in the theory of program schemata has dealt with the concept of identifying classes of programs for which various assertions about programs are mechanically verifiable. What actually needs to be verified is a correctness formula. A correctness formula is a logical formula of the form {p}S{q}, where S is a program and p and q are logical assertions about the variables in the program S. The interpretation is that, if p is true before the execution of S, then q will be true following the execution of S, assuming S terminates. Now once results are established for a particular class of programs (a program schemata) the procedures developed can be applied to any instance in the class. Much of the literature has concentrated on classes of simple programming languages but with using general assertions from a particular logic. Unfortunately, this work has produced mostly negative results. That is, for many classes of programs (and simple assertions) these questions are computationally unsolvable.

A positive result in this area is the result of Cherniavsky and Kamin[3]. They present a simple programming language and an assertion language for which they were able to provide a complete and consistent axiom system. An axiom system includes the programming language, the assertion language and a set of axioms or proof rules from which one can derive (or prove) certain logical assertions about the programs. An axiom system is said to be consistent and complete if the true correctness formulas coincide exactly with the provable ones. (We ignore here any discussion of a model for the assertion language as we expect it to be implicitly defined within each system.)

The programming language of Cherniavsky and Kamin [3] is the loop language L_1(x←0, x←y, x←x+1, x←x$\dot{-}$1). A program P in this language has the form:

$$\begin{array}{l} P : \text{input}(x_1,\ldots,x_k) \\ \quad\ A \\ \quad \text{output}(y_1,\ldots,y_t) \end{array}$$

where A is a block of instructions from the set {x←0, x←y, x←x+1, x←x$\dot{-}$1, *do* x...*end*}, k and t are constants and no nesting of loop structures is permitted in A. In general, a L_i(BB)-program will be defined similarly. In this case, however, the instructions in the block A must be taken from those in BB ∪ {*do* x...*end*} and the maximum level of nesting allowed for loop structures is i. Note then that L_i(x←0, x←y, x←x+1) are the loop languages in the subrecursive hierarchy of Meyer and Richie[21]. In particular, L_1(x←0, x←y, x←x+1) is the language shown to compute exactly the "simple" functions[24]. The assertion language used in the system of Cherniavsky and Kamin[3] is composed of those formulas in the logic of Presburger arithmetic. The computational power of L_1(x←0, x←y, x←x+1, x←x$\dot{-}$1) is quite limited. In fact programs over this language were shown to be capable of computing exactly those functions which are Presburger[2,3,7]. Actually the decidability of correctness formulas is reduced to the problem of deciding the validity of a formula in Presburger arithmetic. The equivalence problem for L_1(x←0, x←y, x←x+1, x←x$\dot{-}$1)-programs was also reduced to the same problem[2,3,7].

In any class of programming languages where the equivalence problem is not decidable, finding interesting correctness formulas whose validity is decidable is clearly not possible. In fact, for almost any class of programs where the equivalence problem is undecidable, the validity of correctness formulas of the form, {true}S{x=y}, cannot be mechanically verified. The equivalence problem for a class of programs is, given two programs in the class, to decide if these programs produce the same outputs when they are

given identical inputs. Much is known about classes of simple programming languages and the corresponding classes of functions which they compute, as well as the difficulty of the respective equivalence problems. As mentioned earlier, Meyer and Richie[21] exhibited the hierarchy of simple programming languages L_i(x←0, x←y, x←x+1) whose union is a class of programs, which is capable of computing exactly the class of primitive recursion functions. The programming language classes, L_1(x←0, x←y, x←x+1) and L_2(x←0, x←y, x←x+1), constitute the lower two levels of this hierarchy. Programs in these classes compute the "simple" functions of Tsichritzis[24] and the elementary recursive functions, respectively. The programming language used by Cherniavsky and Kamin[2,3] is a slight generalization of the L_1(x←0, x←y, x←x+1) language. The class of L_1(x←0, x←y, x←x+1) (L_2(x←0, x←y, x←x+1)) programs has a decidable (undecidable) equivalence problem. Hence possible languages of interest, in terms of computational power, would include those that reside somewhere between L_1(x←0, x←y, x←x+1, x←x$\dot{-}$1) and L_2(x←0, x←y, x←x+1). Many examples of programming languages are known whose computational power is equivalent to that of L_1(x←0, x←y, x←x+1, x←x$\dot{-}$1)[2,3,7,12]. Until recently, however, few examples of programming languages, whose computational power lies properly within this range, appeared in the literature. Recent work by the authors has contributed in this area[16]. A possible approach for further research in this area then would be to examine various simple languages over different instruction sets. Programming languages of interest would have computational power greater than the language of Cherniavsky and Kamin, and yet still have interesting classes of correctness formulas that are mechanically verifiable. Additional work would be required to find suitable assertions. Another approach would be to use a simple assertion language and allow nontrivial, but limited, predicates. In[23], the predicate PERM(M,N), (indicating array M is a permutation of array N), was added to the assertion language of a simple system, and the resulting system still had interesting decidable correctness formulas.

In this paper we introduce simple loop programming languages S and T which are computationally more powerful than L_1(x←0, x←y, x←x+1, x←x$\dot{-}$1). Subsequently, we show that the classes of S and T programs have a decidable equivalence problem. We also show a decidable class of correctness formulas for such programs.

We begin by looking at the language L_1(x←0, x←y, x←x+1, x←x$\dot{-}$1) and see what constructs we can add and still have a decidable equivalence problem. L_1(x←0, x←y, x←x+1, x←x$\dot{-}$1) computes exactly the Presburger functions[2] and those are precisely the functions which are computable by straight-line programs over the instruction set:

(1) x←1
(2) x←x+y
(3) x←x$\dot{-}$y
(4) x←x/k, where / denotes integer division with truncation and
k is a positive integer constant.
(5) x←x *mod* k, where for positive integers x and y, x *mod* y
is defined to be x-y⌊x/y⌋ if y $\neq$ 0 and x
otherwise[19]. Again k is a positive integer
constant.
(6) *if* x=0 *then* I

where I denotes any instruction of type 1-5[12]. In fact, an L_1(x←0, x←y, x←x+1, x←x$\dot{-}$1) program can be converted into an equivalent straight-line program over these instructions in polynomial time[7]. Thus addition and proper subtraction are allowed but multiplication, division, and the modulo operation are only allowed by positive integer constants (i.e., x←k*x, x←x/k, x←x *mod* k). Now what we want is to extend the language L_1(x←0, x←y, x←x+1, x←x$\dot{-}$1) so that the resulting language becomes equivalent to a com-

putationally more powerful class of straight-line programs, but which still has a decidable equivalence problem. Clearly, a more powerful class of straight-line programs would result if we allowed any one of the following constructs: x←x/y, x←x*y, x←x *mod* y. Unfortunately, it is known (see [13]) that programs over {x←1, x←x+y, x←x/y} or over {x←1, x←x+y, x←x$\dot{-}$y, x←x*y} have an undecidable zero-equivalence problem. (The zero-equivalence problem for a class of programs is deciding for a member of that class whether the program outputs a zero for all possible inputs.) The only other case worth considering then is the addition of the construct x←x *mod* y, and for this we can show that equivalence is decidable.

Throughout, U will denote the class of programs over the following instruction set:

(1) x←1
(2) x←x+y
(3) x←x$\dot{-}$y
(4) x←x/k
(5) x←x *mod* y
(6) *if* x=0 *then* I

where I denotes any instruction of type 1-5.

We now define the classes of loop programs S and T which are (polynomially) equivalent to U. For ease of explanation we describe the programming languages S and T over the following seven instruction types:

(1) x←0
(2) x←x+1
(3) x←x$\dot{-}$y
(4) x←y
(5) *for* ι=u *to* v *by* c *do*
A
end
(6) x←y *mod* z
(7) *if* x|y *then* z←0, where '|' means 'divides'

where A is a block of primitive instructions (1-4, 6-7), ι is called the loop control variable and c is a constant. The interpretation is that ι is assigned the value of u on the 1st pass, u+c on the 2nd pass,..., and v-(v-u) *mod* c on the last pass, where changes to variables u, v and ι inside the loop do not affect the number of loop executions or the assignment made to variable ι preceding each pass of the loop. (If v<u, then the loop is not executed.)

The language S allows only the primitive instructions 1-4 and 6, and the language T allows only primitive constructs 1-4 and 7. Restrictions are also placed on loop structures which are allowable in S and T programs. The restrictions are syntactic restrictions placed on the blocks of instructions A that are allowed inside *for* loops. First, however, we need the following definitions.

Consider a block of instructions A=I_1;...;I_l, where each I_j ($1 \leq j \leq l$) is an instruction of type 1-4 or 6-7. Then the functions p_A and b_A are defined to be for $1 \leq j \leq l$,

$$p_A(j,x) = \begin{cases} y & \text{if } I_j \text{ is "x←y" or "x←x } mod \text{ y"} \\ x & \text{otherwise} \end{cases}$$

$b_A(j,x) = p_A(1,...p_A(j\text{-}1,p(j,x))...)$, $1 \leq j \leq l$.

Thus, $p_A(j,x)$ is the name of the variable before the execution of I_j, which is x after the execution of I_j (i.e. the value of x, after the execution of I_j, is derived from the value of $p_A(j,x)$ before the execution of I_j), and $b_A(j,x)$ is the name of the variable, before I_1 is executed, from which the value of x is derived, after the execution of $I_1;...;I_j$.

Note that the functions p_A and b_A are defined with respect to a block of instructions, A. Consider a loop structure of the form "*for* ι=u *to* v *by* c *do* A; *end*;". Then this is an allowable loop structure for S (or T)-programs if A contains only the instructions allowable in S (or T)-programs and restrictions 1 and 2 hold for A.

Restriction 1. If I_j is the instruction "*if* x|y *then* z←0" or "x←y *mod* z", then $b_A(j,y)=\iota$.

Restriction 2. If I_j is the instruction "*if* x|y *then* z←0" or "z←y mod x" then x may not be altered (appear on the left hand side of an assignment statement) within A.

Note that for any block of instructions of types 1-4 and 6-7 restrictions 1 and 2 can be syntactically checked. A S (T)-program is a program over instruction types 1-6 (1-5,7) that allows no nesting of loop structures and where restrictions 1 and 2 hold for each block of instructions, A, which is enclosed within a loop structure.

Clearly S (T)-programs are more powerful than L_1(x←0, x←y, x←x+1, x←x$\dot{-}$1)-programs since an S (T)-program can compute x *mod* y. We shall show that the class of S (T)-programs has a decidable equivalence problem. In each case, we illustrate a polynomial time procedure that converts an S (T)-program into an equivalent U-program.

One might question whether the restrictions are necessary for S and T-programs. While we are not able to provide proofs in either case we provide evidence that indicates, probably so. For example, if restrictions 1 and 2 were not imposed on S-programs then such programs would be capable of computing the function *gcd*(x,y). First, our proof that the equivalence problem is decidable seems to fail if such functions as *gcd*(x,y) are allowed. Secondly, suppose that the introduction of the *gcd* function (i.e. the instruction z←*gcd*(x,y)) adds no computational power to the class of U-programs. Then the *gcd* function would not be harder to compute than multiplication. This would answer an open question in [1] in a very surprising way. In the case of T-programs we show that the removal of restriction 1 or 2 implies that the resulting class of programs has a PSPACE-complete 0-evaluation problem. (The 0-evaluation problem for a class of programs C is given a C-program P with one output variable, does P, when all input variables are initially zero, output a zero?) See [14]. Clearly the class of T-programs has a polynomial time 0-evaluation problem by virtue of the polynomial time translation of a T-program to a U-program.

Another point of comparison for S and T-programs is the class of DL-programs introduced in [6]. In [6] it was shown that the class of functions computable by DL-programs properly include the class of Presburger functions and that the class of DL-programs has a decidable equivalence problem. The essential difference between DL-programs and classes of programs that compute Presburger functions seems to be solely that a DL-program can perform an unbounded number of I/O operations. (In fact it was shown in

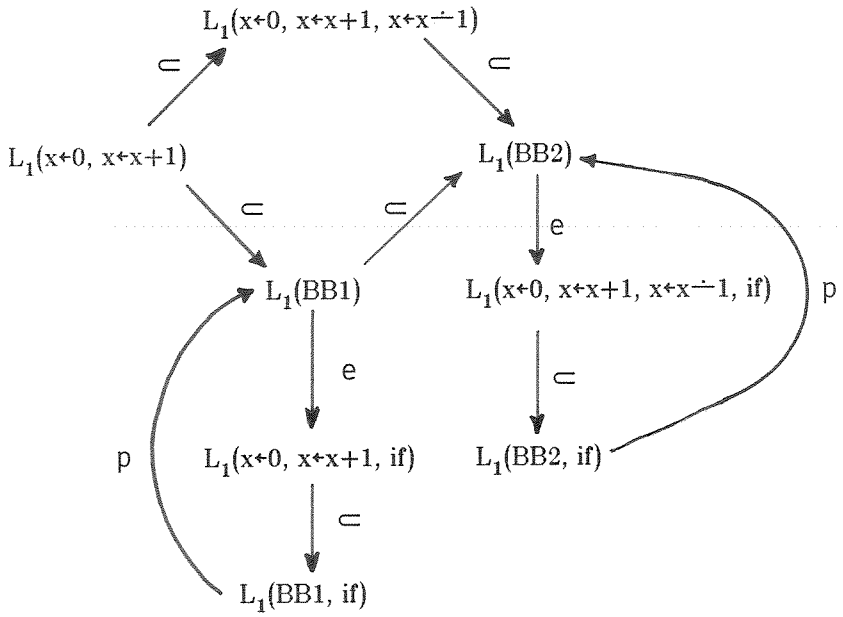
[6], that any DL-program with a bounded number of inputs computed a Presburger function.) Thus, it is clear that the class of functions computable by DL-programs is not comparable with the class of functions computable by S, T or U-programs, although both properly include the class of Presburger functions.

In Section 3, we show that the class of U-programs has a decidable equivalence problem. We then generalize this by looking at a class of unquantified correctness formulas. We show that this class of correctness formulas is decidable. Lastly we mention how this work can be used to extend the class of decidable correctness formulas in [17,23].

In the remaining sections of this paper, we investigate claims made (without proof) in [18], concerning L_1(BB)-programs where BB $\subseteq$ {x←0, x←y, x←x+1, x←x$\dot{-}$1, *if* x=0 *then* A *else* B}, and where A and B are (finite blocks) of the other primitive instructions in the set BB. We paraphrase the following definitions from [18]. Let L and L' be classes of programs, and $\mathcal{C}$ and $\mathcal{C}'$ 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 "$\subset$", "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 [18], 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 BB1={x←0, x←y, x←x+1}. Let BB2 denote the set BB1 $\cup$ {x←x$\dot{-}$1}. The following theorem was claimed without proof in [18]. To make the notation less cumbersome the set brackets have been dropped in expressing the sets BB, of primitive instructions.

Theorem. Let γ_1 and γ_2 be subsets of {x←y, x←x$\dot{-}$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--> $L_1(x\leftarrow 0,\ x\leftarrow x+1,\ if)$ --⊂--> L_1(BB1, if) --p--> L_1(BB1)

and definitely contained an error concerning:

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

The remaining claims of the theorem are correct. From results in [14], it can be noted that L_1(BB1, if) --p--> L_1(BB1) implies PSPACE=PTIME. In [16], it was noted that $L_1(x\leftarrow 0,\ x\leftarrow x+1,\ x\leftarrow x\dot{-}1,\ if)$-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, if) and L_1(BB1).

In Section 4, we consider this problem as well as examine the computational gap between L_1(BB2)-programs and L_1(BB2, if)-programs. We concentrate on allowing the instructions "y←c" and

"y←y$\dot{-}$1" to be conditionally executed. That is, we introduce the constructs "*if* x=0 *then* y←y+1", "*if* x=0 *then* y←y$\dot{-}$1" and "*if* x=0 *then* y←c", where c is any nonnegative integer constant. We are then able to show that:

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

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:

$$L_1(BB2, \textit{if } x=0 \textit{ then } y\leftarrow c) \text{ --e--> } L_1(BB2).$$

This should be contrasted with the result in [16], showing that L_1(BB2, *if* x=0 *then* y←y+1) is strictly more powerful than L_1(BB2). L_1(BB2, *if* x=0 *then* y←y+1)-programs were shown to be computationally equivalent to L_1(x←0, x←x+1, x←x$\dot{-}$1, if)-programs, in [16]. If both constructs, "*if* x=0 *then* y←c" and "*if* x=0 *then* y←y$\dot{-}$1" are concurrently considered it is easy to show that L_1(BB2, *if* x=0 *then* y←1, *if* x=0 *then* y←y$\dot{-}$1) is computationally equivalent to L_1(BB2, *if* x=0 *then* y←y+1). Unfortunately, we are unable to resolve the computational power of L_1(BB2, *if* x=0 *then* y←y$\dot{-}$1)-programs. The "*if* x=0 *then* y←y$\dot{-}$1" construct seems similar to the "*if* x=0 *then* y←y+1" construct, but as pointed out in [14], functions of one variable computed over BB2 $\cup$ {*if* x=0 *then* y←y$\dot{-}$1} are monotonic. Thus the proof techniques used in [16] do not seem to work with this language. This same problem was apparent in [14], 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, *if* x=0 *then* y←z)-programs are computationally equivalent to L_1(BB2, if)-programs.) If L_1(BB2, *if* x=0 *then* y←z)-programs are computationally more powerful than L_1(BB2, *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 [16].) Other corrections to errors in [18] are presented in the last section.

2. THE RELATIONSHIP BETWEEN S, T AND U-PROGRAMS

In this section, we show that there is a polynomial time procedure which translates an S or T-program into an equivalent U-program. Thus when in the next section we show that the class of U-programs has a decidable equivalence problem, the same can be said for the class of S and T-programs, respectively. The time required for the decision procedure is $O(2^{p(n)})$ for U-programs, and thus also for S and T-programs, where p is a polynomial. The last part of this section is concerned with showing that if we relaxed the class of T-programs by removing either restriction 1 or 2, then the resulting class of programs would have a PSPACE-complete 0-evaluation problem.

We start with the following theorem.

Theorem 1. Given an S-program P, we can construct a U-program P', in time polynomial in the length of P, such that P' is equivalent to P.

Proof. Without loss of generality we need only consider how to convert a loop structure into an equivalent sequence of U-instructions. Consider then the loop structure "*for* ι=u *to* v *by* c *do*; A; *end*;".

We consider how to compute the final value for each variable mentioned in the loop structure. Recall the functions defined earlier, p_A and b_A. We now define new functions s_A, a_A and q_A. Let A = $I_1;...;I_l$. Then for $1 \leq j \leq l$

$$q_A(j,x) = p_A(j+1,...p_A(l\text{-}1, p_A(l,x))...)$$

$$s_A(j,x) = q_A(j,b_A(j,x))$$

$$a_A(i,j,x) = s_A^i(j,x),\ i \geq 0$$

Thus $q_A(j,x)$ is the name of the variable after the execution of I_j from which the variable x is derived after the execution of $I_{j+1};...;I_l$. $s_A(j,x)$ is the name of the variable from which x is derived after the execution of $I_{j+1};...;I_l;I_1;...;I_j$ or after one additional pass of the loop. a(i,j,x) is the name of the variable (after the execution of I_j) from which x is derived after i passes of the loop.

Let $w_1,...,w_n$ be the variables mentioned in A. Let A' be the code segment A with all instructions of the form "x←y *mod* z" removed. For a variable $x \epsilon \{w_1,...,w_n\}$ find the *least* nonnegative integer, g(x), such that there exists a j where I_j is the instruction "$a_A(g(x),j,q_A(j,x)) \leftarrow$ y *mod* z". Note that $a_A(g(x),j,q_A(j,x))$ is the name of the variable, g(x) passes previous to the termination of the loop, from which the value of x after the termination of the loop, will be derived. If g(x) exists, choose j to be as large as possible $1 \leq j \leq l$. The reader can verify that if g(x) exists then $g(x) \leq n$. Furthermore, if q were the value of "$a_A(g(x),j,q_A(j,g(x)))$" after the execution of I_j, g(x) passes before the termination of the loop, then there is a function f_x computable by a U-program which can be found in $O((n+l)^2)$ time such that given the value of q, can compute the final value of x. (Note that if g(x) does not exist, then the final value of x will be the same if we execute the loop structure "*for* ι=u *to* v *by* c *do*; A'; *end*;" instead.)

Now the value of q is the value of ι (g(x) passes before the termination of the loop) plus or minus (proper subtraction) some nonnegative constant modulo z (which is not altered in the loop). Let q then, as

a function of ι and z, be calculated by $h_x(\iota,z)$. Clearly, h_x is U-computable using a fixed number of instructions. The value of ι, g(x) passes before the termination of the loop is $u+c(r\dot{-}g(x))$ where r is the total number of times the loop will be executed. (Provided of course that $r>g(x)$). But $r=((v+c)\dot{-}u)/c$. Thus the final value of x will be:

$$f_x(h_x(u+c(r\dot{-}g(x)),z)),$$

providing the loop is executed more than g(x) times. Let this function be $F_x(u,r,c,z)$. Also for the variable x there is a unique variable z involved. Let us denote this variable by z_x.

We now construct the program to simulate "*for* ι=u *to* v *by* c; A; *end*;". Let $w_1,\ldots,w_m$ $(m\leq n)$ be the variables mentioned in A for which g is defined. Let w_1' $(1\leq i\leq n)$, w_i'' $(1\leq i\leq m)$, r, s, and t be new variables not mentioned in A. An equivalent program would be:

```
r ← ((v+c)-.-u)/c                          /* r = the number of times the loop will be executed */

s ← max {g(w_i)}                           /* s really is a constant depending only on A */
      1≤i≤m

w_1" ← F_w1(u,r,c,z_w1)                    /* w_i" (1≤i≤m) is the final value of w_i if r > s (i.e. t≠0) */
   .
   .
   .
w_m" ← F_wm(u,r,c,z_wm)

w_1' ← w_1
   .
   .
   .
w_n' ← w_n

t ← r -.- s

ι ← u

repeat s times [
   A
   if r=0 then w_1 ← w_1'                  /* restore old value of w_i (1≤i≤n) if r=0 indicating
                                              loop simulation is over */
      .
      .
      .
   if r=0 then w_n ← w_1'
   if r≠0 then w_1' ← w_1                  /* save current value of w_i (1≤i≤n) */
      .
      .
      .
   if r≠0 then w_n' ← w_n
   r ← r -.- 1; ι ← ι+c
]

for ι=u+s*c to v by c do;
   A';
end;

if t ≠ 0 then w_1 ← w_1"
   .
   .
   .
if t ≠ 0 then w_m ← w_m"
```

The idea is to precalculate the final values of all w_i $(1 \leq i \leq m)$ under the assumption that the loop will be executed more than s times (these are the values in w_i'' $(1 \leq i \leq m)$). If this was indeed the case then the final values of w_i $(1 \leq i \leq m)$ are set in the last m statements. If the loop is not executed s times, w_i $(1 \leq i \leq m)$ will already have the correct value. Note that the remaining variables w_i $(m+1 \leq i \leq n)$ are also given the proper values. Now since A' contains no instructions of the form "x ← y *mod* z" the remaining

loop structure can be converted to a sequence of U-instructions using results in [7]. The time complexity for this is $O((\text{length}(A'))^{12})$. The remaining statements can also be replaced by sequences of U-instructions in a straightforward manner. The total complexity is then $O((\text{length}(A))^{12})$. □

One might question whether the restrictions imposed are necessary for S-programs. We now indicate why this is probably the case, although we are unable to prove it. If restrictions 1 and 2 were not imposed on S-programs, then such programs would be capable of computing the function *gcd*(x,y). First of all it does not seem likely that U-programs can compute *gcd*(x,y) and hence our proof for equivalence seems to fail. Secondly if we added the instruction "z←*gcd*(x,y)" to U and this did not add computational power to the language, then the *gcd* function would not be harder to compute than multiplication. This would answer an open question in [1] in a very surprising way.

We now consider the case of T-programs.

Theorem 2. Given a T-program P, we can construct a U-program P', in time polynomial in the length of P, such that P' is equivalent to P.

Proof. Without loss of generality we need only consider how to convert a loop structure into an equivalent sequence of U-instructions. Consider the loop structure, "*for* ι=u *to* v *by* c *do*; A; *end*;".

Case 1. Suppose A contains no instructions of the form "*if* x|y *then* z←0", then one can use the techniques developed in [7] to convert A to an equivalent sequence of U-instructions. The time required is then $O((\text{length}(P))^{12})$. Note that the instruction "x←y *mod* z" is not needed for this conversion.

Case 2. Suppose A contains an instruction of the form "*if* x|y *then* z←0". Recall the functions defined earlier, p_A, b_A, q_A, s_A and a_A. Again we assume $A=I_1;...;I_l$. Let the "*if* x|y *then* z← 0" instruction be I_j. Then A is:

A_1;
if x|y *then* z←0;
A_2;

where $A_1=I_1;...;I_{j-1}$ and $A_2=I_{j+1};...;I_l$.

Since $b_A(j,y)=\iota$, we have $y=\iota+k$ or $\iota-k$ for some nonnegative integer k whenever I_j is executed. In what follows we wish to assume that y will always contain a positive integer. Let r_i $(1\leq i\leq n)$ be the variables mentioned in A. Let r_i' $(1\leq i\leq n)$, u_0, v_0 be new variables. Let h be the smallest positive integer such that $h*c>k$. Then the code segment;

for ι=u *to* v *by* c *do*;
    A;
end;

can be replaced by;

```
u0 ← u; v0 ← v;
for ι=u0 to u0+(h-1)c by c do;
    A;
end;
for ι=u0+h*c to v0 by c do;
    A;
end;
```

The latter loop structure satisfies the requirement but the former loop structure must be replaced by the code segment.

h segments

$r_i' \leftarrow r_i \quad (1 \le i \le n)$
$\iota = u_0$
A
if $u_0 > v_0$ then $r_i \leftarrow r_i' \quad (1 \le i \le n)$

$r_i' \leftarrow r_i \quad (1 \le i \le n)$
$\iota = u_0 + c$
A
if $u_0 + c > v_0$ then $r_i \leftarrow r_i' \quad (1 \le i \le n)$

.
.
.

$r_i' \leftarrow r_i \quad (1 \le i \le n)$
$\iota = u_0 + (h-1)\,c$
A
if $u_0 + (h-1)c > v_0$ then $r_i \leftarrow r_i' \quad (1 \le i \le n)$

In what follows then we may assume whenever I_j is executed that $y > 0$.

Let $r_1,\ldots,r_n$ be the variables mentioned in A. Let $w_1,\ldots,w_n$ be new variables. Let u_0 and v_0 be the values of variables u and v before the execution of the loop. For each variable r_i $(1 \le i \le n)$, in turn, let r_i' be the *least* integer such that

$$a_A(r_i', j, q_A(j, r_i)) = z.$$

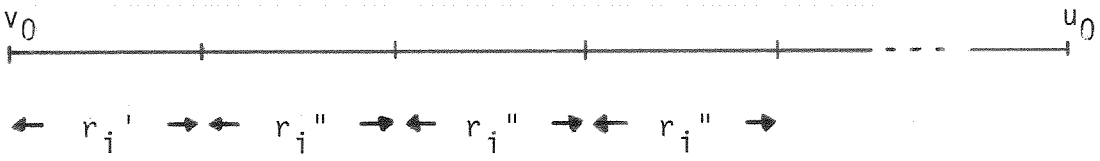
If there is such an r_i', $r_i' \le n$. Let r_i'' be the *least* integer such that

$$a_A(r_i'', j, z) = z.$$

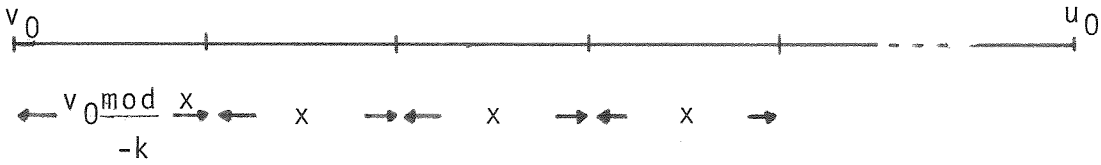
If there is such an r_i'', then $r_i'' \le n$. If either r_i' or r_i'' do not exist (they either both exist or neither exists), set $w_i = 0$ and proceed to the next variable, r_{i+1}.

The significance of r_i' and r_i'' is portrayed in Figure 1. The only times that instruction I_j can affect the outcome of variable r_i is when ι is set to $v_0 - r_i' - h_1 r_i'' \ge u_0$ for some nonnegative integer h_1. On any other pass the execution of I_j does not affect the outcome of r_i.

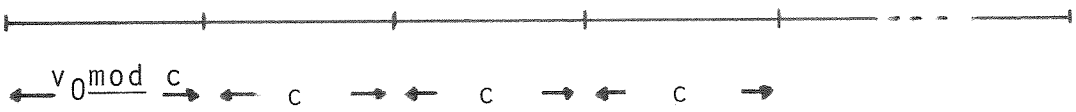
Since $b_A(j,y)=\iota$, we have $y=\iota+k$ for some (possibly negative) integer k. Note the earlier remark



I_j can only affect the outcome of r_i, if before the pass, ι was set to $v_0 - r_i' - h_1 r_i'' \geq u_0$ for some nonnegative integer h_1.



I_j only sets z to zero, on a pass, if before the pass, ι was set to $v_0 - v_0 \underline{\text{mod}}\ x - k - h_2 x \geq u_0$, for some nonnegative integer h_2.



A pass will occur only when ι is set to a value $v_0 - v_0 \underline{\text{mod}}\ c - h_3 c \geq u_0$, for some nonnegative integer h_3.

Figure 1. Passes of the loop and the respective values assigned to ι.

that we assume $u_0+k>0$. Then it is the case that the predicate "x|y" is true only on passes when ι was set to values

$$v_0 - v_0 \; mod \; x - k - h_2 x \geq u_0$$

for some nonnegative integer h_2. (See Figure 1.)

It is also true that the loop is only executed when ι is set to values

$$v_0 - v_0 \; mod \; c - h_3 \; c \geq u_0$$

for nonnegative integer values of h_3. (See Figure 1.)

Thus if the execution of I_j is to affect the outcome of variable r_i, then the instruction I_j must be executed on a pass when ι is set to a value greater than or equal to u_0 matching

$$\begin{aligned} &v_0 - r_i' - h_1 \; r_i' = \\ &v_0 - v_0 \; mod \; x - k - h_2 \; x = \\ &v_0 - v_0 \; mod \; c - h_3 \; c. \end{aligned}$$

In fact we need only find the value ι is set to the last time this occurs. This amounts to finding the smallest $h_1 \geq 0$ such that

$$\begin{aligned} &v_0 \; mod \; x + k + h_1 \; x = r_i' \; mod \; r_i'' \\ &\qquad \text{and} \\ &v_0 \; mod \; x + k + h_1 \; x = u_0 \; mod \; c \end{aligned}$$

If such an h_1 exists, $0 \leq h_1 \leq c * r_i''$. If such an h_1 exists and

$$v_0 - v_0 \; mod \; x - k - h_1 \; x \geq u_0$$

then set $w_i = v_0 - v \; mod \; x-k-h_1(x)$, otherwise set $w_1=0$. This means that the only pass of the loop affecting the outcome of variable r_i is when ι is set to w_i. Now we can proceed to the next variable, r_{i+1}.

When all variables r_i $1 \leq i \leq n$ have been considered we have the variables $w_1,...,w_n$ set to possible values for ι on passes that affect the outcome of variables $r_1,...,r_n$. The reader can verify that a U-program of length O(n) can be written to find the values of w_i $1 \leq i \leq n$. Next we sort the values of w_i, $1 \leq i \leq n$. This can be done with a U-program of length O(n logn). Now the entire loop structure

for i = u *to* v *by* c *do*;
    A;
end;

can be replaced by the following segment of code where r_i', $1 \leq i \leq n$, are new variable names.

$r_1' \leftarrow r_1;\ldots;r_n' \leftarrow r_n$
for $\iota = u_0$ *to* $w_1 \dot{-} c$ *by* c *do*;
    A_1;
    A_2;
end;
    A_1;
    $z \leftarrow 0$
    A_2;
    if $w_1 = 0$ then $r_i \leftarrow r_i'$ $(1 \leq i \leq n)$
    if $w_1 = 0$ then $w_1 + c \leftarrow u_0$
for $\iota = w_1 + c$ *to* $w_2 \dot{-} c$ *by* c *do*;
    A_1;
    A_2;
end;
    A_1;
    $z \leftarrow 0$
    A_2;
    if $w_2 = 0$ then $r_i \leftarrow r_i'$ $(1 \leq i \leq n)$
    if $w_2 = 0$ then $w_2 + c \leftarrow u_0$
    .
    .
    .
for $\iota = w_n + c$ *to* v_0 *by* c *do*;
    A_1;
    A_2;
end;

The length of the above code segment is $O((\text{length}(P))^2)$. The process in step 2 may need to be iterated for each instruction of type 7 in the loop structure. The resulting code then is of length $O((\text{length}(P))^3)$. The theorem then follows from the result in [7] mentioned in case 1. The time necessary becomes $O((\text{length}(P))^{36})$. □

One might question the necessity of the restrictions with respect to T-programs. We now show that it is unlikely that such a polynomial time translation from T-programs to U-programs without both restrictions. Note that this says nothing about the decidability of the equivalence problem for classes of such programs.

In [14] sufficient conditions were given, with respect to loop programs that do not allow nesting of loop structures that ensure that the 0-evaluation problem is PSPACE-hard. The conditions were:

1. Given a constant c, that in a parameterless program of length O(n) one must be able to produce a value greater than 2^c in a variable.
2. There must be an interpretation of variable values such that the logical functions "x.or.y", "x.and.y" and ".not.x" can be computed in such a program utilizing only the primitive instructions.

The first condition is met by $L_1(x \dot{-} x+1)$-programs, hence we only concern ourselves with the second condition. It should be clear that Theorems 1 and 2 imply that the 0-evaluation problem for S and T-programs can be done in polynomial time.

Proposition 1. The 0-evaluation problem for T-programs without restrictions 1 or 2 is PSPACE-complete.

Proof. The proof is by cases. It should be clear that the 0-evaluation problem for such programs is in PSPACE.

Case 1. Consider the absence of restriction 1. Let x being true be identified with the value of x being a multiple of 2 (even), and let x being false be identified with x not being a multiple of 2 (odd). Define the functions:

$$\text{"x.or.y"} = \begin{cases} \text{any odd number} & \text{if x is odd and y is odd} \\ \text{any even number} & \text{if x is even or y is even} \end{cases}$$

$$\text{".not.x"} = \begin{cases} \text{any odd number} & \text{if x is even} \\ \text{any even number} & \text{if x is odd} \end{cases}$$

Then the instruction "z←.not.x" can be simulated by "z←x+1" and the instruction "z←x.or.y" can be simulated by:

```
z ← 1
if 2|x then z ← 0
if 2|y then z ← 0
```

The fact that the 0-evaluation problem for such programs is PSPACE-complete follows from the results in [14].

Case 2. Consider the absence of restriction 2. Let x=1 mean that x is true and x=0 mean that x is false. Define the functions:

$$\text{"x.or.y"} = \begin{cases} 0 & \text{if } x = 0 \wedge y = 0 \\ 1 & \text{if } x = 1 \vee y = 1 \\ \text{don't care otherwise} \end{cases}$$

$$\text{".not.x"} = \begin{cases} 0 & \text{if } x = 1 \\ 1 & \text{if } x = 0 \\ \text{don't care otherwise} \end{cases}$$

Then the instruction "z←.not.x" can be simulated by:

```
z ← 1
if x|ι then z ← 0  (whenever ι > 0)
```

The instruction "z←x.or.y" can be simulated by

```
z ← 1
if x|ι then z ← 0
if y|ι then z ← 0
z ← .not. z        (whenever ι > 0)
```

Again the fact that the 0-evaluation problem is PSPACE-complete for such programs follows from [14]. □

It is interesting to note that if T-programs allowed the instruction "*if* x|y *then* I", where I is an instruction of the form u←u+1, u←u$\dot{-}$1 or u←v, then the equivalence problem becomes undecidable (even with both restrictions). This follows from the fact that such programs can compute integer division. For example, the program

```
w ← x
for ι = 1 to x by 1 do;
    if y|ι then w ← w ∸ 1
end
w ← x ∸ w
```

computes x/y. The undecidability of the equivalence problem then follows from the undecidability of Hilbert's tenth problem[20]. The equivalence problem also becomes undecidable for either S or T-programs if the increment (c) is not constrained to be a constant.

3. THE EQUIVALENCE PROBLEM AND CORRECTNESS FORMULAS FOR U-PROGRAMS

In this section we look at the decidability problem for the class of U-programs as well as the decidability of a restricted class of correctness formulas. First we define a class of unquantified logical formulas, $\mathcal{F}$. The formulas are unquantified because we do not permit quantifiers, either $\forall$ or $\exists$ in any formula. Such a formula is valid, however, if it is true for all possible assignments. This then amounts to considering every variable to be universally quantified.

Definition. The class of logical formulas $\mathcal{F}$ is composed of unquantified logical formulas of the form:

$F(x_1,...,x_n)$ where,

$F(x_1,...,x_n)$ is any logical expression built up from integer constants and the variables $x_1,...,x_n$ such that:

1. The only arithmetic operators are + and -.
2. The relational operators are $<$, $=$, $\neq$, $\leq$ and | (divides).
3. The logical operators are $\wedge$, $\vee$ and $\neg$.

The following two lemmas show the relationship of the formulas of $\mathcal{F}$ to the class of U-programs (and therefore S and T programs).

Lemma 1. Let $F(x_1,...,x_n)$ be a formula in $\mathcal{F}$. Then there exists a U-program P such that P is equivalent to the zero program on n-inputs[3] if and only if $\forall x_1...\forall x_n\ F(x_1,...,x_n)$ is true (i.e., F is valid). Furthermore, P can be found in polynomial time and the length of P is linear in the length of F.

Proof. Left to the reader. □

Lemma 2. Let P_1 and P_2 be U-programs. Then there exists a formula F in $\mathcal{F}$ such that F is valid if and only if P_1 and P_2 are equivalent. Furthermore, F can be found in polynomial time and the length of F is linear in the lengths of P_1 and P_2.

Proof. Let P_1 and P_2 be U-programs, each with input variables $x_1,...,x_n$ and output variables $y_1,...,y_m$. In a straightforward manner one can construct a U-program P of the form;

---

[3]The zero program on n-inputs is the U-program with n input variables that outputs 0 for all possible inputs, e.g., the program:

```
Input(x1,...,xn)
  z ← 0
Output(z)
```

P : Input(x_1,...,x_n)
I_1
.
.
.
I_l
Output(z)

where I_1,...,I_l are instructions in U, and P on inputs a_1,...,a_n outputs a zero (z=0) if and only if P_1 and P_2 output the same values on inputs a_1,...,a_n. Hence P_1 is equivalent to P_2 if and only if P is equivalent to the zero-program on n inputs.

Let v_1,...,v_l be new variables not used in P. Let f(j,x) ($1 \leq j \leq l+1$) be the largest integer such that $1 \leq f(j,x) < j$ and where x is the variable to be altered by $I_{f(j,x)}$. (The variable to be altered by the instruction "*if* x=0 *then* y ← z" is y.) Let

$$g(j,x) = \begin{cases} v_{f(j,x)} & \text{if } f(j,x) \text{ is defined} \\ x & \text{otherwise} \end{cases}$$

Note that the range of g is the set of variable names in $\{x_1,...,x_n, v_1,...,v_l\}$ and the domain of g is $\{i | 1 \leq i \leq l+1\} \times \{$the variable names used in P$\}$.

We now construct a program P' of the form

P' : Input(x_1,...,x_n)
I_1'
.
.
.
I_l'
Output(g(l+1,z))

where I_1',..,I_l' are each sequences of instructions in U.

The sequence of instructions I_j' is defined by case according the instruction I_j. The transformation is shown in the table below (Figure 2). In the last row, involving the case where I_j is the instruction "*if* x=0 *then* I", I is assumed to be a U-instruction of the form 1-5 and u is assumed to be the variable to be altered by I_j if x=0. I' is defined to be the instruction indicated by making a similar transformation on instruction I, and I" is defined to be the logical formula generated by making a similar transformation on I as I_j" is to I_j.

I_j	I_j'	I_j''
$x \leftarrow 0$	$v_j \leftarrow 0$	$(v_j=0)$
$x \leftarrow x+y$	$v_j \leftarrow g(j,x)+g(j,y)$	$(v_j=g(j,x)+g(j,y))$
$x \leftarrow x \dot{-} y$	$v_j \leftarrow g(j,x) \dot{-} g(j,y)$	$(g(j,x) \geq g(j,y) \wedge v_j = g(j,x)-g(j,y)) \vee (g(j,x)<g(j,y) \wedge v_j=0)$
$x \leftarrow x/k$	$v_j \leftarrow g(j,x)/k$	$(k*g(j,x) \leq v_j < k*(g(j,x)+g(j,x))$
$x \leftarrow x$ *mod* y	$v_j \leftarrow g(j,x)$ *mod* $g(j,y)$	$(g(j,y)=0 \wedge v_j=g(j,x) \vee (g(j,y)>0 \wedge g(j,y) \mid (g(j,x)-v_j)$
if $x=0$ *then* I	$v_j \leftarrow g(j,u)$ *if* $g(j,x)=0$ *then* I'	$(g(j,x)>0 \wedge v_j=g(j,u)) \vee (g(j,x)=0 \wedge I'')$

Figure 2

Now after the execution of the sequence of instructions I_j' in P', the value of v_j is the value of the variable on the left hand side of I_j after the execution of instruction I_j in P. The variable g(j,x) is just the name of the variable in P' which before the execution of I_j', contains the same value as the variable x before the execution of I_j in P. Note that the variables $x_1,...,x_n$ are not altered by P' and that once an assignment to a variable v_j is made in P', the variable v_j is not again altered. Clearly then P' is equivalent to P. Furthermore the length of P' is linear in the length of P.

Now we construct a formula F in $\mathcal{F}$ which is valid if and only if P' (and hence P) is equivalent to the zero-program on n-inputs. F is of the form

$$(\bigwedge_{i=1}^{n} (x_i \geq 0)) \wedge (\bigwedge_{j=1}^{l} I_j'') \wedge (g(l+1,z) = 0)$$

where the clauses I_j'' are from the table in Figure 2. It is straightforward to show that F is valid if and only if P' outputs a zero for all inputs. The details will be left to the reader. □

Lemmas 1 and 2 have shown that the decidability of the equivalence problem for U-programs is exactly that of deciding the validity of $\mathcal{F}$ formulas. In [9] an algorithm to decide the truth of logical formulas of the form $\exists x_1 ... \exists x_n \; F(x_1,...,x_n)$ (over the nonnegative integers), where F is in $\mathcal{F}$, was given. Since any formula of the form $\forall x_1 ... \forall x_n \; F(x_1,...,x_n)$ is true if and only if $\exists x_1 ... \exists x_n \; \neg F(x_1,...,x_n)$ is true, this also yields a decision procedure for the validity of formulas in $\mathcal{F}$. The procedure given in [9] runs in polynomial space. This seems to be the best we can do at this time. It should be noted, however, that the inequivalence problem is NP-hard and hence an exponential time algorithm is the best we can hope for. The NP-hardness follows from results in [4].

Next we go a step further and explore the question of decidability for simple correctness formulas of the form $\{p\}S\{q\}$, where p and q are formulas in $\mathcal{F}$ and S is a U-program. Using the strongest post con-

dition calculus of [10], one can derive a formula $F(x_1,...,x_n)$ such that $\forall x_1...\forall x_n\ F(x_1,...,x_n)$ is true if and only if {p}S{q} is true. Unfortunately the length of F in general is exponential in the length of {p}S{q}. This results since the strongest post condition (SPC) of "*if* x=0 *then* y←z" and the formula p(x,y,z), for example would be:

$$(x=0 \wedge SPC(y\leftarrow z,\ p(0,y,z))) \vee (x>0 \wedge p(x,y,z))$$

Hence we provide an alternate proof.

Theorem 3. Let {p}S{q} be a correctness formula where p and q are in $\mathcal{F}$ and S is a U-program. There exist a formula F in $\mathcal{F}$ such that F is valid if and only if {p}S{q} is valid. Furthermore, F can be found in polynomial time and the length of F is linear in the length of {p}S{q}.

Proof. Let {p}S{q} be such a correctness formula. Let $x_1,...,x_n$, $u_1,...,u_m$ be the variables used in S. Using Lemma 1 construct programs P_1 and P_2 such that P_1, on input $a_1,...,a_n$, outputs a zero if and only if $p(a_1,...,a_n)$ is true and P_2, on input $a_1,...,a_n$, $b_1,...,b_m$, outputs a zero if and only if $q(a_1,...,a_n, b_1,...,b_m)$ is true. Let the input variables of P_1 be $x_1',...,x_n'$ and let the output variable be z'. Let the input variables of P_2 be $x_1,...,x_n$, $u_1,...,u_m$ and let the output variable of P_2 be z". Let w be a variable not used in P_1, P_2 or S. From P_1, P_2 and S we can construct a program S'.

```
S' : input(x1,...,xn)
     x1' ← x1;...;xn' ← xn
     P1;
     S;
     P2;
     w ← z';
     if z"= 0 then w ← 0
   output(w)
```

Clearly S' outputs a zero for all inputs if and only if {p}S{q} is valid. The theorem now follows from Lemma 2. □

Let $\mathcal{P}$ be a class of programs and $\mathcal{L}$ a class of assertions. Then the validity of correctness formulas over $\mathcal{P}$ and $\mathcal{L}$ is equivalent to the equivalence problem for $\mathcal{P}$, if it is the case that for each formula F in $\mathcal{L}$, that there is a program P such that F is true (for a given input) if and only if P outputs a zero (for that input), and vice versa. This was actually the case in [3], since in the earlier paper [2] it was shown that the class of Presburger formulas was realized by he class of L_1(x←0, x←y, x←x+1, x←x$\dot{-}$1)-programs. The size of such a realizing program for a given Presburger formula, however, is at least doubly exponential in the length of the formula. If in [3] only unquantified assertions were allowed, the problem of deciding the truth of a correctness formula would be Co-NP complete. Hence an exponential time procedure would more than likely be required.

The previous theorem can be generalized somewhat in that the formula p (q) can also be of the form p∧P or p∨P (q∧Q or q∨Q) where P (Q) is a Presburger formula. It follows from results in [2] that one can construct a U-program for a Presburger formula (with n-free variables) that is equivalent to the zero program on n-inputs if and only if the Presburger formula is valid. The constructed program need not contain any instructions of the form "x←x *mod* y". This is almost Lemma 1. Unfortunately, the length of the resulting program is at least double exponential in the length of the formula[2]. Hence the complexity

of this problem cannot be the same. This follows from the complexity of deciding the validity of Presburger formulas[5].

Consider once again only formulas in $\mathcal{F}$. Then we can observe that for quantified formulas in $\mathcal{F}$, the validity problem is undecidable even if we limit the formulas to a single occurrence of the $\exists$ quantifier[22]. In fact, this is very easy to see using the strongest post condition calculus and adding the instruction "z←*lcm*(x,y)" to U-programs.

$$\begin{aligned}&SPC(z \leftarrow lcm(x,y), Q(x,y,z)) = \\ &\exists w\ Q(x,y,w) \wedge z = lcm(x,y) \equiv \\ &\exists w\ Q(x,y,w) \wedge x|z \wedge y|z \wedge \forall v \\ &\quad ((x|v \wedge y|v) \supset z|v)\end{aligned}$$

The undecidability of such correctness formulas follows in a straightforward manner from [20], since multiplication can be simulated using the "z←*lcm*(x,y)". This follows since $lcm(x,x+1)=x^2+x$. Those readers familiar with the SPC calculus should note that the logical formulas derivable from such correctness formulas, {p}S{q} where p and q are formulas in $\mathcal{F}$ and S is a U-program, are of the form,

$$\begin{aligned}&\forall \bar{x}[\exists \bar{w} W(\bar{w}, \bar{x}) \supset q(\bar{x})] = \\ &\forall \bar{x} \forall \bar{w}[\neg W(\bar{w}, \bar{x}) \vee q(\bar{x})]\end{aligned}$$

It is then the universal quantifier appearing in the SPC(z←*lcm*(x,y), Q(x,y,z)) that is the problem. An interesting question then, is whether the same results for the instruction "z←*gcd*(x,y)" hold. The strongest post condition calculus produces a similar formula as it did for the "z←*lcm*(x,y)" instruction but we are unable to answer the decidability of the validity problem for such correctness formulas, at this time.

In [17,23], it was shown that unquantified Presburger array formulas have a decidable validity problem. In fact it was shown that special predicates could be added to the formulas (in a limited way) and the validity problem remains decidable. Such predicates considered were those concerning properties of the arrays such as orderedness or the property that one array is a permutation of another. The logical formulas in Presburger array theory are equivalent to correctness formulas of the form {p}S{q}, where p and q are logical formulas similar to those in $\mathcal{F}$ although they allow array elements as terms and do not allow the "|" predicate, and S is a {x←1, x←x+y, x←x$\dot{-}$y, x←x/k, x←A(i), A(i)←x, if x=0 then I}-program (I can be any of the other types of instructions). The proofs in [17,23] reduce the Presburger array formulas to equivalent unquantified Presburger formulas. Thus the validity problem for such formulas is decidable. The reduction is such that, even if an additional function f(x) were allowed in the Presburger array formula, the resulting formula is unaffected except that it too contains occurrences of f(x). Hence it is the case that even if we add the *mod* function to the theory in [17,23] it still yields a decidable validity problem.

4. 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 [14-16,18].

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$\dot{-}$1, *if* x=0 *then* y←y+1)-programs, respectively, in polynomial time. This is an improvement over the exponential time needed in [18].

Theorem 4. 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 [7,12] 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 ($1 \leq i \leq k$) are 0/1 valued. Now v can be considered to be a 0/1 valued vector of length k. The function SHIFT(v,j), $1 \leq |j| \leq k$ 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 $x \leftarrow x \dot{-} y$. 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 [18] it was claimed without proof that this could be done in polynomial time. However in [14], 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 5. Let P be an L_1(BB1, 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 $(A')^{l}2(=A';A';...;A')$ 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_1^h and l_2^h denoting the number of executions of A_i' necessary until the first repeating state, say q_h', occurs and then reoccurs again, respectively. From $(A')^{l_2^h}$ 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- 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←0, x←y, x←x+1)-program. In order to see that it can, the reader should consult [24], 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, *if* x=0 *then* y←c)[4]-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 $\#a_1^{i_1}\#...\#a_{m+n}^{i_{m+n}}$, accept if and only if the program on input $i_1,...,i_m$ outputs $i_{m+1},...,i_{m+n}$. The result now follows from the results concerning nondeterministic reversal bounded CM of [11]. See [11] for precise definitions.

Theorem 6. Every L_1(BB2, *if* x=0 *then* y←c)-program computes a Presburger function.

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

Proof. 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 $\cup$ {*if* x=0 *then* y←c}. Using techniques from [7], 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 $\cup$ {*if* x=0 *then* y←c}} → {set of variable names}, as follows. Let B be a (possibly null) segment of code and z a variable name:

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

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

Create the directed graph G^A with a node for each variable mentioned in B^A, which contains the edge u→v, if and only if $g(u,B^A)=v$. The reader can verify that G^A is actually a collection of connected subgraphs $G_1^A,\ldots,G_m^A$ 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 $x_{i1},\ldots,x_{in_i}$ and those not contained in the cycle which we shall denote as $y_{i1},\ldots,y_{ik_i}$. 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 $\#a_1^{i_1}\#\ldots\#a_{m+n}^{i_{m+n}}\#$, accept if and only if the program on inputs $i_1,\ldots,i_m$ outputs $i_{m+1},\ldots,i_{m+n}$.

M will have the following counters;

c<x,i,j>	$1\leq i\leq m$, $1\leq j\leq n_i$
c<y,i,j>	$1\leq i\leq m$, $1\leq j\leq k_i$
c<x,i,j,s,t>	$1\leq i\leq m$, $1\leq j,t\leq n_i$, $1\leq s\leq k$
c<0>	(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<x,i,j>	$1\leq i\leq m$, $1\leq j\leq n_i$
b<y,i,j>	$1\leq i\leq m$, $1\leq j\leq k_i$

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.

$B\langle x,i,j,s,t\rangle$ 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 $B\langle x,i,j,s,t\rangle$ will be "guessed". Let the statement labeled s be "*if* u=0 *then* x_{ij}←c". If the initial value of $B\langle x,i,j,s,t\rangle$ 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 $root\langle x,i,j\rangle = t$, exactly $B\langle x,i,j,s,t\rangle$ times. If $B\langle x,i,j,s,t\rangle$ 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 $root\langle x,i,j\rangle$ 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. $p\langle x,i,j\rangle$ and $p\langle y,i,j\rangle$ are set to $\langle x,i,j\rangle$ and $\langle y,i,j\rangle$, respectively (for appropriate values of i and j).
2. $b\langle x,i,j\rangle$ and $c\langle x,i,j\rangle$ are set so that the input value of x_{ij} is equal to the value in $b\langle x,i,j\rangle$ plus the value in $c\langle x,i,j\rangle$, in a way such that $b\langle x,i,j\rangle = \min\{2 * D,$ the input value of $x_{ij}\}$.
3. $b\langle y,i,j\rangle$ and $c\langle x,i,j\rangle$ 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 ← 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 ← 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 ← u ∸ 1	The buffer for variable u is decremented.
s: *if* u= 0 *then* x_{ij} ← 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 variables whose pointer contents are equal to those of p<x,i,j>. The reader should note that the current value of any two of these variables can differ by at most D, (unless p<x,i,j> is λ). The counter pointed to by p<x,i,j> and the buffer b<x,i,j> are adjusted so that their combined value is the same but that b<x,i,j> = 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<y,i,j> that points to the same counter (other than λ), changes are made to b<y,i,j> 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<x,i,j,s,t> = 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<x,i,j> 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<x,i,j> = 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<x,i,t> = 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<x,i,j> = j that some pointer p<y,r,s> might be the same as p<x,i,j>. (The reader should note that this is not possible for another x variable in the cycle, unless the value of p<x,i,j> 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<x,i,j> 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 l 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 [16], 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 $\dot{-}$ 1,

where w is a new variable. (The converse was shown in [16].) Unfortunately we have been unable to resolve the computational power of L_1(BB2, *if* x=0 *then* y←y$\dot{-}$1). The "*if* x=0 *then* y←y$\dot{-}$1" construct seems similar to the "*if* x=0 *then* y←y+1" construct, but as pointed out in [14] functions of one variable computed over BB2 $\cup$ {*if* x=0 *then* y←y$\dot{-}$1} are monotonic. Thus the proof techniques used in [16] (as well as those presented in the last theorem) do not seem to work with this language. This same problem arose in [14], where the authors were able to show that the 0-evaluation problem for this language is PSPACE-complete.

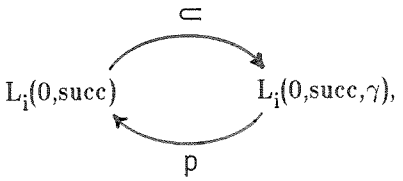
5. L_i(BB)-Programs

In this section, we consider other problems that arose with the claims in [18]. In [18] the following eleven primitive instructions were allowed. In what follows, the accompanying abbreviations are used in place of the instructions.

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

The following are theorems given in [18] along with our accompanying comments. The numbering of the theorems corresponds to that in [18] (e.g. Theorem III.4 in this paper corresponds to Theorem 4 in Section III of [18]).

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

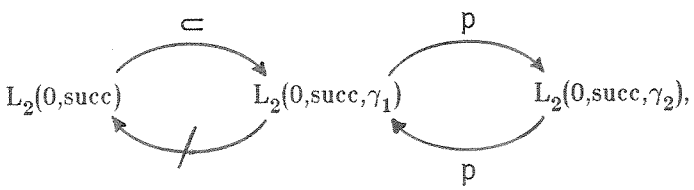


where the degree of the polynomial translation depends on γ and never exceeds 2.

Comments:

This is not true when the operation of "+" is in γ. For a proof see [14].

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

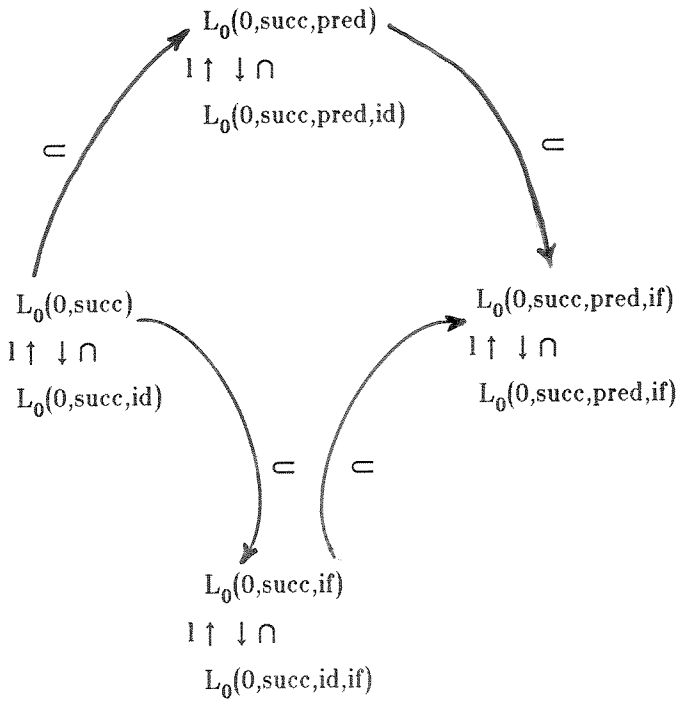


where the degrees of the polynomial translations depend on γ_1 and γ_2, but never exceed 2.

Comments:

This is not true for all possible choices of γ_1 and γ_2. For example it is not true when $\gamma_1 = \{\text{id}\}$ and $\gamma_2 = \{\text{id},+\}$. For a proof see [14]. The above theorem also states that $L_2(0,\text{succ}) \not\subset L_2(0,\text{succ},\gamma_1)$ 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.

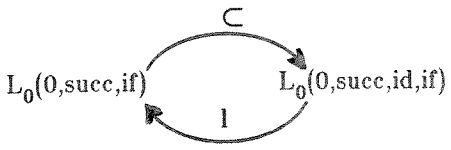
Theorem III.4.



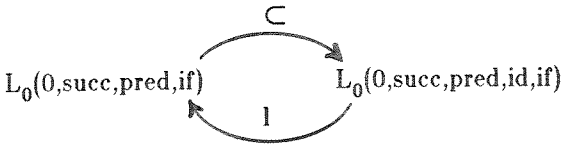
Again here, if an omitted arrow is not obtainable by composition from the arrows in the diagram, then it is a case of non-translatability.

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

Comments:

This was shown to be in error in [16]. If we substitute $\mathcal{C}_1$(0,succ,pred,id) for $\mathcal{C}_1$(0,succ,pred,if) then the theorem is true. In fact this was shown in [7] (see also [2,3,12]).

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,γ) 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[2]. 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 V.5 and V.6 is also taken from [18]. 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 [16], 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 [7]. Then Theorem 14 of [7] is exactly this result. The discussion following Theorem 5 also needs attention. The bound mentioned can be improved by 4 levels of exponentiation (see [7]), and thus Theorem 6 is in error.

Other incorrect claims appear in [18], but they precipitate from those already mentioned. A full accounting can be found in [15].

REFERENCES

[1] Alt, H., Functions Equivalent to Integer Multiplication, *Lecture Notes in Computer Science, No. 85: Automata, Languages and Programming (ICALP 80)*, Springer Verlag, 1980.

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

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

[4] Constable, R., Hunt, H. and Sahni, S., On the Computational Complexity of Scheme Equivalence, Proc. 8th Ann. Princeton Conf. on Information Sciences Systems, Princeton, NJ, 1974.

[5] Fischer, M. and Rabin, M., Super-Exponential Complexity of Presburger Arithmetic, *Project Mac. Tech. Memo 43*, MIT, Cambridge, 1974.

[6] Gurari, E., Decidable Problems for Powerful Programs, to appear in *J. ACM.*

[7] Gurari, E. and Ibarra, O., The Complexity of the Equivalence Problem for Simple Programs, *J. ACM* 28, 3 (July 1981), pp. 535-560.

[8] 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.

[9] Gurari, E. and Ibarra, O., Two-Way Counter Machines and Diophantine Equations, *J. ACM* 29, 3 (July 1982), pp. 863-873.

[10] Hoare, C., An Axiomatic Basis of Computer Programming, *CACM*, Vol. 12, No. 10, pp. 576-580, 1969.

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

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

[13] Ibarra, O. and Leininger, B., On the Equivalence and Simplification Problems for Simple Programs, *J. ACM*, 30, 3 (July 1983), pp. 641-656.

[14] Ibarra, O., Leininger, B. and Rosier, L., A Note on the Complexity of Program Evaluation, accepted for publication in *Math. Systems Theory.*

[15] 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).

[16] Ibarra, O. and Rosier, L., Simple Programming Languages and Restricted Classes of Turing Machines, *Theoretical Computer Science*, Vol. 26, No. 1 and 2, pp. 197-220, September 1983.

[17] Jefferson, D., Type Reduction and Program Verification (Ph.D. thesis), Department of Computer Science, Carnegie-Mellon University, 1980.

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

[19] Knuth, D., The Art of Computer Programming: Vol. 1, Fundamental Algorithms, Addison-Wesley, Reading, MA, 1973.

[20] Matijasevic, Y., Enumerable Sets are Diophantine, *Dodl. Akad. Nauk.*, SSSR 191 (1970), pp. 279-282.

[21] Meyer, A. and Richie, D., The Complexity of Loop Programs, in *Proc. 22nd Nat. Conf. of the ACM*, Thompson Book Co., Washington, DC, 1976, pp. 465-469.

[22] Robinson, J., Definability and Decision Problems in Arithmetic, *J. Symbolic Logic*, 14, pp. 98-114 (1949), MR 11, 151.

[23] Suzuki, N. and Jefferson, D., Verification Decidability of Presburger Array Programs, *J. ACM* 27, 1 (Jan. 1980), pp. 191-205.

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