A Practical Decision Procedure for Arithmetic with Function Symbols

ROBERT E. SHOSTAK

SRI International, Menlo Park, California

ABSTRACT A practical procedure is presented for an extension of quantifier-free Presburger arithmetic that permits arbitrary uninterpreted predicate and function symbols This theory includes many of the formulas one tends to encounter in program verification and is powerful enough to encode the semantics of array operators as well as *MAX*, *MIN*, and *ABSVALUE* An implementation of the procedure has proved to be of great value in a program verification system developed at SRI for the United States Air Force

KEY WORDS AND PHRASES *theorem-proving, Presburger arithmetic, program verification*

CR CATEGORIES 3 64, 3 66, 5 21

1. *Introduction*

The procedure described here operates over an extension of the class of unquantified Presburger formulas. Briefly, Presburger formulas are those that can be built up from integers, integer variables, addition,[1] the usual arithmetical relations ($<, \leq, >, \geq, =$), and the first-order logical connectives. The formula $(\forall x)(\exists y)3x + y = 2 \supset x < y$, for example, falls within the class. The subclass of unquantified Presburger formulas consists of those Presburger formulas having no quantifiers.

The extension of unquantified Presburger we shall be dealing with introduces, for each $n \geq 0$, an unlimited number of n-ary function symbols (interpreted as functions from Z^n to Z) and n-ary predicate symbols (interpreted as relations over Z^n).

The formula

$$x < f(y) + 1 \wedge f(y) \leq x \supset (P(x, y) \equiv P(f(y), y)),$$

for example, is a member of the extended class. One can easily check that this particular formula is valid, that is, that it evaluates to true for all integers x, y, z, no matter what monadic integer function is assigned to f and dyadic integer relation to P.

Function symbols may appear in any term context and may have arbitrary terms as arguments, including expressions containing function symbols. For example, the formula $g(x + 2f(y)) = 4$ is a member of the class.

The extended theory includes a surprisingly large proportion of the formulas encountered in program verification. It is particularly well suited to programs that manipulate arrays and other data structures that can be modeled as uninterpreted functions. The semantics

This work was supported in part by RADC Contract F30602-76-C-0204 and AFOSR Contract F44620-73-C-0068

Author's address Computer Science Laboratory, SRI International, 333 Ravenswood Ave, Menlo Park, CA 94025

[1] Arbitrary multiplication is not permitted It is convenient, however, to use multiplication by constants as an abbreviation for repeated addition, $x + x + x$ is thus written $3x$

of the McCarthy [11] *ACCESS* and *CHANGE* array primitives are easily encoded within the theory; for example, the formula

$$M = ACCESS(CHANGE(A, I, V), J + 2)$$

can be mechanically translated to an equivalent formula

$$\begin{aligned} &J + 2 = I \supset M = V \\ \wedge\ &J + 2 \neq I \supset M = A[J + 2], \end{aligned}$$

where A is now an interpreted function symbol.

A number of other constructs, including *MAX*, *MIN*, and *ABSVALUE*, can be dealt with in a similar manner. For instance, the valid formula

$$x = y + 2 \supset MAX(x, y) = x$$

containing the interpreted function symbol *MAX* translates to

$$\begin{gathered} [x \geq y \supset MAX(x, y) = x \wedge y \geq x \supset MAX(x, y) = y] \\ \supset \\ [x = y + 2 \supset MAX(x, y) = x], \end{gathered}$$

where *MAX* is now uninterpreted.

The discussion that follows is presented in six sections. Section 2 provides historical perspective and cites related work. Section 3 describes a decision method for the unextended class that forms the basis of the extended procedure. Sections 4–6 establish the decidability of the extended class and introduce a basic version of the procedure. Section 7 presents an extremely efficient refinement of the basic procedure.

2. *Related Work*

The class of closed Presburger formulas was first shown to be decidable by Presburger [14] in 1929. The best-known decision procedure for it (described by Kreisel and Krevine [9], among others) is based on a method of elimination of variables. In its raw form, the algorithm is prone to combinatorial explosion, and is therefore not practical for nontrivial problems. More efficient versions of the method of elimination have since been given by Cooper [4]. Cooper's most recent procedure [5] is the most efficient known algorithm for full Presburger. Oppen [13] has shown that Cooper's algorithm is probably the best one can do in the worst case (deterministic time complexity on the order of $2^{2^{2^n}}$ in the length of the formula).

More recent work has focused on the subclass of Presburger without quantifiers. The decision complexity of this subclass is no worse than exponential, making it substantially easier to decide (in theory, at least) than full Presburger. A number of theorem-provers for this class (Bledsoe [3], Shostak [16]) have been successfully implemented and used for program verification.

Oppen[2] has noted that the extension of the unquantified subclass dealt with in this paper is also no worse than exponential deterministic time complexity. It is, perhaps, surprising that the incorporation of predicate and function symbols does not give away decidability altogether. Downey [6] has proved that the addition of even a single predicate symbol to the language of full Presburger produces a reduction class.

An implementation of our procedure (coded in INTERLISP for the DEC PDP-10) has been used for the past two years in conjunction with a system for verifying JOVIAL programs (Elspas et al. [7]). We have found that most formulas of a few lines are handled in seconds, and that larger formulas are generally decided much more quickly than by humans.

[2] Private communication

A number of other theorem-provers dealing with similar theories are currently under development. The systems of Suzuki [17] and of Nelson and Oppen [12] are among these.

3. *The Unextended Class*

The new procedure can best be explained in relation to the author's adaptation [15] of Bledsoe's [3] method for handling the unextended unquantified class. This method is carried out in two stages. In the first stage the formula F to be decided is reduced to a set of integer linear programming problems (ILP's) with the property that F is valid if and only if *none* of the problems has a solution. In the second stage the ILP's are tested one by one for solvability. If one is found to have an integer solution, the solution provides a model for $\neg F$ and therefore a counterexample for F.

Let us now consider these steps in greater detail.

The reduction to a set of ILP's consists of expanding the negation of F into a disjunctive normal form:

$$\neg F \equiv G_1 \lor G_2 \lor \cdots \lor G_p,$$

where each G_i is a conjunction of linear inequalities of the form $A \le B$. During the expansion, terms of the form $A = B$ are replaced by $(A \le B \land B \le A)$. Similarly, $A \ge B$ is replaced by $B \le A$; $A < B$ is replaced by $A + 1 \le B$; $\neg(A \le B)$ is replaced by $B + 1 \le A$; and so on. The conjunctions $G_1, G_2, \ldots, G_p$ in the expanded form make up the set of ILP's. Suppose, for example, that

$$F \equiv [(x < 3y + 2) \land x = 1] \supset x = y.$$

Then

$$\begin{aligned} \neg F &\equiv \neg[x < 3y + 2 \land x = 1 \supset x = y] \\ &\equiv x < 3y + 2 \land x = 1 \land \neg(x = y) \\ &\equiv x \le 3y + 1 \land x \le 1 \land x \ge 1 \land (x + 1 \le y \lor y + 1 \le x) \\ &\equiv (x \le 3y + 1 \land x \le 1 \land 1 \le x \land x + 1 \le y) \\ &\quad \lor (x \le 3y + 1 \land x \le 1 \land 1 \le x \land y + 1 \le x) \end{aligned}$$

so

$$G_1 \equiv x \le 3y + 1 \land x \le 1 \land 1 \le x \land x + 1 \le y$$

and

$$G_2 \equiv x \le 3y + 1 \land x \le 1 \land 1 \le x \land y + 1 \le x$$

$\neg F$ is satisfiable if and only if either G_1 or G_2 has a solution in integers, and so F is valid if and only if neither G_1 nor G_2 has such solutions. The G_i's are now tested for feasibility using an integer programming algorithm.

Continuing the above example, it is easy to see that the ILP G_1 has the integer solution $x = 1$, $y = 0$. These values provide a model for $\neg F$ and hence a counterexample to F. (Counterexamples are also provided by G_2, which is integer feasible as well.)

It should be noted that the completeness of the procedure depends on the completeness of the method used to test for integer feasibility. At present, there are no known complete integer programming methods that are also efficient. In practice, however, this point is of little concern. We have implemented both the SUP-INF method [3, 16] and Gomory's method [8], both of which are reasonably fast, and have never, in two years of experimentation, come across natural examples in which the incompleteness of these methods was manifested.[3]

4. *Decidability of the Extended Class*

The decision mechanism for the extended class elaborates upon a method for reducing an

[3] Gomory's method is incomplete in the sense that it is not guaranteed to terminate

arbitrary formula F in this class to an equivalid formula $\hat{F}$ in the unextended class The reduction is carried out in two steps, the first eliminating uninterpreted predicate symbols and the second eliminating uninterpreted function symbols:

(1) For each uninterpreted n-ary predicate symbol P occurring in F, let f_p be a new n-ary function symbol Obtain F' from F by replacing each atomic formula $P(t_1, t_2, \ldots, t_n)$ by the formula $f_p(t_1, t_2, \ldots, t_n) = 0$

(2) For each pair $f(t_1, t_2, \ldots, t_n)$, $f(u_1, u_2, \ldots, u_n)$ of distinct terms or subterms of terms in F' with the same outermost uninterpreted function symbol f, construct the following axiom

$$t_1 = u_1 \wedge t_2 = u_2 \wedge \cdots \wedge t_n = u_n \supset f(t_1, t_2, \ldots, t_n) = f(u_1, u_2, \ldots, u_n)$$

Let F'' be the formula given by

$$A_1 \wedge A_2 \wedge \cdots \wedge A_r \supset F',$$

where the A_i's are the axioms so constructed Next, for each term t occurring in F'' that has an uninterpreted outermost function symbol, let x_t be a new integer variable Obtain $\hat{F}$ from F'' by replacing each such term t with x_t (In the case where one such term is nested within another, the larger term is replaced)

Consider, as an example, the valid formula F given below:

$$[(P(z) \supset z = 1) \wedge g(y) = z + 4] \supset [f(g(y)) = f(3 + 2z) \vee \neg P(1)].$$

Using step (1) to eliminate the uninterpreted function symbol P, we obtain the formula F' given by

$$[(f_p(z) = 0 \supset z = 1) \wedge g(y) = z + 4] \supset [f(g(y)) = f(3 + 2z) \vee f_p(1) \neq 0].$$

Applying step (2), we observe that F' contains two pairs of distinct terms with the same outermost function symbol—the pair $f_p(z), f_p(1)$ and the pair $f(g(y)), f(3 + 2z)$ The formula F'' is therefore given by

$$[z = 1 \supset f_p(z) = f_p(1)] \wedge [g(y) = 3 + 2z \supset f(g(y)) = f(3 + 2z)]$$
$$\supset$$
$$[(f_p(z) = 0 \supset z = 1) \wedge g(y) = z + 4] \supset [f(g(y)) = f(3 + 2z) \vee f_p(1) \neq 0].$$

Letting $f_p(z)$, $f_p(1)$, $g(y)$, $f(g(y))$, and $f(3 + 2z)$ be replaced by x_1, x_2, x_3, x_4, and x_5, respectively, we obtain $\hat{F}$:

$$[z = 1 \supset x_1 = x_2] \wedge [x_3 = 3 + 2z \supset x_4 = x_5]$$
$$\supset$$
$$[(x_1 = 0 \supset z = 1) \wedge x_3 = z + 4] \supset [x_4 = x_5 \vee x_2 \neq 0].$$

This latter formula is contained within the unextended class, and can therefore be decided by using the method described in Section 3.

The reduction just described is quite similar to Ackermann's [1] method for eliminating function symbols from universally quantified equality formulas in predicate calculus with function symbols and identity. The correctness of the reduction can be proved straightforwardly; given a model for $\neg F$, one can construct a model for $\neg\hat{F}$, and conversely. The details are easily gleaned from Ackermann's proof, and so are omitted here.

Although the reduction confirms the decidability of the extended class, it does not of itself provide a very good computational method. Recall that in step (2) of the reduction, an axiom is constructed for each pair of terms (including nested terms) with the same outermost uninterpreted function symbol. The number of such axioms is thus proportional (in the worst case) to the square of the length of the given formula. Moreover, each axiom at least triples the number of ILP's that must be solved in deciding the reduced formula $\hat{F}$.

Suppose, for example, that the axiom $x = y \supset f(x) = f(y)$ is generated in step (2). It is easy to check that the expansion into disjunctive normal form described in Section 3 produces three disjuncts (corresponding to the cases in which $x < y$, $x > y$, and $f(x) =$

$f(y))$ for each disjunct that would have been produced in the absence of the axiom.

As an illustration of the kind of combinatorial explosion that can result, consider the formula F given by

$$x \le g(x) \wedge x \ge g(x) \supset x = g(g(g(g(x)))).$$

An axiom must be generated for every pair of terms among $g(x)$, $g(g(x))$, $g(g(g(x)))$, and $g(g(g(g(x))))$. There are six such axioms, each one tripling the number of disjuncts appearing in the disjunctive normal form expansion of the corresponding reduced formula $\hat{F}$. Deciding $\hat{F}$ therefore entails the solution of 3^6 (= 729) ILP's.

In the event that the function symbol associated with a given axiom has more than one argument place, the combinatorial effect is even more pronounced; one can easily check that two more cases are developed by each additional argument position.

5. *Basic Procedure for the Extended Class*

The procedure given in this section greatly reduces the combinatorial explosion produced by the reduction process. The improvement is founded on two observations: (1) in most cases, only a small part of the information contained in the generated axioms is of relevance to the validity of the reduced formula; (2) it is frequently possible to determine which information is relevant in advance of its application.

The example formula $F \equiv x \le g(x) \wedge g(x) \le x \supset x = g(g(g(g(x))))$ of Section 4 serves well as an illustration of the basic idea. Suppose we pretend, for a moment, that F is a member of the unextended class, that is, that the terms $g(x)$ and $g(g(g(g(x))))$ are simply integer variables that happen to have fancy names. If we then apply the procedure of Section 3, the expansion into disjunctive form produces the following two ILP's:

$$\{x \le g(x), g(x) \le x, x \le g(g(g(g(x)))) - 1\}$$

and

$$\{x \le g(x), g(x) \le x, g(g(g(g(x)))) \le x - 1\}.$$

Let us focus on the first of these. If this ILP is solved, the following solution (among others) is obtained:

$$x = 0, \quad g(x) = 0, \quad g(g(g(g(x)))) = 1.$$

At this point, the procedure of Section 3 terminates, offering the discovered solution as a counterexample to the formula F.

If we return to the view of F as a formula in the extended theory, however, it can be seen that the above solution is not a legitimate model for $\neg F$. In particular, the substitutivity axioms of equality forbid that x and $g(x)$ be given the same value while $g(g(g(g(x))))$ is given a different value. (Note, incidentally, that this violation occurs in all solutions of the ILP's in question.) The violated substitutivity property is neatly expressed by the following formula.

$$x = g(x) \supset g(x) = g(g(g(g(x)))).$$

If we now assert this formula as a hypothesis of F, the following formula F^* is obtained:

$$\begin{gathered}[x = g(x) \supset g(x) = g(g(g(g(x))))] \\ \supset \\ x \le g(x) \wedge x \ge g(x) \supset x = g(g(g(g(x)))).\end{gathered}$$

If the new formula F^* is viewed as a member of the unextended class and given to the procedure of Section 3, the resulting ILP's are found not to have any integer solutions. The original formula F must, therefore, be valid as a member of the unextended class and hence as a member of the extended class.

Note that, in the case of our example, this approach requires the solution of only seven ILP's (one to provide the illegitimate counterexample and six to decide the augmented formula F^*), as opposed to the 729 required by the reduction method. Part of the improvement is attributable to the omission of unneeded axioms generated in the reduction method. More importantly, the three axioms from that method that *are* relevant,

$$x = g(x) \supset g(x) = g(g(x)),$$
$$g(x) = g(g(x)) \supset g(g(x)) = g(g(g(x))),$$

and

$$g(g(x)) = g(g(g(x))) \supset g(g(g(x))) = g(g(g(g(x)))),$$

are replaced by a single formula:

$$x = g(x) \supset x = g(g(g(g(x)))).$$

This replacement alone accounts for a ninefold reduction in the number of ILP's that must be solved in the example problem.

We now give a detailed description of the procedure:

(1) Using step (1) of the reduction method, all uninterpreted predicate symbols are eliminated from the formula F to be decided
(2) Expressions involving + or * that occur as arguments to uninterpreted function symbols are eliminated through the introduction of new variables. (For example, the formula $x \leq y + f(3z + 5)$ becomes $z' = 3z + 5 \supset x \leq y + f(z')$.) Let F' be the resulting formula
(3) The negation of F' is placed into a disjunctive normal form $G_1 \vee G_2 \vee \cdots \vee G_p$, as described in Section 3. Each G_i is a conjunction of linear inequalities
(4) The G_i's are tested one by one for satisfiability by applying steps (a), (b), and (c) below If none is satisfiable, F is valid.
    (a) The ILP associated with G_i is solved, using either of the methods suggested in Section 3
    (b) If there is no solution, G_i is unsatisfiable
    (c) Otherwise, the discovered solution is examined for violations of substitutivity of equality (in a way described momentarily) If there are no violations, G_i is satisfiable, and the discovered solution provides a counterexample for F If, on the other hand, a violation is found, a formula H that summarizes the violated property of substitutivity is formulated Step (4) is now applied recursively to each of the conjunctions in the disjunctive expansion of $H \wedge G_i$ G_i is satisfiable (in the extended theory) if and only if each of these is satisfiable

We have yet to show how to examine a given solution S for violations of substitutivity and how to generate the associated substitutivity formula. We must also show, of course, that the procedure (step (4) in particular) terminates. For notational convenience, we assume in the explanation that follows that all uninterpreted function symbols appearing in F are monadic; the general case is a straightforward extension.

The technique for detecting violations is founded on the recursive function EQPAIRS defined below. The definition depends on the following conventions. Let S be the discovered integer solution for the G_i whose satisfiability is to be determined, and T the set of terms to which S assigns values. For each term $t \in T$, let $S(t)$ designate the value assigned to t by S. Let U be the set of terms in T together with all of their subterms. (For example, if S is given by $x = 0$, $g(x) = 0$, and $g(g(g(g(x)))) = 1$, then $T = \{x, g(x), g(g(g(g(x))))\}$ and $U = \{x, g(x), g(g(g(g(x)))), g(g(x)), g(g(g(x)))\}$.) Finally, for each term $t \in U$, define the set

$$E_t = \begin{cases} \{t' \in T \mid S(t) = S(t')\} & \text{if } t \in T, \\ \{t\} & \text{if } t \notin T. \end{cases}$$

EQPAIRS is defined as follows:

```
EQPAIRS($t_1$, $t_2$, alreadytried) =
  if $\langle t_1, t_2 \rangle \in$ alreadytried, then return $\varnothing$
  else if $t_1 \in E_{t_2}$, then return $\{\langle t_1, t_2 \rangle\}$
  else if for some function symbol f and terms $u_1$, $u_2$

    (i) $f(u_1) \in E_{t_1}$ and
    (ii) $f(u_2) \in E_{t_2}$ and
    (iii) EQPAIRS($u_1$, $u_2$, alreadytried $\cup \{\langle t_1, t_2 \rangle\}) \neq \varnothing$,

      then return $P_1 \cup P_2 \cup P_3$, where

        $P_1$ = if $t_1 = f(u_1)$ then $\varnothing$ else $\{\langle t_1, f(u_1) \rangle\}$
        $P_2$ = if $t_2 = f(u_2)$ then $\varnothing$ else $\{\langle t_2, f(u_2) \rangle\}$
        $P_3$ = EQPAIRS($u_1$, $u_2$, alreadytried $\cup \{\langle t_1, t_2 \rangle\}$)

      else return $\varnothing$
```

As is evident from the definition, EQPAIRS is a function of three arguments. The first two (t_1 and t_2) are bound to terms in U, and the third (*alreadytried*) is bound to a set of pairs in $U \times U$. The third argument is always bound to the empty set on external calls, and comes into play on internal calls only as a device to prevent infinite recursion. EQPAIRS returns a set of pairs in $T \times T$.

The usefulness of EQPAIRS turns on the following result, stated here without proof:

THEOREM. *If for all $t_1, t_2 \in T$, either $s(t_1) = s(t_2)$ or EQPAIRS(t_1, t_2, $\varnothing$) = $\varnothing$, then S has no violations of substitutivity, and hence G_i is satisfiable as a member of the extended theory. On the other hand, if for some t_1, t_2, $s(t_1) \neq s(t_2)$, and EQPAIRS(t_1, t_2, $\varnothing$) = $\{\langle r_1, s_1 \rangle, \dots, \langle r_n, s_n \rangle\}$, $n \geq 1$, then the formula*

$$H \equiv [r_1 = s_1 \wedge r_2 = s_2 \wedge \cdots \wedge r_n = s_n \supset t_1 = t_2]$$

follows from substitutivity but is not satisfied by S.

To check S for violations of substitutivity, it thus suffices to compute EQPAIRS(t_1, t_2, $\varnothing$) for pairs t_1, t_2 of terms in T assigned different values by S. A violation exists if and only if EQPAIRS(t_1, t_2, $\varnothing$) $\neq \varnothing$ for some such pair. In such a case the formula H summarizes the violation.

Note from the definition of EQPAIRS that, if $t_1, t_2 \in T$, $s(t_1) \neq s(t_2)$, and EQPAIRS(t_1, t_2, $\varnothing$) $\neq \varnothing$, then t_1 and t_2 must be functional terms with the same outermost function symbol. In checking S for violations of substitutivity, therefore, one need only compute EQPAIRS(t_1, t_2, $\varnothing$) for such pairs.

Note also that EQPAIRS is defined nondeterministically; there may be more than one choice of f_1, u_1, and u_2 that satisfies (i), (ii), and (iii) in the definition. It makes no difference which choice is made. Similarly, there may be several pairs t_1, t_2 for which EQPAIRS is nonempty. It suffices to generate H on the basis of the first such pair encountered.

Let us now return to the earlier example:

$$F \equiv x \leq g(x) \wedge g(x) \leq x \supset x = g(g(g(g(x)))).$$

If we apply steps (1), (2), and (3) of the procedure,

$$G_1 \equiv x \leq g(x) \wedge g(x) \leq x \wedge x \leq g(g(g(g(x)))) - 1$$

and

$$G_2 \equiv x \leq g(x) \wedge g(x) \leq x \wedge g(g(g(g(x)))) \leq x - 1$$

are obtained as before. Solving G_1 once again produces the solution S:

$$x = 0, \quad g(x) = 0, \quad g(g(g(g(x)))) = 1.$$

Since $g(x)$, $g(g(g(g(x))))$ form the only pair of terms in T with the same outermost function symbol and with different assigned values, it suffices to compute EQPAIRS for that pair only:

EQPAIRS$(g(x), g(g(g(g(x)))), \varnothing)$

$E_{g(x)} = \{x, g(x)\}$
$E_{g(g(g(g(x))))} = \{g(g(g(g(x))))\}$

Letting $g(x)$ *be* $f(u_1)$, $g(g(g(g(x))))$ *be* $f(u_2)$, *and recursing*:
EQPAIRS$(x, g(g(g(x))), \{\langle g(x), g(g(g(g(x))))\rangle\})$:

$E_x = \{x, g(x)\}$
$E_{g(g(g(x)))} = \{g(g(g(x)))\}$

Letting $g(x)$ *be* $f(u_1)$, $g(g(g(x)))$ *be* $f(u_2)$
EQPAIRS$(x, g(g(x)), \{\langle g(x), g(g(g(g(x))))\rangle, \langle x, g(g(g(x)))\rangle\})$

$E_x = \{x, g(x)\}$
$E_{g(g(x))} = \{g(g(x))\}$

Letting $g(x)$ *be* $f(u_1)$, $g(g(x))$ *be* $f(u_2)$:
EQPAIRS$(x, g(x), \{\langle g(x), g(g(g(g(x))))\rangle, \langle x, g(g(g(x)))\rangle,$
$\langle x, g(g(x))\rangle\})$:
$E_x = \{x, g(x)\}$
$= \{\langle x, g(x)\rangle\}$
$= \{\langle x, g(x)\rangle\}$
$= \{\langle x, g(x)\rangle\}$
$= \{\langle x, g(x)\rangle\}$.

The formula H produced in this case is thus

$$x = g(x) \supset g(x) = g(g(g(g(x)))).$$

The ILP's corresponding to the conjunctions in the disjunctive expansion of $H \wedge G_1$ are all found infeasible; hence, G_1 is unsatisfiable.

In a similar manner, G_2 is found unsatisfiable. The procedure thus halts, reporting that F is valid.

In this last example, only one level of recursion in step (4) was needed. In rare instances, more than one is required. Consider, for example, the following theorem:

$$F \equiv a \le b \le f(a) \le 1 \supset a + b \le 1 \vee b + f(b) \le 1 \vee f(f(b)) \le f(a).$$

Applying steps (1), (2), and (3) of the procedure, one obtains the following single conjunction G:

$$a \le b \wedge b \le f(a) \wedge f(a) \le 1 \wedge 2 \le a + b \wedge 2 \le b + f(b) \wedge f(a) + 1 \le f(f(b)).$$

Solving the corresponding ILP produces as one possibility the following solution S:

$$a = 1, \quad b = 1, \quad f(a) = 1, \quad f(b) = 2, \quad f(f(b)) = 2.$$

The only two pairs of terms for which EQPAIRS need be tried are $f(a)$, $f(b)$, and $f(a)$, $f(f(b))$. Trying $f(a)$, $f(b)$ first, one immediately obtains a violation: EQPAIRS $(f(a), f(b), \varnothing) = \{\langle a, b\rangle\}$. The formula $[a = b \supset f(a) = f(b)] \wedge G$ is now expanded into disjunctive form, giving the three conjunctions

$$a \le b - 1 \wedge G, \quad b \le a - 1 \wedge G, \quad f(a) = f(b) \wedge G.$$

The ILP's associated with the first two of these are infeasible. The third one, however, yields a solution:

$$a = 1, \quad b = 1, \quad f(a) = 1, \quad f(b) = 1, \quad f(f(b)) = 2.$$

Applying EQPAIRS to the pair $f(a)$, $f(f(b))$ produces $\{\langle a, f(b)\rangle\}$. Step (4) is again applied recursively to the three conjunctions in the expansion of $[a = f(b) \supset f(a) = f(f(b))] \wedge f(a) = f(b) \wedge G$. This time, all ILP's are found to be infeasible, and the procedure terminates.

6. *Termination*

It is easy to see, in fact, that the recursive step, (4), must always terminate. Otherwise, some branch of the computation would be infinite, yielding an infinite sequence of ILP's $G, G', G'', \ldots$ with a corresponding sequence of solutions $S, S', S'', \ldots$ and substitutivity formulas $H, H', H'', \ldots$. Since each solution assigns values to exactly the same set T of terms, some H must be repeated in the sequence. This is impossible, however, since each solution S fails to satisfy its corresponding H, but does satisfy all preceding H's.

7. *A More Efficient Version of the Procedure*

Although the procedure given in Section 5 dramatically improves on the naive reduction method, substantial additional improvement is possible.

First note that the expansion of $H \wedge G_i$ into disjunctive form in step (4-c) is unnecessary. The conjunctions that result from this expansion can be precomputed as follows:

$$\begin{aligned} H \wedge G_i &\equiv [r_1 = s_1 \wedge \cdots \wedge r_n = s_n \supset t_1 = t_2] \wedge G_i \\ &\equiv [r_1 \neq s_1 \vee \cdots \vee r_n \neq s_n \vee t_1 = t_2] \wedge G_i \\ &\equiv r_1 \leq s_1 - 1 \wedge G_i \vee s_1 \leq r_1 - 1 \wedge G_i \\ &\quad \vdots \\ &\quad \vee r_n \leq s_n - 1 \wedge G_i \vee s_n \leq r_n - 1 \wedge G_i \\ &\quad \vee t_1 = t_2 \wedge G_i . \end{aligned}$$

Note that $2n + 1$ conjunctions are thus generated, each one augmenting G_i with an inequality. If the ILP solver used can be operated incrementally (as can simplex-based methods), the new ILP's can be solved with little additional effort.

In the great majority of cases encountered in practice, further speedup is possible. Note from the definition of EQPAIRS that $S(r_j) = S(s_j)$ for each j, $1 \leq j \leq n$. Now suppose it can be established, for a given j, that r_j and s_j are equal in *all* solutions for G_i. In this case, the two conjunctions $r_j \leq s_j - 1 \wedge G_i$ and $s_j \leq r_j - 1 \wedge G_i$ are necessarily unsatisfiable and can therefore be dispensed with.

What makes this observation useful is that one can test whether r_j and s_j are equal in all solutions of G_i quite easily; it is necessary only to test (using the ILP solver) for $\max_{G_i}(r_j - s_j) = \min_{G_i}(r_j - s_j) = 0$. This can be done more quickly than testing $r_j \leq s_j - 1 \wedge G_i$ and $s_j \leq r_j - 1$ for feasibility, since it does not involve additional inequalities.

Returning to the earlier example,

$$\begin{aligned} G_1 &\equiv x \leq g(x) \wedge g(x) \leq x \wedge x \leq g(g(g(g(x)))) - 1, \\ H &\equiv x = g(x) \supset g(x) = g(g(g(g(x)))), \end{aligned}$$

we see that x and $g(x)$ must have equal values in all solutions of G_1, and so only the conjunction $g(x) = g(g(g(g(x)))) \wedge G_1$ needs to be tested.

These ideas suggest the following replacement for step (4-c):

(4)(c) The discovered solution S is tested for violations of substitutivity by computing EQPAIRS(t_1, t_2, $\varnothing$) for pairs t_1, t_2, $\in T$ that have different values in S but the same outermost function symbol If EQPAIRS returns $\varnothing$ for all such pairs, the solution S provides a counterexample for F and the procedure halts If t_1, t_2 are found for which EQPAIRS(t_1, t_2, $\varnothing$) $= \{\langle r_1, s_1\rangle, \ldots, \langle r_n, s_n\rangle\}$, $n \geq 1$, then step (4) is applied recursively to

$$t_1 = t_2 \wedge G_i,$$

and for $1 \leq j \leq n$, to

$$r_j \leq s_j - 1 \wedge G_i \quad \text{unless} \quad \max_{G_i}(s_j - r_j) \leq 0$$

and

$$s_j \leq r_j - 1 \wedge G_i \quad \text{unless} \quad \max_{G_i}(r_j - s_j) \leq 0$$

G_i is unsatisfiable if and only if each of the conjunctions thus tested is found unsatisfiable

Finally, it might be remarked that the function EQPAIRS can be implemented much more efficiently than the definition suggests. If, for example, a table is used to record the results of internal calls, the amount of work required to compute EQPAIRS can be made to grow no faster than the square of the length of the input.

ACKNOWLEDGMENTS. The author is grateful to W. Bledsoe, D. Oppen, B. Elspas, and J. Spitzen for helpful discussions about this work.

REFERENCES

(Note References [2, 10, 15] are not cited in the text.)

1 ACKERMAN, W *Solvable Cases of the Decision Problem.* North-Holland Pub Co, Amsterdam, 1954, pp 102–103

2 BLEDSOE, W.W The Sup-Inf method in Presburger arithmetic Memo ATP-18, Math Dept, U of Texas at Austin, Austin, Tex, Dec 1974.

3 BLEDSOE, W W A new method for proving certain Presburger formulas Advance Papers 4th Int Joint Conf on Artif Intell., Tibilisi, Georgia, U S S.R, Sept. 1975, pp 15–21.

4 COOPER, D C Programs for mechanical program verification. In *Mach Intell. 6,* B. Meltzer and D Michie, Eds, American Elsevier, New York, 1971, pp 43–59

5 COOPER, D C Theorem proving in arithmetic without multiplication In *Mach Intell 7,* B Meltzer and D Michie, Eds, American Elsevier, New York, 1972, pp 91–99

6 DOWNEY, P Undecidability of Presburger arithmetic with a single monadic predicate letter Tech Rep 18-72, Center for Research in Computing Technology, Harvard U, Cambridge, Mass, 1972

7 ELSPAS, B, BOYER, R E, SHOSTAK, R, AND SPITZEN, J A verification system for JOVIAL/J3 programs SRI Tech Rep 3756-1, Stanford Research Institute, Menlo Park, Calif, Jan 1976

8. GOMORY, R E An algorithm for integer solutions to linear programs Princeton-IBM Math Res Rep, Nov 1958, also in *Recent Advances in Mathematical Programming,* R.L Graves and P Wolfe, Eds, McGraw-Hill, New York, 1963, pp. 269–302.

9 KREISEL, G, AND KREVINE, J L *Elements of Mathematical Logic* North-Holland Pub Co, Amsterdam, 1967, pp 54–57

10. LEE, R D An application of mathematical logic to the integer linear programming problem *Notre Dame J. Formal Logic XIII,* 2 (April 1972), 279–282

11 MCCARTHY, J. Towards a mathematical science of computation Proc IFIP Congress 62, North-Holland Pub Co, Amsterdam, 1962, pp 21–28.

12 NELSON, G, AND OPPEN, D A simplifier based on efficient decision algorithms Proc Fifth ACM Symp on Prog Langs, Tucson, Ariz, Jan 1978

13 OPPEN, D A $2^{2^{2^{n}}}$ upper bound on the complexity of Presburger arithmetic Ph D. Th, U of Toronto, Toronto, Ont, Canada, 1975, *J. Comptr Syst Sci* (to appear)

14 PRESBURGER, M Uber die Vollstandigkeit eines gewissen Systems der Arithmetik ganzer Zahlen in Welchem die Addition als einzige Operation hervortritt Sprawozdanie z I Kongresu Matematykow Krajow Slowcanskich Warszawa, 1929, Warsaw, Poland, pp. 92–101

15 SHOSTAK, R An algorithm for reasoning about equality Proc Seventh Int Joint Conf on Artif Intell, Cambridge, Mass, Aug 1977

16 SHOSTAK, R. On the SUP-INF method for proving Presburger formulas. *J ACM 24,* 4 (Oct 1977), 529–543.

17 SUZUKI, N Verifying programs by algebraic and logical reduction Proc. Int. Conf on Reliable Software, SIGPLAN Notices (ACM) *10,* 6 (June 1975), 473–481

RECEIVED AUGUST 1977; REVISED MARCH 1978