SOME CLASSES OF NATURALLY PROVABLE PROGRAMS

S. K. Basu and J. Misra
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

Keywords

Program Verification, proving programs correct, inductive assertions, proof rule, uniform loop programs, stack schema, for programs.

Abstract

Three different classes of programs are identified for which the proof of correctness is shown to be "natural", in that the functional input-output specifications of the programs lead, in a straightforward manner, to the verification conditions that should be proven. Furthermore, these verification conditions are shown to be necessary and sufficient so that a proof/refutation follows by proving/disproving the corresponding verification conditions. It is not necessary to follow the exact control flow of the programs to generate these conditions; certain simple checks are enough to show whether a particular program belongs to one of the classes. These apparently different programs have the common feature that they operate "uniformly" on the data domain; changing the input to the program changes the dynamic behavior of the program in a predictable, easily definable fashion. Implications of this feature in program construction are discussed.

1. Introduction

One of the major problems encountered in using the inductive assertion method [4] of program verification is the generation of appropriate assertions. In order to be able to apply the inductive assertion or the Floyd method to a particular program, the programmer has to provide in addition to the input-output specifications of the program, a set of assertions at specific points in the program, which must include at least one cutpoint for each loop. Since in a loop the control may reach the same point several times with altered sets of values of variables, the assertions attached to a loop must capture the invariant properties of the loop iteration. These assertions furthermore, should be strong enough to imply the desired output condition on termination. We say that an assertion P is a loop invariant for a loop of the form <u>while</u> B <u>do</u> S if $P \wedge B \{S\} P$, using the notation introduced in [6].

The main objective of this paper is to consider the problem of deterministically generating loop invariants in order to be able to prove or disprove certain properties of the loop. Several heuristic methods for generation of the loop invariant appear in the literature [5,8,14]. These methods attempt to generate a loop invariant given a loop and its input-output specifications. While these methods are useful in a number of cases, in general, they provide little insight into the relationship between the computations of the loop (the function it is computing) and the invariance preserved by it.

We have attempted to identify certain classes of programs for which the loop invariant can be generated directly or deterministically from the given input-output specifications. Clearly our approach does not cover all possible loop programs. However, a large number of "naturally occurring programs" fall into one of our classes. Basically we show that if the program under consideration is "well behaved" relative to the data domain it operates on, such that changes in the input parameters of interest causes the dynamic program behavior to change in a predictable fashion, then the available partial information about the input output behavior of the program can be deterministically extended to create a loop invariant. In using the word deterministic, we do not intend to imply decidability. In fact, the basic properties of interest are almost always undecidable. However, in case the program under consideration possesses certain specified properties, we show how the desired necessary and sufficient invariance relation can be obtained systematically.

The output assertions we consider state that the program computes a given function over a certain domain. In a certain sense, this is the strongest assertion one can make about the program, and all other weaker properties are derivable from these output assertions. However, it is possible to extend these results to certain classes of relations at output [12].

The research reported here was motivated by attempts to obtain a relationship between computation and invariance properties of loop. We have found that in the process of loop computation, if the loop is constrained to access|alter data in a "uniform" fashion then the loop invariant may be obtained quite systematically. Our primary goal is to develop a set of reasonable rules by which loop invariants may be discovered fairly systematically by programmers.

In section 2, we introduce basic concepts and briefly review some previous results. We formulate the problem precisely and discuss several possible systematic approaches and their trade offs. Section 3 is devoted to accumulating do-while programs. We introduce the notion of independent variables and provide a theorem about the loop invariant for these programs. Several examples are used to illustrate these ideas. Stack programs are investigated in section 4. These programs naturally occur in implementing recursive procedures by iterative structure. We exploit certain specific properties of such programs to obtain a loop invariant. This method is applied to a well known program for preorder traversal of a tree.

In section 5, we study uniform FOR programs, and show that a simple form of induction can be used to prove correctness of such programs. We discuss some extensions and limitations of this approach.

Finally, we discuss the implications of these results, particularly as a methodology for program construction.

2. Loop Programs

Let W(B,S) be an abbreviation for "while B do S". Let W(B,S) accept its input in (a vector of) variables X and produce its output in the same variables. Let S(X) denote the values obtained by executing S with input X.

Let the possible set of input values of X, be drawn from a domain D. We say that W(B,S) computes a function F over D if for any input X_o from D, the output is $X_f = F(X_o)$, (assuming that the program terminates for every input from D).

We define D to be closed with respect to W(B,S) if $X \in D \wedge B(X) \Rightarrow S(X) \in D$.

In other words, D is closed with respect to W(B,S) if and only if $X \in D$ is a loop invariant.

Let P be a loop invariant of W(B,S) and let propositions Q_1, Q_2 denote the input/output specifications of W(B,S). We assume these are predicates on the variables X and their initial values. P is a sufficient loop invariant with respect to Q_1, Q_2 if P is a loop invariant and $Q_1 \Rightarrow P$ and $P \wedge \neg B \Rightarrow Q_2$. P is a necessary loop invariant with respect to Q_1, Q_2 if the falsity of either of the above propositions means that W(B,S) is incorrect with respect to Q_1, Q_2.

In [1,13] we investigated the problem of obtaining a necessary and sufficient loop invariant for W(B,S) with Q_1: $[X \in D]$ and Q_2: $[X_f = F(X_o)]$. We showed that if D is closed, such an invariant is easily obtained. This is the content of the following theorem.

Theorem 1. Given that D is closed with respect to W(B,S), W(B,S) computes F over D if and only if

(1) W(B,S) terminates for every $X \in D$.
and (2) $[B(X) \Rightarrow F(X) = F(S(X))] \wedge [\neg B(X) \Rightarrow F(X)=X]$ for every $X \in D$. $\boxed{X}$

Condition (2) can be broken up into
(i) $[B(X) \Rightarrow [F(X) = F(S(X))]]$
and (ii) $[\neg B(X) \Rightarrow [F(X) = X]]$
(i) says that for an input X when B is false, F(X) = X, if W(B,S) computes F. (This follows trivially.) (ii) says that F(X) remains an invariant quantity through successive iterations with modified values of X. Note that F(X) is well defined since $X \in D$, due to the closure property.

Thus by theorem 1, condition (2) is necessary and sufficient for proving that W(B,S) computes F.

We illustrate this theorem by an example:

Example 1. Consider the following program for computing exponential. Initial value of w is 1. Finally w is set to $u_o^{v_o}$ (u_o, v_o being the initial values of u,v).

```
while v ≠ 0 do
    if (odd v) then w ← w * u;
    v ← v/2; u ← u * u
enddo;
```

We would like to prove/disprove that the program computes $F\langle u, v, w\rangle = \langle \lambda, 0, w * u^v\rangle$ over a domain $D = \{\langle u, v, w\rangle | u, v, w \text{ integer}, v \geq 0\}$ where λ denotes a component value that is of no interest to us at output.

$X : \{u, v, w\}$.
$B(X) : v \neq 0$.
$S(X) : \{$if (odd v) then $w \leftarrow w * u$; $v \leftarrow v/2$; $u \leftarrow u * u$.$\}$

We prove closure of D by proving that $(v \geq 0 \wedge v \neq 0) \Rightarrow v/2 \geq 0$. Then it is necessary and sufficient to prove that

(1) the loop terminates for every $\langle u, v, w\rangle \in D$.
and
(2) $v \neq 0 \Rightarrow [w * u^v = \begin{cases} w * u * (u^2)^{v/2} & \text{when } v \text{ is odd} \\ w * (u^2)^{v/2} & \text{when } v \text{ is even} \end{cases}]$
$\wedge\ v = 0 \Rightarrow (w * u^v = w)$. $\boxed{X}$

Definition - A loop constant is an expression whose value remains unchanged after every iteration of the loop.

A loop invariant is thus a loop constant. Using theorem 1, F(X) is a loop constant. This notion helps us to avoid the use of X_o in statements of theorems.

Usually however the program will not appear in the form given above, but will be preceded by an initialization statement such as w:= 1. In this case our knowledge of the input output behavior of the program is the following.

The input domain of the program is $D' = \{\langle u, v, w\rangle \mid w = 1, v \geq 0, u, v, w \text{ integer}\}$ and we wish to prove/disprove that the program computes the function $F'\langle u, v, 1\rangle = \langle \lambda, 0, u^v\rangle$.

However it is easy to see that D' is not closed with respect to W(B,S) since the intermediate values of w are not in D' and thus theorem 1 becomes inapplicable. The basic problem then is that given D' and F', we need to find a closed domain D and a function F on D which are extensions of D', F'. Then theorem 1 can be applied. Furthermore if D, F are properly chosen extensions of D', F' then a proof/refutation that W(B,S) computes F over D can be used as a proof/refutation that W(B,S) computes F' over D'.

In order to facilitate discovering F, we need to choose D carefully. There are two general approaches for computing D which are quite useful in finding F. One is to identify the various values of global variables that occur in different iterations, starting with an initial input. Once we have the description of such a D in closed form, F is usually easy to guess. The second approach is to consider all possible values of inputs (with suitable restrictions, such as all positive integers, etc.). As is to be expected, F is usually more difficult to guess in such a case. The following example illustrates a situation where finding a closed D almost solves the problem of finding F.

Example 2. Factorial [London]

```
begin  integer s, v;
    while $r < n$ do
        $s \leftarrow 1$; $v \leftarrow u$;
        while  $s \neq r$ do
            $u \leftarrow u + v$; $s \leftarrow s + 1$
        enddo;
        $r \leftarrow r + 1$
    enddo
end;
```

We are required to prove that over the input domain $\{\langle r, u, n\rangle \mid r = 1, u = 1, n \geq 1\}$ the function computed is $F\langle r, u, n\rangle = \langle n, n!, n\rangle$. Using the first technique, we may identify the values of $\langle r, u, n\rangle$ that occur in different iterations to obtain the closed domain $D' = \{\langle r, u, n\rangle \mid 1 \leq r \leq n; u = r!\}$. In fact locating D' and proving closure with respect to D' amounts to a proof of the original conjecture, since $X \in D' \wedge \neg B(X) \equiv r \geq n \wedge u = r! \wedge 1 \leq r \leq n \Rightarrow u = n!$. $\boxtimes$

As the above example illustrates, finding a closed domain D by identifying the intermediate values of variables can be quite difficult. It seems plausible that heuristic methods can be developed which attempt to discover D.

We next investigate three classes of programs for which the desired information can be obtained directly from the output specifications. The conditions imposed on these classes of programs ensure that the dynamic behavior (the execution sequence or components thereof) are well behaved relative to certain components of their data domains. Roughly speaking, we are constraining the dynamic program behavior in order to obtain programs that are more amenable to systematic verification techniques.

3. Accumulating Loop Programs

This class of programs have the following form.

P1:

```
begin
    $z \leftarrow z_0$;
    W(B,S)
end;
```

z is a variable whose value is being computed by the loop; initial value of z is z_0. There are many commonly occurring programs that fit into this form. Usually z "accumulates" the result and the loop is used to step through (and process) a set of elements. Such a schema can be used to find the sum or the maximum of a sequence. Following examples illustrate a few more such programs.

Example 3.

```
(i)   $w \leftarrow 1$;
      while  $v \neq 0$ do
          if (odd v) then $w \leftarrow w * u$;
          $v \leftarrow v/2$; $u \leftarrow u * u$
      enddo;
```

w plays the role of z in this program for computing $w = u_0^{v_0}$, for any $\langle u_0, v_0\rangle \in \{\langle u,v\rangle \mid v \geq 0;\ u,\ v \text{ integer}\}$.

```
(ii)  found $\leftarrow$ false;
      while  $T \neq$ nil do
              found $\leftarrow$ found or (root(T) = key);
              if key $<$ root(T) then $T \leftarrow$ left (T)
                                  else $T \leftarrow$ right(T)
              endif
      enddo;
```

found plays the role of z. This program gives an (inefficient) method for finding out if key is at a node in a given binary search tree (where every node in the left/right subtree of a node are smaller/greater in magnitude); nil denotes a null tree; root (T) gives the element at the root of the tree T; left/right returns left/right subtree. $\boxtimes$

Let (X,z) be all the global variables of W(B,S). We assume that at the end of the program, only the value of z is of interest. We could absorb the initial value of z into the domain D of W(B,S) to get $D = \{\langle X, z\rangle \mid z = z_0;\ X \text{ from the specified domain}\}$. Clearly D is not closed since z usually changes its value. Hence we can't apply theorem 1.

One important aspect of such a schema is that z usually never occurs in a conditional statement inside the loop to govern branching. Thus z is truly an "accumulator"; its specific value does not alter the course of computation. Secondly, assignments to z inside the loop are usually of the form $z \leftarrow z \oplus g(X)$, where $\oplus$ is some binary operator and g is some function on the variables X.

We will prove that under certain conditions on $\oplus$ and z_0, it is possible to specify the loop invariant, if it is required to show that the final value of z is a certain function of the initial values of X.

A variable r is <u>independent</u> of variable t (in a given program) if a change in the input value of t never results in a change in the value of r at any step of computation. r is <u>dependent</u> on t otherwise. Formal conditions on the schema:

1. All global variables X are independent of z.
2. Neither z nor any variable dependent on z is used to govern a conditional branch inside the loop.
3. Every assignment to z inside the loop can be written as $z \leftarrow z \oplus g(X)$ where $\oplus$ is an <u>associative</u> operator that is identical for every assignment to z; g is some function on variables X that may be different for different assignments.
4. z_0 is a (left and right) unity of $\oplus$.

Note: If $\oplus$ has a left and right unity, they are identical and unique.

5. Domain D_1 of the global variables X is closed with respect to W(B,S); i.e. values of X obtained in different iterations belong to the input domain.

Conditions (1) and (2) imply that z is truly an accumulating variable: it does not govern the flow of control nor does it affect the values of other global variables. Conditions (3) and (4) make sure that modification of z is "uniform". Finally condition (5), is a restatement of the closure condition with respect to other variables.

Usually these conditions are verified easily for any given program, such as the ones in example 3 (with reasonable interpretation of the domains).

Theorem 2. Let P1 satisfy conditons 1,2,3,4,5. Let X_0 denote the initial values of X and X_f, z_f denote the final values of X, z (assuming that W(B,S) terminates for every X from D_1). Let h denote some function over D_1. Then, $z_f = h(X_0)$ if and only if

(1) $z \oplus h(X)$ is a loop constant (or equivalently $z \oplus h(X) = h(X_0)$ is a loop invariant).

(2) $[\neg B(X) \wedge X \in D_1] \Rightarrow [h(X) = z_0]$ $\boxtimes$

We give an informal proof below. First, we apply the theorem to the programs in example 3.

Example 3. (continued)

(i) Prove that $w_f = u_o^{v_o}$. We need to show that

1. $w * u^v = u_o^{v_o}$, is a loop invariant.
2. $[v = 0] \Rightarrow [u^v = 1]$.

(ii) To prove that $\text{found}_f = \text{search}(T,\text{key})$;

where search is a function that returns 'true' if and only if key is in the binary search tree T, we need to prove that

1. found <u>or</u> Search(T,key) = Search (T_o,key)
2. (T = nil) $\Rightarrow$ [Search(T,key) = false] $\boxed{X}$

Informal proof of the theorem:

It is obvious that $D = \left\{\langle X,z\rangle \mid X \in D_1,\ z = z_o\right\}$ is not closed with respect to W(B,S). However we know that for any $\langle X,z\rangle$ from D, we have $z_f = h(X_o)$ at the end of the program. We next try to find out the output of W(B,S) when inputs are from $D' = \left\{\langle X,z\rangle \mid X \in D_1\right\}$. Clearly D' is closed. We claim that the output $z_f = z' \oplus h(X')$, where X',z' are the initial values of X, z (from D').

Starting with z_o, X' as input, the value z_f could be written as follows:

$z_f = h(X') = z_o \oplus \{\text{terms dealing with } X'\}$, using associativity of $\oplus$. If we start with z', since computation sequence is identical, and X values don't depend on z,

$z_f = z' \oplus \{\text{same terms dealing with } X'\}$

$= z' \oplus z_o \oplus \{\ "\ \}$, since z_o is a left unity of $\oplus$.

$= z' \oplus h(X')$, from the given conditions.

Hence applying theorem 1, the result follows. Conversely, assume that (i) and (ii) hold. We will consider those inputs for which the loop body is never executed and those for which it is executed once or more. In the former case, $\neg B(X) \wedge X \in D_1$. Hence $z_f = z_o = h(X_o)$. In the later case, we have $z_o \oplus h(X_o) = z_f \oplus h(X_f)$. Clearly, $\neg B(X_f) \wedge X_f \in D_1$. Thus, $h(X_f) = z_o$, and $z_o \oplus h(X_o) = z_f \oplus z_o$; or $z_f = h(X_o)$ since z_o is the unity of $\oplus$. $\boxed{X}$

A cannonical form of such schemas is then

```
z ← z_o;
while h(X) ≠ z_o do

   Modify X, z so as maintain z ⊕ h(X) a constant
enddo;
```

A special case of the above theorem appears in Katz and Manna [8]. An interesting corollary of the theorem is that there always exists a loop invariant for P1, using only the initial value z_o(of z), operator $\oplus$ and the computed function h. Sometimes a program may have to be transformed slightly to fit into the above form.

Example 4. Following program finds the largest character in a string st, where characters are ordered through a positive integer valued function ord. LC denotes the largest character. Assume that ord(null) = 0, where null denotes the null character/string.

```
    LC ← null;
    while st ≠ null do
        if ord (LC) < ord(head(st)) then LC ← head(st);
        st ← tail(st)
    enddo;
```

Since there is a conditional branching involving LC, theorem 2 is not applicable. However that statement may be rewritten as

```
    LC ← maxhead(LC,st),
where maxhead(x,y): if ord(head(x)) < ord(head(y)) then
                        head(y) else head (x)
                    endif;
```

Maxhead is an associative function and all other conditions in theorem 2 are applicable. $\boxed{X}$

4. Stack Schema

An explicit stack is often used to simulate a recursive procedure by an iterative one. Problems of this kind are frequently encountered in graph algorithms, tree traversal routines etc. We consider a stack schema P2, as given below; stk denotes an object of type stack; t denotes an object of a certain type which is initially put in the stack (all elements of the stack would be of this type); the notation stk ⇐ t stands for "push t onto stk"; t ⇐ stk stands for "remove the top element of stk and put it in t"; Λ stands for a null stack; stk ← (t) stands for "let stk contain only the element t".

P2:

```
    begin
        stk ← (t);
        while stk ≠ Λ do S enddo
    end
```

Example 5. This example is a routine for preorder traversal of a tree T; t denotes the root node of T; left (P), right (P) denote the left and right sons of node P.

```
begin
    stk ← (t);
    while  stk ≠ Λ do
        P ⇐ stk; visit P;
        if right (P) ≠ nil then stk ⇐ right(P);
        if left (P) ≠ nil then stk ⇐ left (P)
    enddo
end;
```

$\boxed{X}$

The usual output specification is that a certain function of t is computed (in some specified variables) at the end of the program. Note that the methods of section 3 are not applicable here since the stack variable is used to govern the branching of the main loop.

We are going to exploit the structure of the stack to arrive at the loop invariant: we know that access to the stack is restricted so that the elements are processed in last-in-first-out order. However, a programmer may save the top element, process the next-to-top element and then process the (saved) top element. We believe that such programming is not "clean"; we thus need a syntactic restriction to avoid such programming. We require that the stack be not examined for emptiness inside S. This will effectively restrict the programmer of S so that he can only remove the top element of the stack or push an element onto the stack; however, he cannot get at the next to top element without potentially getting into an error situation (popping off an empty stack).

We now formally state the requirements on the stack schema. Let (X, stk) be the set of global variables to W(B,S). Let the global variables X to the program be from the domain D_1. Let the domain of the elements that are pushed onto the stack be D_2.

We require that P2 satisfy the following two conditions.

1. stk is not examined for emptiness inside S.
2. D_1 is closed with respect to W(B,S) for any $t \in D_2$ in the stack, i.e. if we start with any $X \in D_1$ and any $t \in D_2$ in the stack, the resulting values of variables X at any iteration are in D_1. Assume that Λ is an element of D_2.

Let the initial and final values of the global variables and stk be (X_o,t_o) and (X_f, Λ) respectively. It is required to show that

$X_f = h(X_o,t_o)$, where h is a function

$h: D_1 \times D_2 \rightarrow D_1$.

Assume that $h(X,\Lambda) = X$, for every $X \in D_1$. Also assume that termination has been proven separately, for every $X \in D_1$ and $t \in D_2$. (It then follows that the program will terminate for any X and any number of elements in the stack as input).

Theorem 3. $X_f = h(X_o,t_o)$, for every $X_o \in D_1$ and $t_o \in D_2$ if and only if,
$h(\ldots h(h(X,t_1),t_2)\ldots),t_n)$ is a loop constant, where $(t_1 \ldots t_n)$ denote the contents of the stack from top to bottom (at any iteration). $\boxed{X}$

Example 6. The following program is a slight modification of example 5. The string 'pre' denotes the sequence of node names which are traversed in preorder traversal of a tree; 'name (P)' gives the name of a node P. $\|$ denotes concatanation.

```
pre ← null;
stk ← (t);
while stk ≠ Λ do
    P ⇐ stk; pre ← pre || name(P);
    if right(P) ≠ nil then stk ⇐ right(P);
    if left (P) ≠ nil then stk ⇐ left(P)
enddo;
```

It is required to prove at the end of the program that $pre_f = preorder(t)$ where 'preorder' is a function (defining the preorder traversal of the tree rooted at t). We first dispose of the initialization to 'pre', using methods of section 3. We then have to prove that starting with any initial value pre_o, $pre_f = pre_o \|$ preorder (t) in the above program (without the initialization of 'pre'). Thus $pre_f = h(pre_o,t)$ where $h(x,y) = x \|$ preorder(t), x a string, t a node at which some tree is rooted. Next, using the fact that D_2 is the set of nodes in the tree rooted at t and $h(x,\Lambda) = x$, we can apply theorem 3.

Let $(t_1 \ldots t_n)$ be the stack contents from top to bottom at any iteration. We then have to prove that $h(h \ldots h(h(pre,t_1),t_2)$ is a loop constant, or
pre $\|$ preorder(t_1) $\|$ preorder (t_2) ... preorder (t_n) is a loop constant, or $pre_o \|$ preorder(t) = pre $\|$ preorder(t_1)...preorder(t_n) is a loop invariant.

This says that the current string 'pre' concatanated with the preorder traversals of $(t_1,t_2,\ldots,t_n)$ remains invariant during successive iterations. $\boxed{X}$

Informal proof of theorem 3: We consider only the loop program with input $X \in D_1$ and any number of elements from D_2 in the stack. Clearly we have closure for such a domain. We want to find the function computed by the loop, starting with such inputs so that we can apply theorem 1. Suppose the stack has $(t_1 \ldots t_n)$ from top to bottom. If we start the loop with input X_o and only t_1 in the stack, it would terminate after sometime with X set to $h(X_o,t_1)$. However with $(t_1 \ldots t_n)$ in the stack, the program will not terminate, although the value of X will be $h(X_o,t_1)$ by the time t_1 is popped off. By the time the next element t_2 is popped off, value of X is $h(h(X_o,t_1),t_2)$. Continuing in the similar manner, it follows that $X_f = h(h \ldots h(h(X_o,t_1),t_2)\ldots),t_n)$. Hence applying theorem 1, $h((h(X,t_1),t_2\ldots,t_n)$ is a loop constant. The proof in the opposite direction is straight forward and is omitted. $\boxed{X}$

The property that we have exploited is the nature of access to a stack which implies that computations with only t_1 on the stack is a prefix of the computation with(t_1,t_2) on the stack (t_1 at the top). Even though the loop does not exactly do the same sequence of computations (as in section 3), the sequence can be obtained through a generic sequence. We note that it was important to prohibit the examination of stack for emptiness in S. We are then assured of the sequence of processing of the elements in the stack. Note that many new elements may be pushed onto the stack during the computation with $(t_1 \ldots t_n)$; however it is not necessary for us to know about them to get the output function.

5. FOR Programs

In previous sections, we studied iterative programs in which the number of loop iterations is a priori unknown. Often, however, we encounter iterative structures, where the number of iterations to be performed is known at entry to the loop. In such cases, it is natural to use a FOR loop. If we attempt to use the approach outlined in previous sections, by rewriting the FOR loop in the do-while form, we encounter several difficulties. Specifically, the loop predicate of such a do-while would examine the iteration parameter. Thus the execution sequence depends on the initial value assigned to the iteration parameter.

In this section we show that proving the correctness of FOR programs is amenable to a simple approach under certain conditions.

Consider a program P3 of the form

```
P3:
    begin
        z ← z_o;
        for i ← 1 to n do
            S
        endfor
    end;
```

Let X be the variables of P3 to which values are assigned at input and suppose at output we would like to show that $z = F(X_o,n)$ where X_o is the initial value of X. We could attempt to locate the extended domain and function relative to z, X and n. Instead, we assume that the FOR program satisfies certain conditions which restrict the dynamic control structure of the program.

These conditions are stated below.

(1) neither n, nor any variable dependent on n appear in S.

(2) S does not alter the value of X.

Condition (1) is sufficient to ensure that for any two different values n_1, n_2 of $n(n_2 > n_1)$, computations are identical (output z, X are identical) through the first n_1 iterations.

Condition (2) states that only z is modified in S. (It can be relaxed as we shall show subsequently.) Thus the loop steps through a set of elements and z accumulates the result.

If the FOR program satisfies the above conditions, a simple form of induction can be used to prove correctness of such a program. This is the content of the following theorem.

Theorem 4. If P3 satisfies conditions (1) and (2) then $z = F(X_o,n)$ holds at the exit of P3 for every X_o and $n \geq 0$, if and only if

(i) $z_o = F(X_o,0)$

and

(ii) $(z = F(X_o, i-1)) \{S\} (z=F(X_o,i))$

for $1 \leq i \leq n$ and all permissable initial values X_o. $\boxtimes$

Proof (outline): It is easy to verify that if (i), (ii) hold then the required proposition is true at the end of the program. Conversely, if $z = F(X_o,n)$ for any X_o, $n \geq 0$, then $z_o = F(X_o,0)$. For $n \geq 1$,

```
z ← $z_o$;
for i←1 to n do S;
```

is equivalent to

```
{z ← $z_o$;
 for i ← 1 to n-1 do S};
```

S (with n substituted for every occurrence of i). Part of the program inside braces have the effect of keeping X invariant (condition 1) and setting $z = F(X_o,n-1)$. Hence it must be that if we start with $z = F(X_o,i-1)$ and execute S, we must get $z = F(X_o,i)$. $\boxtimes$

We will illustrate the theorem with a few examples.

Example 7. The following program sums the elements of an array A(1), ... A(n).

```
sum ← 0;
for i ← 1 to n do
sum ← sum + A(i)
endfor;
```

We wish to show that $\text{sum} = \sum_{j=1}^{n} A(j)$. Then it is necessary and sufficient to show that

(i) $(0 = \sum_{j=1}^{0} A(j))$

and (ii) $(\text{sum} = \sum_{j=1}^{i-1} A(j)) \{\text{sum} \leftarrow \text{sum} + A(i)\}$

$(\text{sum} = \sum_{j=1}^{i} A(i))$ $\boxtimes$

Example 8. The following is a program for computing factorial. (See also example 2).

```
u ← 1;
for r ← 1 to n do
    v ← u;
    for s ←1 to r do
    v ← v + u
    endfor;
    u ← v
endfor;
```

At output we would like to prove that $u = (n + 1)!$

Using theorem (4), we need to prove

(i) $1 = 1!$

(ii) $(u = r!) \left\{ \begin{array}{l} v \leftarrow u; \\ \text{for } s \leftarrow 1 \text{ to } r \text{ do} \\ \quad v \leftarrow v + u \\ \text{endfor}; \\ u \leftarrow v; \end{array} \right\} (u = (r+1)!)$

(ii) above simplifies to

$[(u=r!) \wedge (v=r!)] \left\{ \begin{array}{l} \text{for } s \leftarrow 1 \text{ to } r \text{ do} \\ \quad v \leftarrow v+u \\ \text{endfor}; \end{array} \right\} (v=(r+1)!)$

Note that we cannot apply the same theorem again since the initial values of u, v depend on the iteration parameter r. $\boxtimes$

Suppose now that the FOR body S modifies the value of X. Then theorem 4 fails to hold. In this case, knowing the value of z at the beginning of an iteration, the loop body to be executed, and the value of X_o, does not allow us to conclude the desired property of z at the end of the iteration. The following example illustrates this situation.

Example 9. Consider the following program,

```
z ← 0;
for i ←1 to n do
X ← X + 1;
z ← z + X
endfor;
```

We would like to prove at output $z = n X_o + \frac{n(n+1)}{2}$.

Applying theorem 4, we would obtain

$(z=n X_o + \frac{n(n+1)}{2}) \{X \leftarrow X+1; z \leftarrow z+X\} (z=(n+1)X_o + \frac{(n+1)(n+2)}{2})$

But this cannot be proved since no relation between X_o and X is known.

If we include the function computed in the X-component as well, we obtain

$(n \geq 0) \wedge (X=X_o+n) \wedge (z=n X_o + \frac{n(n+1)}{2}) \{X \leftarrow X+1; z \leftarrow z+X\}$

$(z=(n+1)X_o + \frac{(n+1)(n+2)}{2}) \wedge (X=X_o+n+1)$

and the corresponding verification condition generated is a theorem. $\boxtimes$

We thus observe that condition (2) on P3 can be omitted if we take into account the X-component of the function computed as well. This leads us to the following stronger form of theorem 4, which we state without proof. We assume that the values of X obtained after any number of iterations belong to a closed domain D.

Theorem 5. Let P3 satisfy condition (1) above. Then

$(X=X_o) \wedge (X \in D) \wedge (n \geq 0) \{P3\} (\langle z,X \rangle = \langle F(X_o,n), G(X_o,n) \rangle)$

if and only if

(a) $(\langle z_o, X_o \rangle = \langle F(X_o,0), G(X_o,0) \rangle)$

and

$$\text{(b)}\ (X_o \in D) \wedge (\langle z,X\rangle = \langle F(X_o,i-1), G(X_o,i-1)\rangle)\ \{S\}$$
$$(\langle z,X\rangle = \langle F(X_o,i), G(X_o,i)\rangle) \qquad \boxed{X}$$

We have shown that for a wide class of for loop programs, which define well behaved computations, necessary and sufficient conditions for correctness can be generated from the given output specifications. We observe that we have been able to apply a relatively simple form of induction to the problem since the computations of the FOR programs under consideration behave uniformly relative to the iteration parameter. Further, the output specification itself (in a slightly modified form) serves as the induction hypothesis.

While the class of FOR programs we have investigated includes a large number of naturally occurring programs, it is quite easy to construct FOR programs which fall outside this class, as the following example illustrates.

Example 10. This example uses a modified version of a program from [5]. It sets the boolean variable prime to true or false depending on whether input J is prime or not. The domain D of inputs is $\{J \mid J \geq 3$ and integer$\}$.

```
begin
    prime ← true;
    for i ← 2 to J - 1 do
        if i divides J then prime ← false
    endfor
end;
```

The theorem is not applicable since S depends on the iteration parameter. $\boxed{X}$

6. Conclusion and Discussion

We have investigated three classes of (program) schema, for which we have been able to determine a closed domain and hence a necessary and sufficient loop invariant from the given (partial) input output specifications. If we examine these classes closely, we find that in each case, the extended domain is located by using the fact that the dynamic program behavior is in some sense uniform relative to some input parameters. Thus for the while do programs (sec. 3), the conditions imposed assume that the execution sequence of the program is independent of the initial value assigned to z and further that the result of this computation depends on z in a predictable fashion. In case of the stack schema we observe that the execution sequence of the program depends only on the top element of the stack until such time as the element below it is examined for the first time. Similar remarks hold for the FOR loop. We do not attempt to define this notion of uniform behavior of a program relative to a data space here. However, we hope to have convinced the reader that these semantic (dynamic?) considerations play an important role in constructing programs whose correctness can be demonstrated (or refuted) relative to given input output specifications easily. We hasten to add that the above statement makes no reference to the difficulty of proving or disproving the theoremhood of the verification conditions generated. Rather we have attempted to generate invariance information that is in some sense natural to the problem under consideration.

From the above considerations, it appears that certain basic principles are beginning to emerge which supplement the existing methodologies for program construction. These additional constraints on the program development process would thus lead to programs which in some sense are naturally provable. It is interesting to note that almost all programs which have been constructed with provability in mind do obey the principles discussed above. By explicitly enunciating them and studying their properties, we hope this would take us a little closer to the goal of understanding the "art of programming".

Acknowledgments

This research was partially supported by NSF Grant DCR75-90842. An earlier version of these results were presented at the IEEE Workshop on Theorem Proving held at Chicago, Illinois, June 1975.

References

1. Basu, S.K. and J. Misra, "Proving Loop Programs", IEEE Trans. on Software Engineering, vol. SE-1, March 1975.

2. Basu, S.K. and J. Misra, "Deterministic Generation of Inductive Assertions", presented at IEEE Workshop on Theorem Proving, Chicago, Illinois, June 1975.

3. Burstall, R.M., "Program Proving as Hand Simulation with a Little Induction", Proc. IFIP Congress, 1974.

4. Floyd, R.W., "Assigning Meanings to Programs". Proc. Symposium in Appl. Math., vol. 19, 1967.

5. German, S.M. and B. Wegbreit, "A Synthesizer of Inductive Assertions", IEEE Trans. on Software Engineering, vol. SE-1, March 1975.

6. Good, D.I., "Provable Programming", Proc. Int. Conf. on Reliable Software, Los Angeles, April 1975.

7. Hoare, C.A.R., "An Axiomatic Basis for Computer Programming", CACM 12, 1969.

8. Katz, S. and Z. Manna, "Logical Analysis of Programs", Technical Report, Dept. of Applied Math., Weizmann Institute of Science, Rehovot, Israel, 1975.

9. King, J. C., "A Program Verifier", Carnegie-Mellon University, PhD Thesis, 1969.

10. London, R.L., "A View of Program Verification", Proc. International Conf. on Reliable Software, Los Angeles, April 1975.

11. Manna, Z., "A Mathematical Theory of Computation", McGraw-Hill, 1975.

12. Misra, J., "Relations Univormly conserved by a Loop", Proc. of Int. Symp. on Proving and Improving Programs, Are et Senans, France 1975.

13. Misra, J., "A Study into the Nature of Loop Computation", unpublished manuscript.

14. Morris, J.H., & Wegbreit, B., "Subgoal Induction", Xerox Palo Alto Research Center, Report CSL 75-6, (July 1975)