Verification Decidability of Presburger Array Programs

Norihisa Suzuki & David Jefferson

June 14, 1977

Abstract: A program annotated with inductive assertions is said to be verification decidable if all of the verification conditions generated from the program and assertions are formulas in a decidable theory. We define a theory, which we call Presburger array theory, containing two logical sorts: integer and array-of-integer. Addition, subtraction, and comparisons are permitted for integers. We allow array contents and assign functions, and, since the elements of the arrays are integers, array accesses may be nested. The first result is that the validity of unquantified formulas in Presburger array theory is decidable, yet quantified formulas in general are undecidable. We also show that, with certain restrictions, we can add a new predicate Perm(M,N) -- meaning array M is a permutation of array N -- to the assertion language and still have a solvable decision problem for verification conditions generated from unquantified assertions. The significance of this result is that almost all known sorting programs, when annotated with inductive assertions for proving that the output is a permutation of the input, are verification decidable.

Keywords: data structures, decidability, Presburger array theory, program verification, theorem proving.

To be presented at Conference on Theoretical Computer Science at University of Waterloo, August 1977.

The research described here was supported in part by the Defense Advanced Research Projects Agency (Contracts: F44620-73-C-0074, monitored by the Air Force Office of Scientific Research, and DAHC-15-72-C-0308). The views expressed are those of the authors.

1. Introduction

The theory of program schemata gains power by dealing with classes of programs instead of individual programs. Once we establish some result about a program schema we can apply that result to any program which is an instance of the schema. Unfortunately, for those of us interested in verification, the theory of program schemata has not provided many positive results, and is still unsuccessful in providing tools for proving program correctness.

One reason for this might be that schemata do not divide the class of all programs into the kind of subclasses useful for verification. A correctness proof for one instance of a schema is of almost no use when trying to find a proof of correctness for another instance, simply because the two programs may be working with entirely different data types, functions and predicates. The fact that two programs share the same control structure has almost no verification significance.

We suggest that programs be classified according to the kinds of verification conditions they generate. Since the verification conditions depend on both the program and the inductive assertions, we classify not programs per se, but annotated programs, complete with pre- and post-condition and loop invariant assertions. For example, if a program uses only type integer with +, -, =, and < and if all of its inductive assertions use only +, -, =, and < as well, then all of the verification conditions will be well-formed formulas of Presburger arithmetic. Since the theory of Presburger arithmetic is well-known to be decidable, the weak correctness problem for the entire class of "Presburger arithmetic programs" is decidable.

The advantage of this classification is that most of the variants of programs which implement the same or the similar algorithms can be in one class. The assertions of the programs which implement the similar algorithm are very similar. Thus, one can use the same proof procedure for all the programs which implement the similar algorithms.

Unfortunately not much work has been done exploring the decision problems for weak correctness of classes of programs defined this way. When all of the verification conditions for a class of (annotated) programs fall in a decidable theory, we say that the class is verification decidable. What we will explore here is the verification decidability of certain classes of programs which use arrays. We investigate the theory of arrays of integers with operations restricted to addition and subtraction and call this Presburger array theory. The first result of this paper is in section 3: the validity problem for unquantified well-formed formulas of Presburger array theory is decidable. We conclude from this that the weak correctness problem for programs using integers and arrays of integers and having unquantified assertions is decidable. We also show that since we can encode multiplication by using addition and one dimensional arrays, the theory is undecidable for quantified formulas in general.

There are probably not many interesting array programs whose inductive assertions are expressible in such a weak assertion language . What we would like is an assertion language powerful enough to express interesting assertions about an interesting class of programs, such as sorting programs, but for which the decision problem for the verification conditions generated is solvable.

One way to extend the assertion language is to add new (interpreted) predicate symbols. In section 4 we consider the addition, in a limited way, of a predicate Perm(M,N), meaning array M is a permutation of array N. The perm predicate can be defined by a second-order formula as follows:

$$\mathrm{Perm}(M,N) \equiv (\exists f)[(\forall x,y)(f(x)=f(y) \supset x=y) \wedge (\forall z)(M[z]=N[f(z)])].$$

We show in section 4 that the weak correctness problem for annotated programs using the Perm predicate in assertions (subject to limitations) is decidable. This result is valuable because for almost every known one-array sort program it is the case that the inductive assertions necessary to prove that the output is a permutation of the input can be written easily in the assertion language we permit. Thus, the problem of verifying whether or not a candidate sorting program satisfies the permutation condition is decidable.

2. Notations and Definitions

Presburger arithmetic is the first order theory of integers with addition and no multiplication. The particular characterization we choose has

constants : 0,1
functions symbols : +,-
predicate symbols : =,<.

This theory is known to be decidable [Hilbert].

Presburger array theory, which we denote by L_{PA}, is a two-sorted theory with sort integer and sort array of integer. We use D_I to denote the domain of integers and D_A to denote the domain of array of integers. The language consists of,

constants : constants of Presburger theory;
function symbols : +,-
$<*,*,*> : D_A \times D_I \times D_I \rightarrow D_A$, (array assign)
$*[*] : D_A \times D_I \rightarrow D_I$. (array access)
We used * to denote the location of the arguments for the two functions involving arrays.
predicate symbols: =,<.

Terms of sort integer is defined as follows.
1) The constants and the variables of sort integer are terms of sort integer.

2) If a_1 and a_2 are terms of sort integer, so are a_1+a_2 and a_1-a_2.
3) If A is a term of sort array and i is a term of sort integer, then A[i] is a term of sort integer.
4) These are all the terms of sort integer.

Terms of sort array are defined as follows.

1) Variables of sort array are terms of sort array.
2) If A is a term of sort array, and i and e are terms of sort integer, then <A,i,e> is a term of sort array.
3) These are all the terms of sort array.

Atomic formulas are defined as follows.

1) If a_1 and a_2 are terms of sort integer then $(a_1=a_2)$ and $(a_1<a_2)$ are atomic formulas.
2) These are all the atomic formulas.

Well-formed formulas are defined as follows.

1) Atomic formulas are well-formed formulas.
2) If A and B are well-formed formulas and x is a variable, then $(\neg A)$, $(A \vee B)$, $(A \wedge B)$, $(A \supset B)$, $(A \equiv B)$, $(\exists x.A)$, and $(\forall x.A)$ are all well-formed formulas.
3) These are all the well-formed formulas.

McCarthy [McCarthy] has introduced the notion of states and described the semantics of Algol-like programs. He defined two functions, assign and contents, to change states and obtain values of program variables in the state. He defined these functions by two axioms:

A1. contents(assign(S,x,e),x) = e
A2. contents(assign(S,x,e),y) = contents(S,y)

where x and y are distinct variables.

Kaplan [Kaplan] has shown that these axioms are complete if the only well-formed formulas permitted are equality between terms and if no function symbols are interpreted except assign and contents.

King [King] has used McCarthy's idea to describe effects of assignments on arrays. In his formalism assign(M,i,e) changes the value of the i-th element of array M to e, and contents(M,i) obtains the value of the i-th element of array M. The axioms corresponding to McCarthy's axioms are:

Ax1. $i=j \supset$ contents(assign(M,i,e),j) = e
Ax2. $i\neq j \supset$ contents(assign(M,i,e),j) = contents(M,j).

In this paper we will use more popular notations <M,i,e> and M[i] instead of assign and contents respectively.

Besides the axioms Ax1 and Ax2, we use two other axioms equating the meaning of arrays to functions.

Ax1. $\forall x,y,e,M.(x=y \supset <M,x,e>[y]=e)$.
Ax2. $\forall x,y,e,M(.x\neq y \supset <M,x,e>[y]=M[y])$.
Ax3. $\forall x,y,a,b.\exists M. (M[x]=y \wedge (x\neq y \supset M[y]=b))$.
Ax4. $\forall x,M,N. (M[x]=N[x] \supset M=N)$.

We will denote the above set of axioms by A. In addition we will use the axioms of Presburger arithmetic augmented with equality substitution axioms for any wffs of Presburger array theory. We denote this set by P.

3. Decision Procedure for Presburger Array Theory

In this section we present an algorithm for deciding the truth or falsity of unquantified formulas of Presburger array theory, L_{PA}.

The algorithm is as follows.

Step 1

From the definition of well-formed formulas there is at least one occurrence of a term of the form <M,x,e>[y] if there is at least one occurrence of the array assignment function <M,x,e>. We eliminate this occurrence of the array assignment by the following procedure. Let us denote the formula by R(<M,x,e>[y]), where <M,x,e>[y] indicates the occurrence in question. We transform this formula to

$$[\ x=y \supset R(e)\ ] \wedge [\ x\neq y \supset R(M[y])\ ] .$$

Note that this is still a formula of L_{PA}. It has one fewer occurrences of the assignment function than the original formula. We repeat step 1 until there are no more occurrences of the assignment function.

Step 2 and Step 3 are repeated for each different array.

Step 2

If the formula is of the form $R(M[x_0])$ where x_0 does not contain any occurrence of contents function, we create a new variable a_0 and replace the formula by

$M[x_0]=a_0 \supset R(a_0)$. If there are still occurrences of the contents function in $R(a_0)$ then we apply this transformation again to $R(a_0)$ and iterate. Finally we get a formula of the form

$$M[x_0]=a_0 \supset (M[x_1]=a_1 \supset (...(M[x_n]=a_n \supset R(a_0,...,a_n))...))$$

where $R(a_0,...,a_n)$ does not contain any occurrence of the contents function. This formula is equivalent to

$$(M[x_0]=a_0 \wedge ... \wedge M[x_n]=a_n) \supset R(a_0,...,a_n).$$

Step 3

There are no nested occurrences of the contents function in the formula obtained after step 2. We convert the antecedent part $(M[x_0]=a_0 \wedge ... \wedge M[x_n]=a_n)$ to the formula $Q(n)$ defined below.

$$Q(0) \equiv \text{True}.$$

$$Q(j+1) \equiv Q(j) \wedge [(x_1=x_{j+1} \supset a_1=a_{j+1}) \wedge ... \wedge (x_j=x_{j+1} \supset a_j=a_{j+1})]. \quad (j \geq 0)$$

Thus, we obtain

$$Q(n) \supset R(a_0,...,a_n).$$

Since there is no assignment or contents function and this formula is a formula of Presburger arithmetic we can decide the validity.

end of procedure.

It is obvious that this procedure terminates. In each iteration of step 1 we eliminate one occurrence of <M,x,e>, and in each iteration of step 2 we eliminate one occurrence of $M[x]$. Step 3 terminates because the definition of $Q(n)$ is primitive recursive. What we will prove is that the procedure transforms the formula to an "equivalent" formula.

Theorem

This decision procedure transforms a formula R to an "equivalent" formula R' in the sense that $P\ ,\ A \mid\!- R$ iff $P\ ,\ A \mid\!- R'$.

Proof

1. Transformation by step 1) is correct:

The following is an obvious consequence of Ax1, Ax2, and equality substitution.
$P\ ,\ A \vdash R(<M,x,e>[y]) \equiv (x=y \supset R(e)) \wedge (x \neq y \supset R(M[y]))$.

2. Transformation by step 2) is correct:

We prove that for any formula R
$P\ ,\ A \vdash R(M[x]) \equiv (\forall a.a=M[x] \supset R(a))$.
by the following chain of reasoning,
$(\forall a.a=M[x] \supset R(a)) \equiv (\forall a.a=M[x] \supset R(M[x])) \equiv (\exists a.a=M[x]) \supset R(M[x]) \equiv R(M[x])$.

3. Transformation by step 3) is correct.

We will prove
$P\ ,\ A \vdash (M[x_0]=a_0 \wedge \ldots \wedge M[x_n]=a_n) \supset R(a_0,\ldots,a_n)$ iff $P\ ,\ A \vdash Q(n) \supset R(a_0,\ldots,a_n)$.
Since there is no free occurrence of M in $R(a_0,\ldots,a_n)$,
$P\ ,\ A \vdash (M[x_0]=a_0 \wedge \ldots \wedge M[x_n]=a_n) \supset R(a_0,\ldots,a_n)$
iff
$P\ ,\ A \vdash (\exists M.(M[x_0]=a_0 \wedge \ldots \wedge M[x_n]=a_n)) \supset R(a_0,\ldots,a_n)$.
We now reduce the problem to showing
$P\ ,\ A \vdash (\exists M.(M[x_0]=a_0 \wedge \ldots \wedge M[x_n]=a_n))$ iff $P\ ,\ A \vdash Q(n)$,
which we prove by induction on n.

1) If n=0 the left hand side is
$\exists M.M[x_0]=a_0$.
From Ax3 $P\ ,\ A \vdash \exists M.M[x_0]=a_0$. Since $P\ ,\ A \vdash Q(0)$, the proposition is true for n=0.

2) Assume the proposition is true for n=j,
that is $P\ ,\ A \vdash \exists M.(M[x_0]=a_0 \wedge \ldots \wedge M[x_j]=a_j)$ iff $P\ ,\ A \vdash Q(j)$.

To prove the proposition in the forward direction for n=j+1
we assume $(\exists M.(M[x_0]=a_0 \wedge \ldots \wedge M[x_{j+1}]=a_{j+1})$, which is equivalent to
$\exists M.[(M[x_0]=a_0 \wedge \ldots \wedge M[x_j]=a_j) \wedge M[x_{j+1}]=a_{j+1}]$.
For a new array constant M_0
$(M_0[x_0]=a_0 \wedge \ldots \wedge M_0[x_j]=a_j) \wedge M_0[x_{j+1}]=a_{j+1}$,
is true from the assumption.
Using the inductive hypothesis we can deduce
Q(j).
Also by equality substitution
$x_i=x_{j+1} \supset M_0[x_i]=M_0[x_{j+1}]$.
By equality substitution
$x_i=x_{j+1} \supset a_i=a_{j+1}$ for any i $(1 \leq i \leq j)$.
Thus, $\bigwedge_{1 \leq i \leq j} [x_i=x_{j+1} \supset a_i=a_{j+1}]$.
So we can deduce Q(j+1).

Conversely, to prove the proposition for n=j+1 in the reverse direction, assume Q(j+1) that is $Q(j) \wedge (\bigwedge_{1\le i\le j}[x_i=x_{j+1} \supset a_i=a_{j+1}])$ is true.

By inductive hypothesis $\exists M. \bigwedge_{1\le i\le j} M[x_i]=a_i$ is true.

That is, for a new constant M_0, $\bigwedge_{1\le i\le j}(M_0[x_i]=a_i)$.

By Ax3, for new constants $M_1, \ldots, M_j$

$M_1[x_{j+1}]=a_{j+1} \wedge (x_1 \neq x_{j+1} \supset M_1[x_1]=a_1)$.

...

...

$M_j[x_{j+1}]=a_{j+1} \wedge (x_j \neq x_{j+1} \supset M_j[x_j]=a_j)$.

If $x_1=x_{j+1}$ then $a_1=a_{j+1}$ and thus $M_0[x_1]=M_1[x_{j+1}]$ or

$M_0[x_1]=M_1[x_1]$. Using Ax4 $M_0=M_1$ or $M_0[x_{j+1}]=a_{j+1}$.
Repeating the above step for i=1 to j, $\bigwedge_{1\le i\le j}(x_i=x_j \supset M_0[x_{j+1}]=a_{j+1})$.

On the other hand if $x_1 \neq x_{j+1} \wedge \ldots \wedge x_j \neq x_{j+1}$ then $M_0=M_1 \wedge \ldots \wedge M_0=M_j$

or $x_1 \neq x_{j+1} \wedge \ldots \wedge x_j \neq x_{j+1} \supset M_0[x_{j+1}]=a_{j+1}$.

Thus, $M_0[x_{j+1}]=a_{j+1}$.
So $\exists M. \bigwedge_{1\le i\le j+1}(M[x_i]=a_i)$.

QED

We have shown that unquantified Presburger array formulas are decidable. However, we cannot in general decide the validity of quantified Presburger array formulas. The reason is that we can encode square function by an array as follows:

$M[0]=0 \wedge \forall i. i\ge 0 \supset M[i+1]=M[i]+i+i+1$.

Then the multiplication can be performed as M[a+b]-M[a]-M[b]. With multiplication along with addition we can encode any recursive functions, and the validity problem in this theory becomes unsolvable.

The implication of the verification decidability results is that if the only function symbols the program uses on integer sort expressions are addition and subtraction, and the assertions are written by Presburger array language, then the correctness is decidable.

This is not itself a very strong result. To be able to decide correctness of more interesting programs like sorting programs we have to find finer subclasses of Presburger array theory than is possible by classifying according to prenex normal form quantifier prefixes.

One way is to follow what the people have been doing in practice [Suzuki]. We introduce new predicate symbols to denote certain well-formed formulas and obtain the decision procedure for the limited formulas. The next section deals with such an example.

4. Decision Procedure for Permutation

In this section we consider the problem of deciding whether or not a designated array in some program has a final value which is a permutation of its initial value. Thus, we want a procedure which can prove (or disprove) results of the following form:

$$P \wedge \mathrm{Perm}(M,M_0) \ \{\ \mathrm{program}(M)\ \}\ \mathrm{Perm}(M,M_0)\ .$$

The variable M is assumed to be the array in question, and M_0 is its initial value. The atomic formula $\mathrm{Perm}(M,M_0)$ means that the array M is a permutation of the array M_0. The symbol P stands for other preconditions which do not use the Perm predicate.

More precisely, we consider the class of all programs which use the data sorts integer and array-of-integer. For the integers we allow operations + and -, and predicates = and <. Multiplication and division are excluded, as before, so we can work in the decidable theory of Presburger arithmetic. For our purposes it will be sufficient to consider arrays which are infinite in both directions. The complications which are introduced by arrays with upper and lower bounds are unnecessary for the simple sorting programs which are our targets. The array contents and assign functions are, of course, permitted for arrays, but the array equality predicate is not.

We require inductive assertions to be of the form

$$P \wedge \mathrm{Perm}(M,M_0)$$

where either conjunct may be absent. M may be any array expression, but M_0 must be a simple variable which does not appear anywhere in the program (though it may appear in the Perm conjunct of other assertions.) P may be any unquantified Presburger array formula over the sorts integer and array-of-integer, but it may not contain any occurrence of the Perm predicate. We call this assertion language L_{PA} with Perm.

For programs and inductive assertions of the kind we have described the verification conditions all have the form

(1) $\quad P_1 \wedge \mathrm{Perm}(M,M_0) \supset P_2 \wedge \mathrm{Perm}(M,M_0)$

where P_1 and P_2 are Presburger array formulas, M is an array expression, and M_0 is an array variable which does not appear in M, P_1, or P_2. Most of the remainder of this section is devoted to a decision procedure for formulas of the form (1). Throughout this algorithm we rely heavily on the result of the previous section that unquantified formulas in the language of Presburger array theory, L_{PA}, are decidable.

Before we give the decision procedure, however, we should note that the theory we are developing is applicable to almost all known one-array sorting programs. In each case they confine themselves to the Presburger arithmetic subtheory of the integers. Furthermore, they satisfy the assertion language restrictions we made since loop invariants sufficiently strong to prove the permutation-preserving property of the program can be written very naturally in the assertion language L_{PA} with Perm. In fact, they usually can be written as single Perm atomic formula without the need for the optional Presburger array formula conjunct that we allow. In that sense the result we present is stronger than needed for our target sort programs.

We now proceed with the decision procedure for formulas of the form (1).

Step 1:

Formula (1) can be broken into two smaller formulas, namely

(2) $P_1 \wedge \mathrm{Perm}(M,M_0) \supset P_2$

and

(3) $P_1 \wedge \mathrm{Perm}(M,M_0) \supset \mathrm{Perm}(N,M_0)$.

Clearly formula (1) is TRUE if and only if formulas (2) and (3) are both TRUE.

We can dispose of (2) easily by noting that since M_0 does not occur in P_1 or P_2, the $\mathrm{Perm}(M,M_0)$ conjunct of the hypothesis is irrelevant and can be eliminated. Formula (2) is true if and only if

(4) $P_1 \supset P_2$

is true. Since (4) is in L_{PA}, its truth is decidable. The proof that (2) is equivalent to (4) is quite short.

$P_1 \wedge \mathrm{Perm}(M,M_0) \supset P_2$ | Assumption

$\forall M_0.(P_1 \wedge \mathrm{Perm}(M,M_0) \supset P_2)$ | $\forall$-gen

$P_1 \wedge \exists M_0.(\mathrm{Perm}(M,M_0)) \supset P_2$ | M_0 does not occur free in P_1 or P_2

$P_1 \supset P_2$ | $\exists M_0.(\mathrm{Perm}(M,M_0)) \equiv \mathrm{TRUE}$

Each step in the above transformation is reversible, so (2) is true if and only if (4) is true.

If (4) is false we terminate the decision procedure negatively. If not, we continue to try to prove formula (3).

Step 2:

Because Perm is an equivalence relation, formula (3) is equivalent to

(5) $P_1 \wedge \mathrm{Perm}(M,M_0) \supset \mathrm{Perm}(M,N)$

Once again, because M_0 does not occur free in P_1, M, or N, we can demonstrate, by a proof nearly identical to the one in step 1, that the second conjunct of (5) is irrelevant and (5) is true if and only if

(6) $P_1 \supset \mathrm{Perm}(M,N)$

is true.

In (6) both M and N are terms of array sort, i.e.

$M = <\ldots<V_M,i,e>\ldots>$ and

$N = <\ldots<V_N,j,f>\ldots>$.

Thus, both M and N represent infinite arrays to which at most finite number of changes (assign operations) have been made. Since P_1 is unquantified, it can only constrain the values of a finite number of the elements of M and N. Consequently, the only way that (6) can be true for all assignments of the variables -- in particular for all assignments of V_M and V_N -- is for V_M and V_N to be the same variable.

Thus, if V_M is not the same variable as V_N, terminate negatively.

Explanation about Step 3:

We now come to the heart of the decision procedure. By step 2 we can rewrite (6) as

(7) $P_1 \supset \mathrm{Perm}(<\ldots<V,i,e>\ldots>,<\ldots<V,j,f>\ldots>)$

Formula (7) says that array V, after a certain finite sequence of assign operations, is a permutation of the same (infinite) array V after a different finite sequence of assign operations. Each assign operation can be viewed as the removal of one element from the array V and the insertion of another. We come, then, to the fundamental idea of our decision procedure: if we let I_M be the multiset of elements inserted into the first array by assignments, and D_M be the multiset of elements deleted from it by assignments, and if we let I_N and D_N be defined similarly for the second term, then (7) holds if and only if

(8) $I_M + D_N = I_N + D_M$

is TRUE as a multiset equation with the assumption P_1. (The + stands for multiset union.) More precisely, since I_M, I_N, D_M, and D_N are multisets of terms, we must show that (8) is true for all assignments of the variables for which P_1 is TRUE, i.e.

(9) $P_1 \supset I_M + D_N = I_N + D_M$.

At this point we are in a position to conclude that formulas of the form of (1) are decidable. We have reduced it to the problem of deciding the truth of formulas of the form (9). Since in (9) the multisets in the consequence are finite and explicitely listed, we can express the equation as a finite set of disjuncts of conjuncts. For example, the following formula in the form of (9)

$$P \supset \{a,b,c\} = \{d,e,f\}$$

can be expressed less tersely as

$$\begin{aligned} P \supset\ & (a=d \wedge b=e \wedge c=f) \vee \\ & (a=e \wedge b=d \wedge c=f) \vee \\ & \quad\vdots \\ & (a=f \wedge b=e \wedge c=d) \end{aligned}$$

with six disjuncts. In general, there will be n! disjuncts if the multisets contain n expressions each. The resulting formula is in L_{PA}, and therefore decidable. But using the decision procedure for L_{PA} directly in this way would be intolerably slow in most cases, and therefore we propose a more practical continuation of the decision procedure in step 3.

Step 3:

We begin by computing I_M, I_N, D_M, and D_N using the following

symbolic algorithm.

```
begin
    multiset of integer expression : $I_M, I_N, D_M, D_N$;
    array expression : M, N, MM, NN, X;
    integer expression : i,e;

    $I_M \leftarrow I_N \leftarrow D_M \leftarrow D_N \leftarrow \{\}$;

    $MM \leftarrow M$ ; $NN \leftarrow N$ ;

    do
        $MM \sim \langle X,i,e \rangle \rightarrow D_M \leftarrow D_M + \{X[i]\}; I_M \leftarrow I_M + \{e\}; MM \leftarrow X$        ||
        $NN \sim \langle X,i,e \rangle \rightarrow D_N \leftarrow D_N + \{X[i]\}; I_N \leftarrow I_N + \{e\}; NN \leftarrow X$
    od
end
```

The first four lines of the algorithm are declarations indicating the types of expressions the variables may take as values. The do -- od construct is Dijkstra's nondeterministic repetitive guarded command construct [Dijkstra]. The ~ sign is a pattern match operator which can be read "is of the form". It returns true or false according to whether or not the match succeeds, and has the side-effect of binding the variables in the right-hand argument whenever the match succeeds.

Having computed I_M, I_N, D_M and D_N, we need to prove (9). We can do this if we have an algorithm for proving

(10) $P \supset S_1 = S_2$

where P is a Presburger array formula and S_1 and S_2 are multisets of integer expressions. We propose to find pairs of elements $e_1 \in S_1$ and $e_2 \in S_2$ such that

$$P \supset e_1 = e_2 .$$

Whenever we find such a pair of equal expressions we remove them from the multisets and continue with the smaller multisets, attempting to show

$$P \supset S_1 - \{e_1\} = S_2 - \{e_2\} .$$

The following iteration will remove pairs of equal elements from S_1 and S_2:

```
do
    $x \in S_1 \wedge y \in S_2 \wedge [P \supset x=y] \rightarrow S_1 \leftarrow S_1 - \{x\};\ S_2 \leftarrow S_2 - \{y\}$
od
```

Once again we have used Dijkstra's iterative guarded command construct. The guard is intended to be a rather elaborate pattern match operation which means "find $x \in S_1$ and $y \in S_2$ such that '$P \supset x=y$' is true". If the pattern match succeeds, the variables x and y are bound to the matching elements, the action to the right of the arrow is executed, and the iteration continues. If the pattern match fails, the iteration terminates.

It might seem that writing the loop the way we did makes the algorithm obscure. We could as well have written the following doubly nested loop.

```
for all $x \in S_1$ do
    for all $y \in S_2$ do
        if $[P \supset x=y]$
            then
                ( $S_1 \leftarrow S_1 - \{x\}$;
                  $S_2 \leftarrow S_2 - \{y\}$
                ).
```

However, we felt that the explicit double loop structure precludes opportunities for optimization which could be important in an actual implementation.

If this iteration succeeds in reducing S_1 and S_2 to empty, then formula (10), and hence formula (1), are TRUE, and the decision procedure terminates. However, if S_1 and S_2 are not reduced to empty, it is not necessarily the case that (10) is FALSE, as we explain in the next step.

Step 4:

Once all such pairs of elements which are equal as a consequence of P have been removed, the remaining multisets still may or may not be equal. It may happen that under one assignment of values to the program variables the multiset elements are pairwise equal according to one correspondence, and under another assignment the elements are pairwise equal under a different correspondence. There might be no two elements which are pairable under all assignments. Probably the simplest example of this phenomenon is the following:

$$\mathrm{TRUE} \supset \{V[i], <V,i,1>[j]\} = \{V[j], <V,j,1>[i]\}\ .$$

In any assignment in which $i=j$ holds, the multisets are equal because their first elements are equal and their second elements are equal. And in any assignment in which $i \neq j$ holds, the multisets are also equal, but the elements are paired according to the other correspondence. Thus, the multisets are equal under all assignments, but there is no pair of elements which are equal under all assignments.

In order to decide the truth of

(11) $P \supset S_1 = S_2$

we rewrite the formula as

(12) $P \supset \{ m_i \mid i \leq k \} = \{ n_i \mid i \leq k \}$

Formula (12) is equivalent to

(13) $P \wedge (m_1=n_1 \vee m_1=n_2 \vee \dots \vee m_1=n_k \vee (m_1 \neq n_1 \wedge m_1 \neq n_2 \wedge \dots \wedge m_1 \neq n_k))$
$\supset \{ m_i \mid i \leq k \} = \{ n_i \mid i \leq k \}$.

Formula (13) can be broken up into smaller formulas such that (13) is true if and only if all of the following k+1 formulas are true.

(14) $P \wedge m_1=n_1 \supset \{ m_i \mid i \leq k \} = \{ n_i \mid i \leq k\}$
...
$P \wedge m_1=n_k \supset \{ m_i \mid i \leq k \} = \{ n_i \mid i \leq k \}$

$P \wedge (m_1 \neq n_1 \wedge m_1 \neq n_2 \wedge \dots \wedge m_1 \neq n_k) \supset \{ m_i \mid i \leq k \} = \{ n_i \mid i \leq k \}$

Each of the formulas of (14) can in turn be further simplified as follows:

(15) $P \wedge m_1=n_1 \supset \{ m_i \mid 2 \leq i \leq k \} = \{ n_i \mid i \leq k \wedge i \neq 1 \}$

$$\overset{\dots}{P \wedge m_i{=}n_k} \supset \{\, m_i \mid 2 \le i \le k \,\} = \{\, n_i \mid i \le k \wedge i \ne k \,\}$$

(15*) $P \wedge (m_1{\ne}n_1 \wedge m_1{\ne}n_2 \wedge \dots \wedge m_1{\ne}n_k) \supset \text{FALSE}$

We abbreviate (15*) as

(16*) $P \supset (m_1 \in S_2)$

In (15) all of the multisets are smaller by one than those in (11). This fact forms the basis of a recursive procedure for proving (or disproving) formula (11). We define a recursive procedure TESTEQUAL(P,S_1,S_2) which returns TRUE or FALSE according to the truth of (12). The multisets S_1 and S_2 must have the same number of expressions in them. TESTEQUAL works in four steps.

(i) If $S_1{=}S_2{=}\{\}$, the procedure returns TRUE immediately.

(ii) As an optimization the procedure then checks that P is satisfiable. If P is unsatisfiable, TESTEQUAL returns TRUE immediately.

(iii) Choose an element m_1 from S_1 and break (11) into the first k-cases of (15). Call TESTEQUAL recursively to test them. The recursive calls must all return true, or TESTEQUAL returns false.

(iv) Test (16*) and return its truth value.

Here is the body of the procedure in an Algol-like syntax.

```
Boolean procedure TESTEQUAL($P,S_1,S_2$);
begin
    formula: P,R;
    multiset of integer expressions: $S_1,S_2$;
    integer expressions: x,y;

    Comment: $S_1$ and $S_2$ must have equal cardinality;
    if $|S_1| \ne |S_2|$ then abort;

    Comment: Null multisets are always equal;
    if $S_1=\{\}$ then return TRUE;

    Comment: Check that P is satisfiable;
    if $[\neg P]$ then return TRUE;

    Comment: Check all cases of (15) except (15*);
    $x \leftarrow$ choice($S_1$);
    for all $y \in S_2$ do
        if not TESTEQUAL($P \wedge x=y$ , $S_1 - \{x\}$ , $S_2 - \{y\}$ ) then return FALSE;
        Comment: Return the result of (16*);
        return [$P \supset (x \in S_2)$]
end.
```

Again we use [P]-notation as a Boolean expression meaning "P is true". Of course we only use this notation when P is a formula in a theory known to be decidable. The choice function merely returns some (random) element of its multiset argument. We have been rather loose with some of the syntax in this program, but we trust that the reader will be able to supply the missing interpretations from the discussion above.

A few additional remarks should be made about the decision procedure we have described above. Our experience indicates that for the kinds of programs people actually write, step 4 of our decision procedure is unnecessary; if the verification condition is in fact a theorem, this is established by step 3. Therefore, if this decision procedure is embedded in a real verifier, it might be wise to issue a warning message to the user before (or instead of) proceeding to step 4, since the worst case complexity of the TESTEQUAL is at least n! in the size of the multiset arguments.

We have only treated the case that the values of the array elements are integer. However, the procedure can be adapted for arrays of reals if the allowed operations on reals are within Tarskian arithmetic [Tarski]. As a matter of fact, the decision procedure can be adapted for any data type in which the equality among terms is decidable.

We observe the procedure for reals. The formula we are going to deal with has the form

$$P \wedge \mathrm{Perm}(A,A0) \supset \mathrm{Perm}(B,A0)$$

where A and B are terms of sort array of reals and P is an unquantified well-formed formula of the two-sorted theory of integers and reals. The restriction here is that we do not allow any mixed sort terms or atomic formulas, so that you cannot equate or add terms of integer and real. Because of this restriction one cannot use a real term to be the index into an array.

The procedure described in this section can be carried out without modification, except where we have to test the truth of particular unquantified formulas. In such cases we can apply the procedure of the previous section to eliminate arrays. Then we can transform the formula to conjunctive normal form. All of the conjuncts have to be valid. Each conjunct consists of disjunction of atomic formulas and we can split these atomic formulas into two classes, one for integer and the other for real. The validity of both disjunctions are independent, and we can use the separate decision procedures for integer and real. Thus, we can decide the permutation property of arrays of real with the same basic algorithm.

Example

The following is an insertion sort program. We can show that the final array is the permutation of the initial array by the decision procedures given in this paper.

The annotated program is

```
assert Perm(A,$A_0$);
J←2;
invariant Perm(A,$A_0$)
while J≤N do
        begin
        KEY←A[J];
        I←J-1;
up:      assert Perm(<A,I+1,KEY>,$A_0$);
        if A[I]≤KEY then goto exit;
        A[I+1]←A[I];
        I←I-1;
        if I≥1 then goto up;
exit:   A[I+1]←KEY;
        J←J+1
        end;
assert Perm(A,$A_0$);
```

Since this program conforms to the restrictions of the Presburger array programs with Perm(A,A_0) assertions, its correctness is decidable.

The verification conditions are

1) $\mathrm{Perm}(A,A_0) \supset \mathrm{Perm}(A,A_0)$

2) $\mathrm{Perm}(A,A_0) \wedge J \le N \supset \mathrm{Perm}(<A,J,A[J]>,A_0)$

3) $\mathrm{Perm}(A,A_0) \wedge \neg\, J \le N \supset \mathrm{Perm}(A,A_0)$

4) $A[I] \le KEY \wedge \mathrm{Perm}(<A,I+1,KEY>,A_0) \supset \mathrm{Perm}(<A,I+1,KEY>,A_0)$

5) $1 \le I-1 \wedge \neg A[I] \le KEY \wedge \mathrm{Perm}(<A,I+1,KEY>,A_0) \supset \mathrm{Perm}(<<A,I+1,A[I]>,I,KEY>,A_0)$

6) $A[I] \le KEY \wedge \mathrm{Perm}(<A,I+1,KEY>,A_0) \supset \mathrm{Perm}(<A,I+1,KEY>,A_0)$

7) $\neg 1 \le I-1 \wedge \neg A[I] \le KEY \wedge \mathrm{Perm}(<A,I+1,KEY>,A_0) \supset \mathrm{Perm}(<<A,I+1,A[I]>,I,KEY>,A_0)$.

The non-trivial verification conditions are 5) and 7), which are very similar. Let us examine 5).

VC 5: $1 \le I-1 \wedge \neg A[I] \le KEY \wedge \mathrm{Perm}(<A,I+1,KEY>,A_0) \supset \mathrm{Perm}(<<A,I+1,A[I]>,I,KEY>,A_0)$

Step 1 Transform to

$1 \le I-1 \wedge \neg A[I] \le KEY \supset Perm(<A,I+1,KEY>,<<A,I+1,A[I]>,I,KEY>)$

Step 2 The base array variables of the two array terms are the same; proceed.

Step 3

$I_M = \{KEY\}$
$D_M = \{A[I+1]\}$
$I_N = \{A[I], KEY\}$
$D_M = \{A[I+1], <A,I+1,A[I]>[I]\}$.

Transform to $P \supset I_M+D_N=I_N+D_M$ form, i.e.

$$1 \le I-1 \wedge \neg A[I] \le KEY \supset \{KEY, A[I+1], <A,I+1,A[I]>[I]\}=\{A[I], KEY, A[I+1]\}.$$

By inspection we can see that the two multisets would be reduced to empty by Step 3, because

$1 \le I-1 \wedge \neg A[I] \le KEY \supset KEY=KEY$
$1 \le I-1 \wedge \neg A[I] \le KEY \supset A[I+1]=A[I+1]$
$1 \le I-1 \wedge \neg A[I] \le KEY \supset <A,I+1,A[I]>[I]=A[I]$

Step 4 Unnecessary, because step 3 reduced the multisets to null.

5. Conclusion

Unlike the decidability results for program schemata, verification decidability is not influenced by the control structure of programs. That is, the decidability results are not sensitive to individual programming style or to variations in algorithms for the same task.

Our permutation decidability results can be applied to almost all of the sorting programs people usually write. We therefore feel that the methods developed in this paper shows the value of having domain specific, specially interpreted predicates such as Perm in the assertion language. Had we not used the Perm predicate as we did, we might have had to write a second-order formula to express the same thing, such as the following:

$$Perm(M,N) = (\exists F)(\forall x)(\forall y)[\ F(x) = F(y) \supset x = y \wedge M[x] = N[F(x)]\ ]$$

It seems very unlikely that verification conditions allowing this kind of quantification over functions will be decidable.

The next target of our research will be the orderedness properties of Presburger arrays. Eventually we hope to find a single assertion language in which the inductive assertions for both the orderedness and permutation properties of ordinary sort programs can be expressed, and for which we can find an algorithm to decide the resulting verification conditions.

There are various other directions that future research in this area might take. For each algorithm domain we should try to establish assertion vocabularies for which the resulting verification conditions are decidable. When decision procedures are discovered, they should be formulated in such a way that they can provide useful debugging information when a proof fails. And, of course, a long range goal is to build a verifier which can recognize programs of the decidable domains, and verify them without human aid.

Bibliography

[Dijkstra] Dijkstra, E.W., A Discipline of Programming, Prentice-Hall, 1976.

[Hilbert] Hilbert, D. and Bernays, P., Grundlagen der Mathematik I, Springer-Verlag, 1968.

[Kaplan] Kaplan, D., Some Completeness Results in the Mathematical Theory of Computation, JACM, Vol.15, No.1, pp.124-134, 1968.

[King] King, J. C., A Program Verifier, Ph.D. thesis, Carnegie-Mellon University, 1969.

[Knuth] Knuth, D.E., The Art of Computer Programming, Vol.2, Addison-Wesley.

[McCarthy] McCarthy, J., Towards a Mathematical Science of Computation, Proc. of IFIP Congress 62, pp.21-28, North-Holland Publishing Company, Amsterdam, 1962.

[Suzuki] Suzuki, N., Verifying Programs by Algebraic and Logical Reduction, Proc. Intl. Conf. on Reliable Software, Sigplan Notices, Vol.10, No.6, June, 1975.

[Tarski] Tarski, A., A Decision Method for Elementary Algebra and Geometry, RAND Corporation, Santa Monica, Ca., 1948.