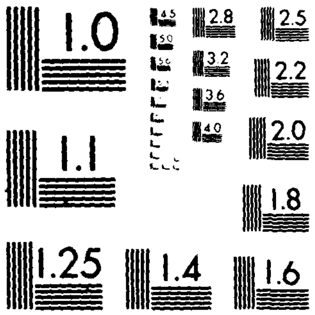
1.0
4.5
5.0
5.6
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

12

COMPUTER SCIENCE
TECHNICAL REPORT SERIES

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

12

Technical Report TR-1115 October, 1981
[redacted]-F49620-80-C-[redacted]
-0001

A Heuristic For Deriving Loop Functions*

Douglas D. Dunlop and Victor R. Basili

Department of Computer Science
University of Maryland
College Park, MD 20742

---

*This work was supported in part by the Air Force Office of Scientific Research Contract [redacted]-F49620-80-C-0001 to the University of Maryland. The material contained in this paper will become part of a dissertation to be submitted to the Graduate School, University of Maryland, by Douglas D. Dunlop, in partial fulfillment of the requirements for the Ph.D. degree in Computer Science.

409 [illegible]

ABSTRACT

The problem of analyzing an initialized loop and verifying that the program computes some particular function of its inputs is addressed. A heuristic technique for solving these problems is proposed which appears to work well in many commonly occurring cases. The use of the technique is illustrated with a number of applications. A hierarchy of initialized loops is suggested which is based on the "effort" required to apply this methodology in a deterministic (i.e. guaranteed to succeed) manner. It is explained that in any case, the success of the proposed heuristic relies on the loop exhibiting a "reasonable" form of behavior. An informal categorization of such programs is made which is based on two opposing problem solving strategies. It is suggested that our heuristic is naturally suited for use on programs in one of these categories.

KEYWORDS and PHRASES: program verification, initialized loop programs, loop functions, constraint functions, BU programs, TD programs

CR CATEGORIES: 5.24

A Heuristic For Deriving Loop Functions

1. Introduction

In this report, we will consider programs of the following form:

```
<INITIALIZATION STATEMENTS>
while <LOOP PREDICATE> do
   <LOOP BODY STATEMENTS>
   od.
```

These programs tend to occur frequently in programming in order to accomplish some specific task, e.g. sort a table, traverse a data structure, calculate some arithmetic function, etc. More precisely, the intended purpose of such a program is often to compute, in some particular output variable(s), a specific function of the program inputs. In this paper, we address the problem of analyzing a program of the above form in order to prove its correctness relative to this intended function.

One common strategy taken to solve this problem is to heuristically synthesize a sufficiently strong inductive assertion (i.e. loop invariant [Hoare 69]) for proving the correctness of the program. A large number of techniques to aid in the discovery of these assertions have appeared in the literature (see, for example, [Wegbreit 74, Katz & Manna 76]). It is our view, however, that these techniques seem to be more "machine oriented" than "people oriented." That is, they seem geared toward use in an assertion generator for an automatic program verification system. Furthermore, a sizable portion of the complexity of these techniques is due to their general purpose nature. The methodology proposed here is intended to be used by programmers in the process of reading (i.e. understanding, documenting, verifying, etc.) programs and is tailored to the commonly occurring verification problem discussed above.

An alternative to the inductive assertion approach which is taken in this paper is to invent an hypothesis concerning the general input/output behavior of the WHILE loop. Once this has been done, the loop can be proven/disproven correct with respect to the hypothesis using standard techniques [Mills 72, Mills 75, Basu & Misra 75, Morris & Wegbreit 77, Wegbreit 77, Misra 78]. If the hypothesis is shown to be valid, the correctness/incorrectness of the program in question follows immediately. It has been shown [Basu & Misra 75, Misra 78, Misra 79, Basu 80] that this loop hypothesis can be generated in a deterministic manner (i.e. one that is guaranteed to succeed) for two restricted classes of programs. The approach suggested here is similar to this method in that the same type of loop behavior seems to be exploited in order to obtain the hypothesis. Our approach is not deterministic in general, but as a result, is intended to be more widely applicable and easier to use than those previously proposed in the literature.

One view of the problem of discovering the general input/output behavior of the WHILE loop under consideration might be to study it and make a guess about what it does. One might go about doing this by "executing" the loop by hand on several sample inputs and then guessing some general expression for the input/output behavior of the loop based on these results. Decisions that need to be made when using such a technique include how many sample inputs to use, how should these inputs be selected, and how should the general expression be inferred. Another consideration is that hand execution can be a difficult and an error prone task. Indeed, it seems that the loops for which hand execution can be carried out in a straightforward manner are the ones that are least in need of verification or some other type of formal analysis.

Our methodology is similar to this technique in that we attempt to infer the general behavior of the loop from several sample loop behaviors. In contrast to this technique, however, the sample behaviors are not obtained from hand execution, rather they are obtained from the specification for the initialized loop program. In many of the cases we have studied, the general behavior of the loop in question is quite easy to guess from these samples. This is not to say that the loop computes a "simple" function of its inputs or that the loop necessarily operates in a "simple" manner. Much more accurately, the ease with which the general behavior can be inferred from the samples is due to a "simple" connection between a change in the input value of an initialized variable and the corresponding change caused in the result produced by the loop. We will expand on this idea in what follows.

2. The Technique

In order to describe the proposed technique, we represent the verification problem discussed above as follows:

```
{X ∈ D(f)}
X := K(X);
while B(X) do
    X := H(X)
   od
{v=f(X0)}.
```

In this notation, X represents the data state of the program. The symbols K and H are data state to data state functions corresponding to the effects of the initialization and loop body respectively. The function B is a predicate over the data state. The program is specified to produce in the variable v a function f of the input data state X0. The notation D(f) appearing in the program precondition is the domain of the function f, i.e. the set of states for which f is defined.

If D is the set of all possible program data states and T is the set of values that the variable v may assume, the specification function f has the functionality f : D -> T. In order to verify a program of this form, we choose to find a function g : D -> T which describes the input/output characteristics of the WHILE loop over a suitably general input domain. Specifically, this input domain must be large enough to contain all the intermediate data states generated as the loop iterates. If this is the case, the loop is said to be closed [Basu & Misra 75, Misra 73] for the domain of g.

A Heuristic For Deriving Loop Functions

We briefly consider two alternative approaches to synthesizing this loop function g. The alternatives correspond to the "top down" and "bottom up" approaches to creating inductive assertions discussed in [Katz & Manna 73, Ellozy 81]. In the "top down" alternative, the hypothesis g answers the question "what would the general behavior of the loop have to be in order for the program to be correct?" If such an hypothesis can be found and verified, the correctness of the program is established. If the program is incorrect, no such valid hypothesis exists. In the "bottom up" alternative, the hypothesis g answers the question "what is the general behavior of the loop?" In this case, a valid hypothesis always exists. Once it has been found and verified, the program is correct if and only if the initialization followed by g is equivalent to the function f.

The advantage of a "top down" approach is that it is usually easier to apply in practice because the verifier has more information to work with when synthesizing the hypothesis. The disadvantage of such an approach is that it may not be as well-suited to disproving the correctness of programs. This is because to disprove a program, the verifier must employ an argument which shows that there does not exist a valid hypothesis. The method described in this paper is based on the "top down" approach. We will return to a discussion of this advantage and disadvantage later.

We begin by assuming the program in question is correct with respect to its specification. We then consider several properties of the function g which result from this assumption. First, the correctness of the program implies

(1) $X0 \in D(f) \rightarrow f(X0)=g(K(X0))$.

That is, for inputs satisfying the program precondition, the initialization followed by the loop yields the desired result. Secondly, since the loop computes g,

$$B(X0) \rightarrow g(X0)=g(H(X0))$$

holds by the "iteration condition" [Misra 78] of the standard technique for showing the loop computes g. This implies

$$B(K(X0)) \rightarrow g(K(X0))=g(H(K(X0))).$$

Combining with (1) yields

(2) $X0 \in D(f),\ B(K(X0)) \rightarrow f(X0)=g(H(K(X0)))$.

At this point we choose to introduce an additional universally quantified state variable X into each of (1) and (2). The results are the equivalent conditions

(1') $X0 \in D(f),\ X=K(X0) \rightarrow g(X)=f(X0)$

and

(2') $X0 \in D(f),\ B(K(X0)),\ X=H(K(X0)) \rightarrow g(X)=f(X0)$.

We summarize by saying that if the program is correct with respect to its specification, conditions (1') and (2') hold.

Suppose now that the specification (f), and the input/output behavior of the initialization (K), loop predicate (B) and loop body (H) are known. Given this, (1') and (2') can be used to solve for the loop hypothesis g on a certain set of inputs assuming the correctness of the program. Indeed, (1') and (2') can be thought of as defining portions of the unknown loop function g we are seeking. Specifically, each of (1′) and (2′) can be viewed as defining a function g with a restricted domain. In this light, for example, (1′) defines the function (i.e. set of ordered pairs)

$$g = \{(X,Z) \mid \underline{TE}\ X0 \in D(f)\ \underline{ST}\ (X=K(X0)\ \&\ Z=f(X0))\}.$$

We call (1′) and (2′) <u>constraint</u> <u>functions</u> since they are functions and serve as constraints (i.e. requirements) on the general loop function. More precisely put, the constraint functions are subsets of the general loop function. The hope is that if these subsets are representative of the whole, the general loop function may be inferred through analysis of the constraint functions.

In what follows we describe a 4 step process for constructing a general loop function g from these constraint functions. We suggest that the reader not be taken aback by what may appear to be considerable complexity in the description of our technique. We intentionally have attempted to describe the procedure in a careful, precise manner. Furthermore, the technique is based on a few simple ideas and, once those ideas have been learned, we feel it can be applied with a considerable amount of success.

<u>Example 1</u> - As we describe these steps, we will illustrate their application on the following trivial program to compute multiplication:

```
{v>=0}
z := 0;
while v ≠ 0 do
   z := z + k;
   v := v - 1
   od
{z=v0*k}.
```

We proceed with the example analysis as follows.

Step 1 : RECORD - The first step consists of recording the constraint functions (copied from (1′) and (2′))

C1: X0 ∈ D(f), X=K(X0) -> g(X)=f(X0)

C2: X0 ∈ D(f), B(K(X0)), X=H(K(X0)) -> g(X)=f(X0)

As a notational convenience, we dispense with the data state notation and use program variables (possibly subscripted by 0 to denote their initial values) in these function definitions. The terms X0 ∈ D(f) and f(X0) come from the pre and post conditions for the initialized loop respectively. The term X=K(X0) is based on the input/output behavior of the initialization, and the terms B(K(X0)) and X=H(K(X0)) together describe the input/output behavior of the initialization followed by exactly 1 loop iteration. We illustrate these ideas with the multiplication program in Example 1. The constraint functions for this program are as follows:

C1: v0>=0, v=v0, z=0 -> g(z,v,k)=v0*k

C2: v0>0, v=v0-1, z=k -> g(z,v,k)=v0*k.

We make the following comments concerning these function definitions. First, in the interest of simplicity, we do not RECORD the "affect" of the initialization or loop body on the constant k (i.e. we dispense with k=k0 and the need for a symbol k0).

Secondly, g is defined as a function of each program variable which occurs in the loop predicate or loop body. That is, g is a function of the variables on which the behavior of the loop directly depends. Furthermore, note that in C2, the term v0>0 captures both X0 $\in$ D(f) (i.e. v0>=0) and B(K(X)) (i.e. v0$\neq$0). As a final remark, in a constraint function we will use the phrase <u>domain requirement</u> to refer to the collection of terms to the left of the "->" symbol and <u>function expression</u> to refer to the expression which defines the value of g (e.g. v0*k in both C1 and C2 above).

<u>Step 2</u> : <u>SIMPLIFY</u> - All variables which appear in the function definition but not in the parameter list for g must eventually be <u>eliminated</u> from the definition. On occasion, it is possible to solve for the value of such a variable in the domain requirement and substitute the equivalent expression for it throughout the definition. To illustrate, in the definition C1 above, v0 is a candidate for elimination. We know its value as a function of v (i.e. v0=v), hence we can SIMPLIFY this definition to

```
  C1:  v>=0, z=0 -> g(z,v,k)=v*k.
```

Note that the term v=v0 has disappeared since with the substitution it is equivalent to TRUE. In a similar manner, the second constraint function can be SIMPLIFIED to (using v0=v+1)

```
  C2:  v>=0, z=k -> g(z,v,k)=(v+1)*k.
```

Although applying this simplifying heuristic is most often a straightforward process, care must be taken to insure that the domain of the constraint function is not mistakenly extended. For example, if d and d0 are integer variables, the definition

    d0>0, d=d0*2 -> g(d)=d0*8

does not SIMPLIFY to

    d>0 -> g(d)=d*4

since the first function defines a value of g only for positive, even values of d while the second definition defines a value of g for all positive d. The first function does SIMPLIFY to

    d>0, EVEN(d) -> g(d)=d*4

where EVEN(d) is a predicate which is TRUE iff d is even.

Step 3 : REWRITE - Variables which appear in the parameter list for g but not in the function expression of its definition are candidates to be introduced into the function expression. Each of these variables will be bound to a term in the domain requirement of the definition. The purpose of this step is to rewrite the function expression of C2 (based on the properties of the operation(s) involved) in order to introduce these terms into the function expression. To illustrate, consider the above SIMPLIFIED C2 definition. The variable z is a candidate to be introduced into the function expression (v+1)*k. It is bound to the term k in the domain requirement. Thus we need to introduce an additional term k into this function expression. One way to do this is to translate the expression to v*k+k. Based on this, we REWRITE C2 as

  C2: v>=0, z=k -> g(z,v,k)=v*k+k.

Step 4 : SUBSTITUTE - In steps 2 and 3, the constraint functions are massaged into equivalent definitions in order to facilitate step 4. The purpose of this step is to attempt to infer a general loop function from these constraints. We motivate the process as follows. Suppose we are searching for a particular relationship between several quantities, say E, m and c. Furthermore, suppose that through some form of analysis we have determined that when m has the value 17, the relationship E=17*(c**2) holds. A reasonable guess, then, for a general relationship between E, m and c would be E=m*(c**2). This would be particularly true if we had reason to suspect that there was a relatively simple connection between the quantities m and E. We arrived at the general relationship by substituting the quantity m for 17 in the relationship which is known to hold when m has the value 17. Viewed in this light, the purpose of the constraint function C2 is to obtain a relationship which holds for a specific value of m (e.g. 17). The step REWRITE exposes the term 17 in this relationship. Finally, SUBSTITUTE substitutes m for 17 in the relationship and proposes the result as a general relationship between E, m and c. In terms of the multiplication program being considered, the SUBSTITUTE step calls for replacing one of the terms k in the above rewritten function expression with the term z. The two possible substitutions lead to the following general functions:

```
    v>=0 -> g(z,v,k)=v*k+z
```

and

```
    v>=0 -> g(z,v,k)=v*z+k.
```

Both of these (necessarily) are generalizations (i.e. supersets) of C2, however, only the first is also a generalization of C1. Hence this function is hypothesized as a description of the general behavior of the above WHILE loop.

We have applied the above 4 steps to obtain an hypothesis for the behavior of the loop in question. Since this description is sufficiently general (specifically, since the loop is closed for the domain of the function), we can prove/disprove the correctness of the hypothesis using standard verification techniques [Mills 75, Misra 78]. Specifically, the hypothesis is valid if and only if each of

- the loop terminates for all v>=0,
- v=0 -> z=z + v*k, and
- z + v*k is a loop constant (i.e. v0*k0=z + v*k is a loop invariant)

hold. We remark that the loop hypothesis is selected in such a way that if it holds (i.e. the loop does compute this general function), the initialized loop is necessarily correct with respect to f.

We emphasize that there are usually an infinite number of generalizations of the constraint functions C1 and C2, and that, depending on how REWRITE and SUBSTITUTE are applied, the technique is capable of generating any one of these generalizations. For example, REWRITE and SUBSTITUTE applied to the multiplication example could have produced

C2: v>=0, z=k -> g(z,v,k)=

```
     v*k + 3*k + k*k*(v-7)/(4*k) + k*k*k/(k*k)
                   - k*k*k*(v-7)/(4*k*k) - k*k*k*3/(k*k)
```

and

```
        v>=0 -> g(z,v,k)=
     v*k + 3*z + z*z*(v-7)/(4*k) + z*z*z/(k*k)
                   - z*z*z*(v-7)/(4*k*k) - z*z*z*3/(k*k)
```

respectively, where "/" denotes an integer division (with truncation) infix operator which yields 0 when its denominator is 0. This last function is also a generalization of C1 and C2.

It has been our experience, however, that many initialized loops occur in which there exists some relatively simple connection between different input values of the variables constrained by initialization and the corresponding result produced by the WHILE loop. Most often in practice, these variables are bound to values in the domain requirement of C2 which suggest an application of REWRITE that uncovers this relationship and leads to a correct hypothesis concerning the general loop behavior. In the following section we illustrate a number of example applications of this technique.

3. Applications

Example 2 - The following program computes integer exponentiation. This example serves to illustrate the use of the technique when the loop body contains several paths:

```
{d>=0}
w:=1;
while d ≠ 0 do
    if odd(d) then w := w * c fi;
    c := c*c; d := d/2
    od
{w=c0 ^ d0}.
```

The infix operator ^ appearing in the postcondition represents integer exponentiation. The first constraint function is easily obtained:

```
 d0>=0, c=c0, d=d0, w=1 -> g(w,c,d)=c0^d0
```

and SIMPLIFIES to

```
C1: d>=0, w=1 -> g(w,c,d)=c^d.
```

Since there exist two paths through the loop body, we will obtain two second constraint functions. The first of these deals with the path which updates the value of w and is executed when the input value of d is odd. The function is

```
 d0>0, odd(d0), w=c0, c=c0*c0, d=d0/2 -> g(w,c,d)=c0^d0
```

which SIMPLIFIES to

```
C2a:  d>=0, c=w*w -> g(w,c,d)=w^(d*2+1).
```

The function corresponding to the other loop body path is

```
 d0>0, even(d0), w=1, c=c0*c0, d=d0/2 -> g(w,c,d)=c0^d0
```

and SIMPLIFIES to

```
       d>=0, w=1, SQUARE(c) -> g(w,c,d)=SQRT(c)^(d*2)
```

i.e.

```
C2b:  d>=0, w=1, SQUARE(c) -> g(w,c,d)=c^d
```

where SQUARE(x) is a predicate which is TRUE iff x is a perfect square and SQRT(x) is the square root of the perfect square x. This term is necessary in the domain requirement since the unSIM-

PLIFIED function is only defined for values of c which are perfect squares. Note that C2b is a subset of C1 and hence is of no additional help in characterizing the general loop function. The heuristic suggested in REWRITE is to rewrite the function expression w^(d*2+1) of C2a in terms of w, w*w (so as to introduce c) and d. The peculiar nature of the exponent in this expression leads one to the equivalent formula w*((w*w)^d). Applying SUBSTITUTE in C2a yields

```
  d>=0 -> g(w,c,d)=w*(c^d).
```

This function is in agreement with (i.e. is a superset of) C1 and thus is a reasonable hypothesis for the general loop function.

In this example, the portion of C2 corresponding to the loop body path which bypasses the updating of the initialized data is redundant with C1. Based on this, one might conclude that such loop body paths should be ignored when constructing C2. Considering all loop body paths, however, does have the advantage that an incorrect program could possibly be disproved (at the time the general loop function is being constructed) by observing an inconsistency between constraint functions C1 and C2. For instance, in the example, if the assignment to c had been written "c:=c*2", the above analysis would have detected an inconsistency in the constraints on the general loop function. Such an inconsistency implies that the hypothesis being sought for the general behavior of the loop does not exist, and hence, that the program is not correct with respect to its specification.

A Heuristic For Deriving Loop Functions

In the previous section, the reader may recall that awkwardness in disproving programs was offered as a disadvantage of a "top down" approach to synthesizing g. It has been our experience, however, that, as in the above instance, an error in the program being considered often manifests itself as an inconsistency between C1 and C2. Such an inconsistency is usually "easy" to detect and hence the program is "easy" to disprove. While it is difficult to give a precise characterization of when this will occur, intuitively, it will be the case provided that the "error" (e.g. c*2 for c*c) can be "executed" on the first iteration of the loop.

Example 3 - The following program counts the number of nodes in a nonempty binary tree using a set variable s. It differs from the previous example in that more than 1 variable is initialized. The tree variable t is the input tree whose nodes are to be counted. We use the notation left(t) and right(t) for the left and right subtrees of t respectively. The predicate empty(t) is TRUE iff t is the empty tree (i.e. contains 0 nodes).

```
{~empty(t)}
n := 0; s := {t};
while s ≠ {} do
    select and remove some element e from s;
    n := n + 1;
    if ~empty( left(e)) then s := s U {left (e)} fi;
    if ~empty(right(e)) then s := s U {right(e)} fi
    od
{n=NODES(t)}
```

The notation NODES(t) appearing in the postcondition stands for the number of nodes in binary tree t. The first constraint function is

```
  C1: ~empty(t), n=0, s={t} -> g(n,s)=NODES(t).
```

Rather than considering each of the 4 possible paths through the loop body individually, we abstract the combined effect of the two IF statements as the assignment

```
       s := s U SONS(e),
```

where SONS(x) is the set of 0, 1 or 2 nonempty subtrees of x. Applying this, the second constraint function is

```
  C2: ~empty(t), n=1, s=SONS(t) -> g(n,s)=NODES(t).
```

We choose to REWRITE the function expression for C2 using the recursive definition that NODES(x) for a nonempty tree x is 1 plus the NODES value of each of the 0, 1 or 2 nonempty subtrees of x. Specifically, this would be

```
     1+SUM(x,SONS(t),NODES(x))
```

where SUM(A,B,C) stands for the summation of C over all A ∈ B. Applying SUBSTITUTE in the obvious way yields

```
     ~empty(t) -> g(n,s)=n+SUM(x,s,NODES(x))
```

which is in agreement with C1 and is thus a reasonable guess for the general loop function g.

Two remarks are in order concerning this example. The first deals with the condition ~empty(t) appearing in the domain requirement of the obtained function. The reader may wonder, if t is not referenced in the loop (it is not in the parameter list for g), how can the loop behavior depend on empty(t)? The answer is that it obviously cannot; the above function is simply equivalent to g(n,s)=n+SUM(x,s,NODES(x)).

For the remainder of the examples of this section, we assume that these unnecessary conditions are removed from the domain requirement of the constraint function as part of the SUBSTITUTE step.

As a second point, in Example 3 we encounter the case where the obtained function is, strictly speaking, too general, in that its domain includes "unusual" inputs for which the behavior of the loop does not agree with the function. For instance, in the example, the loop computes the function

g(n,s)=n+SUM(x,s,NODES(x))

only under the provision that the set s does not contain the empty tree. This is normally not a serious problem in practice. One proceeds as before, i.e. attempts to push through a proof of correctness using the inferred function. If the proof is successful, the program has been verified; otherwise, the characteristics of the input data which cause the verification condition(s) to fail (e.g. s contains an empty tree) suggest an appropriate restriction of the input domain (e.g. s contains only nonempty trees) and the program can then be verified using this new, restricted function.

Example 4 [Gries 79] - Ackermann's function A(m,n) can be defined as follows for all natural numbers m and n:

```
A(0,n)       = n+1
A(m+1,0)     = A(m,1)
A(m+1,n+1)   = A(m,A(m+1,n)).
```

The following program computes Ackermann's function using a sequence variable s of natural numbers. The notation s(1) is the rightmost element of s and s(2) is the second rightmost, etc. The sequence s(..3) is s with s(2) and s(1) removed. We will use < and > to construct sequences, i.e. a sequence s consisting of n elements will be written <s(n), ... ,s(2),s(1)>.

```
{m>=0,n>=0}
s := <m,n>;
while size(s) ≠ 1 do
   if s(2) = 0 then      s:=s(..3)||<s(1)+1>
   elseif s(1)=0 then    s:=s(..3)||<s(2)-1,1>
   else                  s:=s(..3)||<s(2)-1,s(2),s(1)-1> fi
   od
{s=<A(m,n)>}
```

For this program, the first constraint function is

```
  C1:  m>=0, n>=0, s=<m,n> -> g(s)=<A(m,n)>.
```

The second constraint functions corresponding to the 3 paths through the loop body are

```
  C2a:  m=0, n>=0, s=<n+1>         -> g(s)=<A(m,n)>
  C2b:  m>0, n =0, s=<m-1,1>       -> g(s)=<A(m,n)>
  C2c:  m>0, n >0, s=<m-1,m,n-1>   -> g(s)=<A(m,n)>.
```

REWRITING these 3 based on the above definition of A yields

```
        m=0, n>=0, s=<n+1>         -> g(s)=<n+1>
        m>0, n =0, s=<m-1,1>       -> g(s)=<A(m-1,1)>
        m>0, n >0, s=<m-1,m,n-1>   -> g(s)=<A(m-1,A(m,n-1))>.
```

SUBSTITUTING here yields

```
        s=<s(1)>                   -> g(s)=<s(1)>
        s=<s(2),s(1)>              -> g(s)=<A(s(2),s(1))>
        s=<s(3),s(2),s(1)>         -> g(s)=<A(s(3),A(s(2),s(1)))>.
```

Note that the second of these functions implies C1. The 3 seem to suggest the general loop behavior (where n>1)

```
  g(<s(n),s(n-1), ... ,s(1)>) =
                         <A(s(n),A(s(n-1), ... A(s(2),s(1)) ... ))>.
```

We remark that in the first 3 examples, the heuristic resulted in a loop function which was sufficiently general (i.e. the loop was closed for the domain of the inferred function). Example 4 illustrates that this does not always occur. The loop function heuristic is helpful in the example in that it suggests a behavior of the loop for general sequences of length 1, 2 and 3. Based on these results, verifier is left to infer a behavior for a sequence of arbitrary length.

Example 5 - Let v be a one dimensional array of length n>0 which contains natural numbers. The following program finds the maximum element in the array:

```
m := 0; i := 1;
while i <= n do
   if m < v[i] then m := v[i] fi;
   i := i + 1
   fi
{m=BIGGEST(v)}
```

The notation BIGGEST(v) appearing in the postcondition stands for the largest element of v. The following constraint functions are obtained

```
  C1:  m=0,     i=1 -> g(m,i,v,n)=BIGGEST(v)
  C2:  m=v[1], i=2 -> g(m,i,v,n)=BIGGEST(v).
```

Noticing the appearance of v[1] and 2 in C2, we REWRITE BIGGEST(v) in C2 as MAX(v[1],BIGGEST(v[2..n])), where MAX returns the largest of its two arguments, and v[2..n] is a notation for the subarray of v within the indicated bounds. The generalization which suggests itself,

g(m,i,v,n)=MAX(m,BIGGEST(v[i..n])),

agrees with C1.

Example 6 - If p is a pointer to a node in a binary tree, let POST(p) be the sequence of pointers which point to the nodes in a postorder traversal of the binary tree pointed to by p. The following program constructs POST(p) in a sequence variable vs using a stack variable stk. We use the notation l(p) and r(p) for the pointers to the left and right subtrees of the tree pointed to by p. If p has the value NIL, POST(p) is the empty sequence. The variable rt points to the root of the input tree to be traversed.

```
p := rt; stk := EMPTY; vs := <>;
while ~(p=NIL & stk=EMPTY) do
   if p≠NIL then
      stk <= p /* push p onto stk */ ;
      p := l(p)
   else
      p <= stk /* pop stk */ ;
      vs := vs || <p>;
      p := r(p) fi
   od
{vs = POST(rt)}.
```

Up until now, we have attempted to infer a general loop function from two constraint functions. Of course, there is nothing special about the number two. In this example, the "connection" between the initialized variables and the function values is not clear from the first two constraint functions and it proves helpful to obtain a third constraint function. Functions C1 and C2 correspond to 0 and 1 loop body executions, respectively. The third constraint function C3 will correspond to 2 loop body executions. We will use the notation (e1, ... ,en) for a stack containing the elements e1, ... ,en from top to bottom. The constraint functions for this program are

```
C1:                         p=rt,        stk=EMPTY,       vs=<>    ->
        g(p,stk,vs)=POST(rt)
C2:  rt≠NIL,                p=l(rt),     stk=(rt),        vs=<>    ->
        g(p,stk,vs)=POST(rt)
C3a: rt≠NIL, l(rt)≠NIL, p=l(l(rt)), stk=(l(rt),rt), vs=<>    ->
        g(p,stk,vs)=POST(rt)
C3b: rt≠NIL, l(rt)=NIL, p=r(rt),     stk=EMPTY,       vs=<rt> ->
        g(p,stk,vs)=POST(rt).
```

Note that there are two third constraint functions. C3a and C3b correspond to executions of the first and second loop body paths (on the second iteration), respectively. There is only 1 second constraint function since only the first loop body path can be executed on the first iteration. Using the recursive definition of POST, we REWRITE C2, C3a and C3b as follows:

```
C2':  rt≠NIL,               p=l(rt),     stk=(rt),        vs=<>    ->
        g(p,stk,vs)=POST(l(rt)) ||<rt>|| POST(r(rt))
C3a': rt≠NIL, l(rt)≠NIL, p=l(l(rt)), stk=(l(rt),rt), vs=<>    ->
        g(p,stk,vs)=POST(l(l(rt))) ||<l(rt)>|| POST(r(l(rt)))
                                   ||<rt>|| POST(r(rt))
C3b': rt≠NIL, l(rt)=NIL, p=r(rt),    stk=EMPTY,       vs=<rt> ->
        g(p,stk,vs)=<rt> || POST(r(rt)).
```

Applying SUBSTITUTE to each of C2´, C3a´ and C3b´ suggests

```
 stk=(e1),    vs=<> -> g(p,stk,vs)=POST(p) ||<e1>|| POST(r(e1))
 stk=(e1,e2), vs=<> -> g(p,stk,vs)=POST(p) ||<e1>|| POST(r(e1))
                                           ||<e2>|| POST(r(e2))
 stk=EMPTY          -> g(p,stk,vs)=vs || POST(p)
```

respectively. The first 2 of these functions imply the following behavior for an arbitrary stack where vs has the value <>:

```
 stk=(e1, ..., en), vs=<> -> g(p,stk,vs) =
           POST(p) || (<e1>|| POST(e1) || ... ||<en>|| POST(en))
```

and in combination with the last function, the general behavior

```
 stk=(e1, ..., en)        -> g(p,stk,vs) =
     vs || POST(p) || (<e1>|| POST(e1) || ... ||<en>|| POST(en))
```

is suggested.

In this section we have illustrated the use of our technique on a number of example programs. The reader has seen that the success of the method hinges largely on the way REWRITE is performed. What guidelines can be used in deciding how to apply this step? The general rule given above is to identify the variables that need to be introduced into the expression and then to rewrite the expression using the terms to which these variables are bound. For instance in Example 3, NODES(t) was rewritten using the terms 1 and SONS(t). Beyond this rule, however, the reader may have noticed an additional similarity in the way REWRITE was applied in these examples. If f is the function or operation the initialized loop program is intended to compute, each REWRITE step involved <u>decomposing</u> an application of f in some way. In Example 1, for instance, a multiplication operation was decomposed into an addition and multiplication operation; in Example 3, a NODES operation was decomposed into a summation and a number of NODES operations; in Example 5, a BIGGEST operation was decomposed into a MAX and a BIGGEST operation. In Section 6 we will characterize this idea of decomposing the intended operation of the initialized loop program and discuss several implications of the characterization for the proposed technique.

In Example 6, we saw that the technique generalizes to the use of 3 (and indeed an arbitrary number of) constraint functions. We have seen that each of these functions defines a subset of the general loop function g being sought. If the constraint functions themselves are sufficiently general, it may be that the first several of these functions, taken collectively, constitute a complete description of g. We consider this situation in the following section.

4. Complete Constraints

The technique described above for obtaining a general loop function is "nondeterministic" in that the constraint functions do not precisely identify the desired function; rather they serve as a formal basis from which intelligent guesses can be made concerning the general behavior of the loop. Our belief is that it is often easy for a human being to fill in the remaining "pieces" of the loop function "picture" once this basis has been established.

A Heuristic For Deriving Loop Functions

There exist, however, circumstances when the constraints do constitute a complete description of an adequate loop function. Specifically, this description may be complete through the use of 1, 2 or more of the constraint functions. The significance of these situations is that no guessing or "filling in the picture" is necessary; the program can be proven/disproven correct using the constraints as the general loop function. In this section we give a formal characterization of this circumstance.

Definition - For some N > 0, an initialized loop is N-closed with respect to its specification f iff the union of the constraint functions C1,C2, ... ,CN is a function g such that the loop is closed for the domain of g. In this case, the constraints C1,C2, ... CN are complete.

Thus if a loop is N-closed for some N>0, the union of the first N constraint functions constitutes an adequate loop function for the loop under consideration. Intuitively, the value N is a measure of how quickly (in terms of the number of loop iterations) the variables constrained by initialization take on "general" values.

Example 7 - The following program

```
{b>=0}
a := a + 1;
while b > 0 do
      a := a + 1;
      b := b - 1
      od
{a=a0 + b0 + 1}
```

is 1-closed since the first constraint function is

```
  C1:  b0>=0, a=a0+1, b=b0 -> g(a,b)=a0+b0+1
```

which SIMPLIFIES to

```
       b>=0 -> g(a,b)=a+b
```

and the loop is closed for the domain of this function. Thus C1 by itself defines an adequate loop function.

Initialized loops which are 1-closed seem to occur rarely in practice. Somewhat more frequently, an initialized loop will be 2-closed. For these programs, the loop function synthesis technique described above (using 2 constraint functions) is deterministic.

Example 8a - Consider the program

```
    sum := 0;
    while seq ≠ EMPTY do
       sum := sum + head(seq);
       seq := tail(seq)
       od
    {sum=SIGMA(seq0)}.
```

The notation SIGMA(seq0) appearing in the postcondition stands for the sum of the elements in the sequence seq0. The program is 2-closed since the second constraint function is

```
  C2:  seq0≠EMPTY, sum=head(seq0), seq=tail(seq0) ->
                                         g(sum,seq)=SIGMA(seq0)
```

which SIMPLIFIES to

```
       g(sum,seq)=sum+SIGMA(seq).
```

The loop is trivially closed for the domain of this function.

Example 8b - As a second illustration of a 2-closed initialized loop, the following program tests whether a particular key appears in an ordered binary tree.

```
success := FALSE;
while tree ≠ NULL & ~success do
   if     name(tree) = key then success := TRUE
   elseif name(tree) < key then tree := right(tree)
   else                         tree := left(tree) fi
   od
{success = IN(key,tree0)}
```

The notation IN(key,tree0) is a predicate which is true iff key occurs in ordered binary tree tree0. This program is also 2-closed. Note that the first constraint function

C1: success=FALSE, tree=tree0 ->

g(success,tree,key)=IN(key,tree0)

SIMPLIFIES to

success=FALSE -> g(success,tree,key)=IN(key,tree).

If we consider the first path through the loop body, the second constraint function is

C2: success=TRUE, tree0≠NIL, tree=tree0, key=name(tree) ->

g(success,tree,key)=IN(key,tree0)

which SIMPLIFIES to

success=TRUE, tree≠NIL, key=name(tree) ->

g(success,tree,key)=IN(key,tree).

Although the domain of the union of these two functions is somewhat restricted, i.e.

{<success,tree,key> |

((~success) OR (tree≠NIL & key=name(tree)))},

the loop is nevertheless closed for this domain and hence the initialized loop is 2-closed.

Example 8c - Consider the sequence of initialized loops P1,P2,P3 ... defined as follows for each I>0:

```
PI : {x>=0}
     x := x * I;
     while x > 0 do
        x := x - I;
        y := y + k
        od
     {y=y0 + x0*I*k}.
```

For any I>0, the first I constraint functions for program PI are

```
  C1: x0>=0,    x=x0*I,        y=y0          -> g(z,y,k)=y0+x0*I*k
  C2: x0>=1,    x=x0*I-1,      y=y0+k        -> g(z,y,k)=y0+x0*I*k
   .
   .
   .
  CI: x0>=I-1, x=x0*I-(I-1), y=y0+k*(I-1) -> g(x,y,k)=y0+x0*I*k.
```

These SIMPLIFY to

```
      x>=0, MI(x)        -> g(x,y,k)=y+x*k
      x>=0, MI(x+1)      -> g(x,y,k)=y+x*k
       .
       .
       .
      x>=0, MI(x+(I-1)) -> g(z,y,k)=y+x*k
```

where MI is a predicate which is TRUE iff its argument is a multiple of I. Since the union of these is the function

x>=0 -> g(x,y,k)=y+x*k,

and the loop is closed for the domain of this function, we conclude that for each I>0, program PI is I-closed.

For many initialized loops which seem to occur in practice, however, there does not exist an N such that they are N-closed with respect to their specifications. This means that no finite number of constraint functions will pinpoint the appropriate generalization exactly; i.e. when applying the above technique in these situations, some amount of inferring or guessing will always be necessary. A case in point is the integer multiplication program from Example 1. The constraint functions C1,C2,C3, ... define the general loop behavior for z=0, z=k, z=2*k, ... etc. The program cannot be N-closed for any N since with input v=N+1, the last value of z will be (N+1)*k which is not in the domain of any of these constraint functions.

As a final comment concerning N-closed initialized loops, it may be instructive to consider the following intuitive view of these programs. All 1-closed and 2-closed initialized loops share the characteristic that they are "forgetful", i.e. they soon lose track of how "long" they have been executing and lack the necessary data to recover this information. This is due to the fact that intermediate data states which occur after an arbitrary number of iterations are indistinguishable from data states which occur after 0 (or 1) loop iterations. To illustrate, consider the 2-closed initialized loop of Example 3a which sums the elements contained in a sequence. After some arbitrary number of iterations in an execution of this program, suppose we stop it and inspect the values of the program variables sum and seq.

Based on these values, what can we tell about the history of the execution? The answer is not too much; about all we can say is that if sum is not zero then we know we have previously executed at least 1 loop iteration, but the exact number of these iterations may be 1, 10 or 10000.

By way of contrast, again consider the integer multiplication program of Example 1, an initialized loop we know not to be N-closed for any N. Suppose we stop the program after an arbitrary number of iterations in its execution. Based on the values of the program variables z, v and k, what can we tell about the history of the execution? This information tells us a great deal; for example, we know the loop has iterated exactly z/k times and we can reconstruct each previous value of the variable z.

Initialized loops which have the information available to reconstruct their past have the <u>potential</u> to behave in a "tricky" manner. By "tricky" here, we mean performing in such a way that depends unexpectedly on the history of the execution of the loop (i.e. on the effect achieved by previous loop iterations). The result of this loop behavior would be a loop function which was "inconsistent" across all values of the loop inputs and which could only be inferred from the constraint functions with considerable difficulty. We consider this phenomenon more carefully in the following section; for now we emphasize that it is precisely the potential to behave in this unpleasant manner that is lacking in 1-closed and 2-closed initialized loops and which allows their general behavior to be described completely by the first 1 or 2 constraint functions.

5. 'Tricky' Programs

The above heuristic suggests inferring g from 2 subsets of that function, C1 and C2. Constraint function C2 is of particular importance since REWRITE and SUBSTITUTE are applied to this function and it, consequently, serves to guide the generalization process. C2 is based on the program specification f, the initialization and the input/output behavior of the loop body on its first execution. In any problem of inferring data concerning some population based on samples from that population, the accuracy of the results depends largely on how representative the samples are of the population as a whole. The degree to which the sample defined in C2 is is representative of the unknown general function we are seeking depends entirely on _how representative the input/output behavior of the loop body on the first loop iteration is of the input/output behavior of the loop body on an arbitrary subsequent loop iteration_.

To give the reader the general idea of what we have in mind, consider the program to count the nodes in a binary tree in Example 3. If the loop body did something peculiar when, for example, the set s contained 2 nodes with the same parent node, or when n had the value 15, the behavior of the loop body on its first execution would not be representative of its general behavior. By "peculiar" here, we mean something that would not have been anticipated based solely on input/output observations of its initial execution. An application of our heuristic on programs of this nature would almost certainly fail since (apparently) vital information would be missing from C1 and C2.

Example 9 - Consider applying the technique to the following program which is an alternative implementation of the integer multiplication program presented in Example 1:

```
{v>=0}
z := 0;
while v ≠ 0 do
    if z=0 then           z := k
    elseif z=k then       z := z * 2 * v
    else                  z := z - k fi;
    v := v - 1
    od
{z=$\overline{v0}$*k}.
```

The constraint functions C1 and C2 are identical to those for the program in Example 1 and we have no reason to infer a different function g. Yet this function is not only an incorrect hypothesis, it does not even come close to describing the general behavior of the loop. The difficulty is that the behavior of the loop body on its first execution is in no way typical of its general behavior. This is due to the high dependence of the loop body behavior on the input value of the initialized variable z.

We make the following remarks concerning programs of this nature. First, our experience indicates that they occur very rarely in practice. Secondly, because they tend to be quite difficult to analyze and understand, we consider them "tricky" or poorly structured programs. Thirdly, the question of whether the (input/output) behavior of the loop body on the first iteration is representative of its behavior on an arbitrary subsequent iteration is really a question of whether its behavior when the initialized variables have their initial values is representative of its behavior when the initialized variables have "arbitrary" values. Put still another way, the question is whether the loop body behaves in a "uniform" manner across the spectrum of possible values of the initialized data.

In practice, a consequence of a loop body exhibiting this uniform behavior is that there exists a simply expressed connection between different input values of the initialized data and the corresponding result produced by the WHILE loop. It is the existence of such a connection which motivates the SUBSTITUTE step above and which is thus a necessary precondition for a successful application of the technique. This explains its failure in dealing with programs such as that in Example 9. We make no further mention of these "tricky" programs, and in the following section discuss an informal categorization of "reasonable" programs and consider its implications for our loop function synthesis technique.

6. DU and TD Loops

In this section, we discuss general characteristics of many commonly occurring iterative programs. These characteristics are used to suggest two categories of these programs. This categorization is of interest since the above heuristic for synthesizing loop functions is particularly useful when applied to initialized loops in one of these categories.

In solving any particular problem, it often makes sense to consider certain instances of the problem as being "easier" or "harder" to solve than other instances. For example, with the problem of sorting a table, an instance of the problem for a table containing N elements might be harder to solve than an instance of the problem for a table containing N-1 elements. Similarly, if the problem is multiplying natural numbers, a*b might be easier to solve than (a+1)*b. This notion of "easier" and "harder" instances of a problem is particularly apparent for problems with natural recursive solutions. These solutions solve complex instances in terms of less complex instances and hence support the idea of one problem instance being easier to solve than another.

For the purpose of this discussion, we divide the data modified by the initialized loop under consideration into two sections: the accumulating data and the control data. The accumulating data is the specified output variable(s) of the loop. The remaining modified data is the control data and often serves to "guide" the execution of the loop and determine the point at which the loop should terminate. Both the accumulating data and the control data are typically (but not always) constrained by initialization in front of the loop.

A Heuristic For Deriving Loop Functions

Example 10a - In the program

```
{n>=0}
z := 1; t := 0;
while t ≠ n do
   t := t + 1;
   z := z * t
   od
{z=n!}
```

the variable z is the specified output of the loop and is hence the accumulating data. The other modified variable, t, is used to control the termination of the loop and is the control data.

In many cases, the control data can be viewed as representing an instance (or perhaps several instances) of the problem being solved. As the loop executes and the control data changes, the control data represents different instances of this problem. To illustrate, we can think of the control data t in the previous example as a variable describing a particular instance of the factorial problem. As the loop executes, the variable t takes on the values 0, 1, ..., n, and these values can be thought to correspond to the problems 0!, 1!, ... n!.

Based on these informal observations, we characterize a BU (from the Bottom Upward) loop as one where the control data problem instances are generated in order of increasing complexity, beginning with a simple instance and ending with the input problem instance to be solved. In the execution of a BU loop, the control data can be viewed as representing the "work" that has been accomplished "so far." We consider the factorial program above to be a BU loop. At any point in time, the "work" so far accomplished is t! and t moves from 0 (a simple factorial instance) to n (the input factorial instance).

Conversely, we characterize a TD (from the Top Downward) loop as one where the control data problem instances are generated in order of decreasing complexity, beginning with the input problem instance and ending with a simple problem instance. In the execution of a TD loop, the control data can be viewed as representing the "work" that remains to be done.

Example 10b - We consider the following alternative implementation of factorial to be a TD loop:

```
{n>=0}
z := 1; t := n;
while t ≠ 0 do
    z := z * t;
    t := t - 1
    od
{z=n!}.
```

As before z and t are the accumulating and control data respectively. The variable t moves from n (the input factorial instance) and ends with 0 (a simple factorial instance). After any iteration, the product n*(n-1)* ... *(t+1) has been accumulated, leaving t! as the "work" that remains to be done.

Example 11 - As an additional illustration, consider the following 3 initialized loops which compute integer exponentiation:

```
A: {y>=0}
   w:=1; t:=0;
   while t≠y do
      w := w*x;
      t := t+1
      od
   {w=x^y}
```

```
B: {y>=0}
   w:=1; t:=y;
   while t≠0 do
      w := w*x;
      t := t-1
      od
   {w=x^y}
```

```
C: {y>=0}
   w:=1; c:=x; t:=y;
   while t≠0 do
      if odd(t) then
         w := w*c fi;
      c:=c*c; t:=t/2
      od
   {w=x^y}
```

As before, the symbol ^ is used as an infix exponentiation operator. We consider program A to be a BU loop. The control data t moves from 0 to y and corresponds to the problem instances x^0, ..., x^y. On the other hand, B is TD since the control data t moves from y to 0 and corresponds to the problem instances x^y, ..., x^0. Program C (similar to that in Example 2) is slightly more difficult to analyze. The control data is the pair <c,t>. The pair is initialized to <x,y> and ends with the value <c',0>, where c' is some complex function of x and y. It seems reasonable to consider <c,t> as representing the problem c^t. Hence we conclude C is also TD. This conclusion also makes sense in light of the fact that C is really an optimized version of B which saves iterations by exploiting the binary decomposition of y.

The characterization of BU and TD loops described here is, of course, an informal one and depends largely on one's interpretation of the meaning or purpose of the control data. We classified the above programs by using what we considered to be the most "natural" or intuitive interpretation; other interpretations are always possible. Occasionally, two different interpretations of the control data seem equally valid and hence the program may be considered as either BU or TD, depending on one's point of view. For example, consider the following program which adds up the elements in a subarray between indices p1 and p2:

```
sum := 0; i := p1;
while t <= p2 do
   sum := sum + a[i];
   i := i + 1
   od
{sum=ASUM(a[p1..p2])}.
```

The notation ASUM(a[p1..p2]) appearing in the postcondition stands for the summation of the elements in the indicated subarray. The question which arises in attempting to classify this program is as follows: as the control data i moves through the values p1, p1+1, ..., p2, is it most appropriate to think of it as representing the problem instance which has been solved (i.e. ASUM(a[p1..i])) or as representing the problem instance which remains to be solved (i.e. ASUM(a[i..p2])). Both views seem equally intuitive, that is, the program seems to be as much BU as it is TD.

As a final example, we refer back to the program in Example 3 which counts the nodes in a binary tree. It is clear n and the set variable s are the accumulating and control data respectively. Initially, s contains the tree whose nodes are to be counted; when the program terminates s is empty. In between, s contains various subtrees of the original tree. It seems natural to view the set as containing progressively simpler and simpler instances of the NODES problem since the trees in s consist of fewer and fewer nodes as the loop executes. Thus we classify the program as a TD loop.

A Heuristic For Deriving Loop Functions

We have seen that the problem solving method taken by a BU loop is one of approaching the general problem instance from some simple problem instance. Of course, this problem solving method is reasonable only when there exists some technique whereby one is guaranteed to "run into" the general problem instance. Our view is that in many cases, such a convergence technique either does not exist or requires so much support that the BU approach is not practical. This appears to be particularly true for programs dealing with sophisticated data types (i.e. something other than integers) and for programs requiring a high degree of efficiency in their number of iterations.

To help see this point of view, again consider the NODES program of Example 3. Previously we argued that this was a TD program. What would a BU program which computed the same function look like? The following program skeleton suggests itself:

```
n := 0;  tl := "an empty tree";
while tl ≠ t do
   "add a node to tl to make it look more like t";
   n := n + 1
   od
{n=NODES(t)}.
```

Here, the tree variable tl is the control data and it represents the problem NODES(tl). The difficulty with this attempt at a program solution is the implementation of the modification of tl. Such a modification requires close inspection (i.e. a traversal) of t in order to move tl toward t. In light of this, it seems more reasonable to count the nodes of t while it is being inspected and to dispense altogether with the variable tl.

A Heuristic For Deriving Loop Functions

As an illustration of another circumstance where the BU approach seems unreasonable, the reader is encouraged to imagine a BU implementation of integer exponentiation which operates as efficiently as the exponentiation program C from Example 11. Again, a program skeleton suggests itself:

```
{y>=0}
w := 1;  c := ?;  d := 0;
while <c,d> ≠ <x,y> do
   c := sqrt(c);
   if ? then
      d := d * 2 + 1; w := w * c
   else d := d * 2 fi
   od
{w=x^y}.
```

Here, we are attempting to move the control data <c,d> toward <x,y> as fast as we moved it away from <x,y> in TD program C. As with the BU NODES program, the problem here is how to complete the program so as to achieve the desired effect. Our conclusion concerning this program is that supplying an appropriate initial value for c and determining the proper loop body path to be executed requires such complexity that this approach is not a feasible alternative to program C.

In this section we have suggested two informal categories of initialized loop programs. We offered the opinion that the approach taken in a BU program solution has rather limited applicability and that TD programs tend to occur more frequently in practice. We feel that this characterization is useful as a study of opposing problem solving philosophies but our main source of motivation is to investigate the kinds of commonly occurring programs on which the loop function synthesis technique described above works well.

Consider applying this technique to a general TD program. In the second constraint function, the control data is bound to a value which represents a slightly less complex instance of the general problem being solved by the initialized loop. In practice, the appearance of this value in the constraint function suggests the problem decomposition being exploited by the programmer in order to achieve the program result. Applying this decomposition in REWRITE leads quite naturally to the desired general loop function.

Example 12 - Consider the TD factorial program from Example 10b. The second constraint function is

```
  C2:  n>0, z=n, t=n-1 -> g(z,t)=n!
```

The control data t being bound to n-1 suggests REWRITING n! as n*(n-1)!. This leads to the correct general loop function. On the other hand, consider the second constraint function for the TU factorial program from Example 10a:

```
  C2:  n>0, z=1, t=1 -> g(z,t,n)=n!
```

How can the expression n! be rewritten in terms of 1, 1 and n? To obtain the correct general function, the expression would have to be rewritten as (1*n!)/(1!) which seems much less intuitive than that required for the TD version. As another point of comparison, consider the second constraint function for the TD exponentiation program 3 from Example 11:

```
  C2:  y>0, v=x, t=y-1 -> g(v,t,x)=x^y
```

and the second constraint function for the BU exponentiation program A from the same example:

```
  C2:  y>0, w=x, t=1 -> g(w,t,x,y)=x^y.
```

In both cases, the proper loop function may be obtained by using the REWRITE rule x^y = x*(x^(y-1)); however, this particular rule seems more strongly suggested in the constraint function for the TD program.

We remark that the same general phenomenon occurs with TD programs in the event the control data has been SIMPLIFIED out of the domain requirement for C2. In this case, the fact that the control data represents a slightly less complex instance of the general problem being solved manifests itself in the function expression for the SIMPLIFIED C2 being a slightly more complex instance of the problem being solved. For example, the constraint function C2 above for the TD exponentiation program B of Example 11 can be SIMPLIFIED to

```
        t>=0, w=x -> g(w,t,x)=x^(t+1).
```

Before, the appearance of y-1 in the domain requirement suggested rewritting x^y as x*(x^(y-1)). Here, the appearance of t+1 in the function expression suggests rewritting x^(t+1) as x*(x^t) (see also Examples 1 and 2).

Suppose f is the operation or function the initialized loop program is intended to compute. In Section 3 we observed that each REWRITE in the examples of that section involved "decomposing" an application of f. This decomposition corresponds to rewritting that problem instance in terms of a slightly less complex problem instance (or instances). In general, of course, there are many ways this decomposition can be performed. In the examples of that section, however, as with all TD programs, the nature of the control data guides this decomposition and thus tends to make the REWRITE step quite straightforward in practice.

The reader may have noticed that the general loop functions for the BU factorial and exponentiation programs contain more program variables and operations on those variables than their TD counterparts. For instance, the general loop functions for the BU and TD factorial programs are

```
     0<=t<=n -> g(z,t,n)=z*(n!/t!)
```

and

```
     0<=t -> g(z,t)=z*t!
```

respectively. This fact, by itself, helps explain why the REWRITE step seems more difficult for the BU programs. It would be a mistake, however, to assume that the BU programs are more "complex" or are more difficult to analyze or prove. We consider TD loops to be somewhat more susceptible to the form of induction employed in functional loop verification. More precisely, the inductive hypothesis required in this type of proof (i.e. a general statement concerning the loop input/output behavior) seems to be more easily stated for TD programs than for BU programs. On the other hand, BU programs seem somewhat more susceptible to an inductive assertion proof. The inductive hypothesis required in this type of proof (i.e. a sufficiently strong loop invariant) involves fewer program variables and operations on those variables than the same type of hypothesis for the corresponding TD loop. As an example, the BU and TD factorial programs have adequate loop invariants 0<=t & z=t! and 0<=t<=n & z=n!/t! respectively.

In [Manna & Waldinger 70], the authors describe a program synthesis technique and point out that their method produces either of the above factorial programs depending upon which type induction rule the synthesizer is given to employ.

7. Related Work

In [Basu & Misra 76, Misra 73, Misra 79], the authors describe two classes of "naturally provable" programs for which generalized loop specifications can be obtained in a deterministic manner. Our technique sacrifices determinism in favor of wide applicability and ease of use. It handles in a fairly straightforward manner typical programs in these two program classes (e.g. Examples 1-3) as well as a number of programs which do not fit in either of the classes (e.g. Examples 4-6).

Due to the close relationship between loop functions and loop invariants (see, for example, [Morris & Wegbreit 77]), any technique for synthesizing loop invariants can be viewed as a technique for synthesizing general loop functions (and vice versa). In this light, our method bears an interesting resemblance to a loop invariant synthesis technique described in [Wegbreit 74, Katz & Manna 76]. In this technique stronger and stronger "approximations" to an adequate loop invariant are made by pushing the previous approximation back through the loop once, twice, etc.

By way of illustration, consider the exponentiation program of Example 2. The loop exit condition can be used to obtain an initial loop invariant approximation

    d=0 -> w=c0^d0.

This approximation can be strengthened by pushing it back through the loop to yield

    (d=0 -> w=c0^d0) & (d=1 -> w*c=c0^d0).

In the analysis presented in Example 2, we obtained a value for the generalized function specification for each of two different values of the initialized variable w (i.e. 1 and SQRT(c)); here we have obtained a "value" for the loop invariant we are seeking for each of two different values of the variable which controls the termination of the loop d. Applying the analysis in [Morris & Wegbreit 77], these loop invariant "values" can be translated to constraint functions as follows:

    d=0 -> g(w,c,d)=w,
    d=1 -> g(w,c,d)=w*c.

Of course, the function expression w*c in the second constraint can be rewritten w*(c^1); SUBSTITUTING as usual suggests the general loop function

        g(w,c,d)=w*(c^d).

If we then add the program precondition as a domain restriction on this function, the result is the same general loop function discovered in Example 2.

A Heuristic For Deriving Loop Functions

We summarize the relationship between these two techniques as follows. As the initialized loop in question operates on some particular input, let X[0], X[1], .. ,X[N] be the sequence of states on which the loop predicate is evaluated (i.e. the loop body executes N-1 times). Of course, in X[0], the initialized variables have their initialized values, and in X[N], the loop predicate evaluates to FALSE. The method proposed in this paper suggests inferring the unknown loop function g from X[0], X[1], g(X[0]) and g(X[1]). The loop invariant technique described above, when viewed as a loop function technique, suggests inferring g from X[N], X[N-1], g(X[N]) and g(X[N-1]). Speaking roughly then, one technique uses the first several executions of the loop, the other uses the last several executions. One ignores the information that the loop must compute the identity function on inputs where the loop predicate is FALSE, the other ignores the information that the loop must compute like the initialized loop when initialized variables have their initialized values.

Earlier we discussed "top down" and "bottom up" approaches to synthesizing g and indicated that our technique fit in the "top down" category. The technique based on the last several iterations is a "bottom up" approach. It is difficult to carefully state the relative merits of these two opposing techniques. In our view, however, there are a number of circumstances under which the technique based on the first several loop executions seems more "natural" and easily applied. These examples include the NODES program, the program to compute Ackermann's function and the TD factorial program discussed above. The reason is that a critical aspect of the general loop function is the function computed by the initialized loop program (e.g. exponentiation in the above illustration). In the technique based on the first several iterations, this function appears explicitly in the constraint functions. In the other technique, this information must somehow be inferred from the corresponding constraint functions (e.g. by looking for a pattern in these functions, etc.). This difficulty is inherent in any "bottom up" approach to synthesizing g.

8. Concluding Remarks

In this paper we have proposed a technique for deriving functions which describe the general behavior of a loop which is preceded by initialization. These functions can be used in a functional [Mills 75] or subgoal induction [Morris & Wegbreit 77] proof of correctness of the initialized loop program. It is not our intention to imply that verification should occur after the programming process has been completed. There are, however, a large number of existing programs which must be read, understood, modified and verified by "maintenance" personnel. We offer the heuristic as a tool which is intended to facilitate these tasks.

It has been argued [Misra 78] that the notion of closure of a loop with respect to an input domain is fundamental in analyzing the loop. In Section 4, this idea is applied to initialized loop programs. The result is that a loop function g for a loop which is N-closed (for some $N>0$) can be synthesized in a deterministic manner by considering the first N constraint functions. Hence this categorization can be viewed as one measure of the "degree of difficulty" involved in verifying initialized loop programs.

An interesting direction for future research is the development of a precise characterization of programs which are not "tricky" (as discussed in Section 5). Preliminary results along this line are described in [Dunlop & Basili 81] (see also [Basu 80]).

In Section 6 we discussed on an informal level the opposing BU and TD problem solving strategies and their corresponding initialized loop realizations. We argued that the TD approach appeared to be more widely applicable and that, in practice, TD programs seem to occur more frequently. We explained the success of the proposed loop function creation technique on these programs in terms of an easily applied REWRITE step. These results are offered to help support our view that the technique may be successfully applied in a wide range of applications.

9. References

[Basu 80]
Basu, S. A Note on Synthesis of Inductive Assertions, *IEEE Transactions on Software Engineering*, SE-6 (January, 1980).

[Basu & Misra 75]
Basu, S. and Misra, J. Proving Loop Programs, *IEEE Transactions on Software Engineering*, SE-1 (March, 1975).

[Basu & Misra 76]
Basu, S. K. and Misra, J. Some Classes of Naturally Provable Programs, *Proc. 2nd International Conf. on Software Engg.*, San Francisco, Oct. 1976.

[Dunlop & Basili 81]
Dunlop, D. and Basili, V. Generalizing Specifications for Uniformly Implemented Loops, University of Maryland Computer Science Center Technical Report TR-1116, September 1981.

[Ellozy 81]
Ellozy, H. The Determination of Loop Invariants for Programs with Arrays, *IEEE Transactions on Software Engineering*, SE-7 (March 1981), pp. 197-206.

[Gries 79]
Gries, D. Is Sometime Ever Better Than Alway?, *Transactions on Programming Languages and Systems*, Vol. 1, No. 2, Oct 1979.

[Hoare 69]
Hoare, C. A. R. An Axiomatic Basis for Computer Programming, *CACM*, 12 (October 1969), pp. 576-583.

[Katz & Manna 73]
Katz, S. and Manna, Z. A Heuristic Approach to Program Verification, *Proc. 3rd Int. Joint Conf. Artificial Intell.*, Stanford, CA 1973, pp. 500-512.

[Katz & Manna 76]
Katz, S. and Manna, Z. Logical Analysis of Programs, *CACM* 19 (April 1976), pp. 188-206.

[Manna & Waldinger 70]
Manna, Z. and Waldinger, R. Towards Automatic Program Synthesis, Stanford Artificial Intelligence Project, Memo AIM-127, July 1970.

[Mills 72]
Mills, H. D. Mathematical Foundations for Structured Programming, IBM Federal Systems Division, FSC 72-6012 (1972).

[Mills 75]
Mills, H. D. The New Math of Computer Programming, CACM, 18 (January 1975).

[Misra 78]
Misra, J. Some Aspects of the Verification of Loop Computations, IEEE Transactions on Software Engineering, SE-4 (November 1978), pp. 478-486.

[Misra 79]
Misra, J. Systematic Verification of Simple Loops, University of Texas Technical Report TR-97, March 1979.

[Morris & Wegbreit 77]
Morris, J. H. and Wegbreit, B. Subgoal Induction, CACM 20 (April 1977), pp. 209-222.

[Wegbreit 74]
Wegbreit, B. The Synthesis of Loop Predicates, CACM 17 (February 1974), pp. 102-112.

[Wegbreit 77]
Wegbreit, B. Complexity of Synthesizing Inductive Assertions, JACM, Vol. 24 (July 1977), pp. 504-512.

Uncl

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR-TR- 81 -0793	2. GOVT ACCESSION NO. AD-A108660	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A HEURISTIC FOR DERIVING LOOP FUNCTIONS		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER TR-1115
7. AUTHOR(s) Douglas D. Dunlop and Victor R. Basili		8. CONTRACT OR GRANT NUMBER(s) [redacted]-F49620-80-C-0001
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Computer Science University of Maryland College Park, Maryland 20742		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61102F 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS Math & Info Sciences, AFOSR Bolling AFB Washington, D. C. 20332		12. REPORT DATE October 1981
		13. NUMBER OF PAGES 50 plus title & abstract pgs.
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

program verification, initialized loop programs, loop functions, constraint functions, BU programs, TD programs

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The problem of analyzing an initialized loop and verifying that the program computes some particular function of its inputs is addressed. A heuristic technique for solving these problems is proposed which appears to work well in many commonly occurring cases. The use of the technique is illustrated with a number of applications. A hierarchy of initialized loops is suggested which is based on the "effort" required to apply this methodology in a deterministic (i.e. guaranteed to succeed) manner. It is explained that in any case, the success of the proposed heuristic relies on the loop exhibiting a "reasonable" form of behavior. An informal categorization

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

52

of such programs is made which is based on two opposing problem solving strategies. It is suggested that our heruistic is naturally suited for use on programs in one of these categories.

FILMED
8