Acta Informatica 5, 333—352 (1975)
© by Springer-Verlag 1975

A Closer Look at Termination

Shmuel Katz and Zohar Manna

Received October 16, 1974

Summary. Several methods for proving that computer programs terminate are presented and illustrated. The methods considered involve (a) using the "no-infinitely-descending-chain" property of well-founded sets (Floyd's approach), (b) bounding a counter associated with each loop (*loop* approach), (c) showing that some exit of each loop must be taken (*exit* approach), or (d) inducting on the structure of the data domain (Burstall's approach). We indicate the relative merit of each method for proving termination or non-termination as an integral part of an automatic verification system.

Introduction

In recent years a considerable number of verification systems for proving correctness of computer programs have been developed (e.g., [7, 12, 15, 19]) but, surprisingly, very few of these try to treat the problem of termination. (One of the interesting exceptions is the work of Cooper [6].) A program is said to *terminate* if for all legal input values the execution of the program will eventually reach a HALT statement. In this paper we give an overview of several possible methods for proving termination, and indicate which method seems to us to be most compatible with automatic verification systems.

In Section 1, we outline the classic *Floyd technique* [10] for proving termination, which uses the "no-infinitely-descending-chain" property of well-founded sets. We demonstrate two possible directions for overcoming some difficulties in practically applying the method.

In Section 2, we introduce a *loop approach* to proving termination. In this approach, we associate a counter with each loop, reflecting the number of times the loop has been executed, and show that all the counters are absolutely bounded from above. (A similar technique has been suggested by Elspas [8].)

In Section 3, an *exit approach* is defined, where termination is shown by directly proving that for each loop the conditions for exiting the loop must be true at some stage of the computation (see also Sites' Ph.D. thesis [18]).

Finally, in Section 4, we illustrate the possibility of proving termination along with correctness by using a technique suggested by Burstall [3]. In this technique, we show that if some property p_A is assumed at a point A (in particular, the START point), we must *eventually* reach another point B (in particular, a HALT point), with some property q_B true. This is shown by induction on the possible values of the data domain.

In each section we try to point out briefly the advantages and the disadvantages of each method. As indicated in Section 2, we consider the loop approach to be the method for proving termination which can be most easily integrated into an automatic verification system. This method also provides the greatest information about the complexity and control behavior of the program.

Since we may not assume *a priori* that the program actually terminates, any automatic verification system should also attempt to prove non-termination of a program which loops forever for some input value. In Section 3 we claim that the exit approach, while of limited use in proving termination, is actually the natural way to prove non-termination.

1. Floyd's Method

The traditional method of proving termination, which was suggested by Floyd [10], makes use of a well-founded set $(W, \succ)$ with a partial ordering $\succ$ having the property that there is no infinitely descending chain of elements from W, i.e., any chain of the form $w_1 \succ w_2 \succ \cdots$ must be finite. The procedure requires finding a set of cutpoints which cut every loop of the program at least once. Then for each cutpoint A, a partial function u_A and an assertion q_A must be chosen. The function u_A maps elements of the program's data domain into W, while q_A serves to restrict the domain of u_A. The assertion q_A must be true each time the cutpoint A is reached (and thus is called an *invariant*); it indicates a set of values of the data domain that includes all those values that can be reached at A during the execution of the program. The proof of termination consists of showing that $u_A \succ u_B$ each time control moves along a simple path (which is a part of a loop), from a cutpoint A to a cutpoint B. A path is *simple* if it contains no other cutpoints.). Thus clearly no loop or combination of loops could be executed indefinitely because the no-infinitely-descending-chain condition would be violated.

In the above method the actual proof of termination is generally mechanical once the proper choices of a well-founded set $(W, \succ)$, cutpoints $\{A\}$, functions $\{u_A\}$, and assertions $\{q_A\}$ have been made. In fact, considerable progress has been achieved recently in automatically finding the invariants of a program, and there are several existing or proposed systems for this purpose (e.g., [4, 9, 11, 14, 17, 20]). Thus the main remaining requirement for an automation of Floyd's technique for proving termination is a systemization of techniques for finding the well-founded set $(W, \succ)$ and the functions $\{u_A\}$. Unfortunately, making the correct choice of the functions $\{u_A\}$ is a difficult task qualitatively different from the discovery of the invariants $\{q_A\}$. In the following example we demonstrate one possible heuristic which sometimes can be profitably applied to yield such functions.

Example 1 (Floyd's approach). The program in Fig. 1 is a flowchart version of McCarthy's "91-function". It computes the function

$$z = \textbf{if } x > 101 \textbf{ then } x - 10 \textbf{ else } 91$$

over the integers. We will consider only termination.

For convenience we will call the path around the loop which is taken when $y_1 \leqq 100$, the *left* path, and the path around the loop which is used when $y_1 > 100$ and $y_2 \neq 1$, the *right* path. We choose point A as the cutpoint which cuts both paths around the loop. Let us take the set N of all natural numbers, with the regular $<$ ordering, as the well-founded set. We might initially try to show that

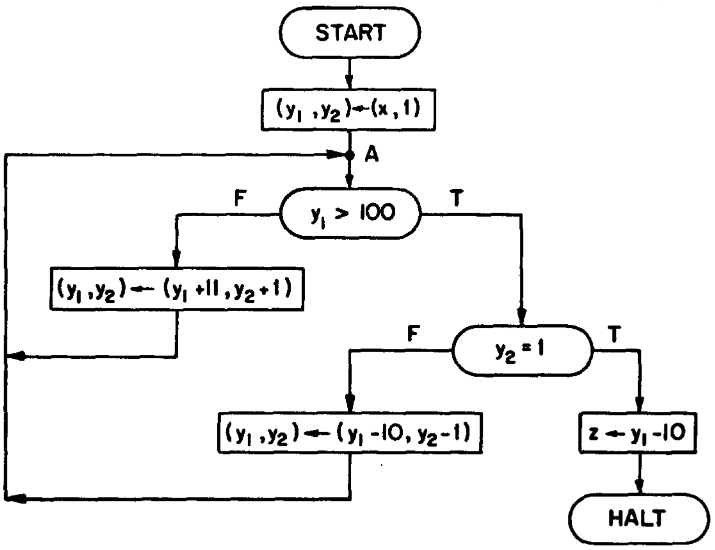


Fig. 1. The "91-function" program

either y_1 or y_2 alone are strictly monotonic, and are bounded. A glance at the program will show that such an attempt would fail since the two variables both increase and decrease in the loop.

As a next heuristic step, we assume that a *linear* function involving y_1 and y_2 is required. That is, that u_A has the form

$$\alpha \cdot y_1 + \beta \cdot y_2 + \gamma$$

for some constants α, β, and γ. By considering the two paths around the loop, and the requirement that there be a drop in the value of u_A, we can see that α, β, and γ must fulfill

$$\alpha \cdot y_1 + \beta \cdot y_2 + \gamma > \alpha \cdot (y_1 + 11) + \beta \cdot (y_2 + 1) + \gamma \cdots \quad \text{for the left path}$$

and

$$\alpha \cdot y_1 + \beta \cdot y_2 + \gamma > \alpha \cdot (y_1 - 10) + \beta \cdot (y_2 - 1) + \gamma \cdots \quad \text{for the right path.}$$

Thus we have obtained a set of inequalities.

Simplifying, we have

$$0 > 11 \cdot \alpha + \beta$$

and

$$0 > -10\alpha - \beta.$$

These may be solved; one (integer) solution is $\alpha = -2$, $\beta = 21$. Thus we have found that for any u_A of the form $-2y_1 + 21y_2 + \gamma$, there will be a drop in the value of the (integer-valued) functions each time the loop is executed.

In order to show that the resultant sequence is well-founded, we would like to choose the non-negative integers N as the domain W and fix γ so that the values of u_A will always be non-negative. For this purpose, we seek an upper bound a on

y_1 and lower bound b on y_2. Using known invariant-generating techniques, it is possible to find that

$$(y_1 \leqq 111 \wedge y_2 \geqq 1) \vee (y_1 = x \wedge y_2 = 1)$$

is an invariant at A. Thus, $a = \max(111, x)$ and $b = 1$. Therefore the smallest possible value of the function is $-2 \cdot \max(111, x) + 21 + \gamma$, and a sufficient γ to guarantee that the function is always non-negative is $\gamma = 2 \cdot \max(111, x)$. We have thereby obtained the function

$$u_A\colon -2y_1 + 21y_2 + 2 \cdot \max(111, x).$$

Note that the heuristic of assuming a linear u_A was crucial to the development. □

In the following example we illustrate another common problem which involves the complexity of the required functions $\{u_A\}$. As it stands, in order to prove termination using Floyd's method, a drop must be shown along every simple path from a cutpoint to another cutpoint (which is on a loop). This often makes the choice of functions very sensitive to the placement of the cutpoints, and requires adding unnatural components to the functions $\{u_A\}$ in order to ensure a drop. As we demonstrate, this difficulty may be overcome by slightly generalizing Floyd's method, showing that for every possible path simple or not from a cutpoint, there will eventually be a drop in the function.

Example 2 (Floyd's approach). The program in Fig. 2 computes the greatest common divisor (g.c.d.) of two positive integers x_1 and x_2. Since this program consists of two inner loops and an outer loop, it is natural to choose A, B, and C as the cutpoints. If we use the original Floyd method with these cutpoints, a typical set of functions[1] is

u_A: $(y_1 + y_2, 2)$

u_B: **if** $y_1 \neq y_2$ **then** $(y_1 + y_2, 1)$ **else** $(y_1 + y_2, 4)$

u_C: **if** $y_1 < y_2$ **then** $(y_1 + y_2, 0)$ **else** $(y_1 + y_2, 3)$

where the well-founded set is the set of all pairs of non-negative integers with the lexicographical ordering[2]. The needed invariants at A, B, and C are $y_1 > 0$ and $y_2 > 0$.

There will be a drop in the path from B to C, for example, because the path condition $y_1 \leqq y_2$ implies that either $y_1 < y_2$, so that u_B is $(y_1 + y_2, 1)$ and u_C is $(y_1 + y_2, 0)$, or $y_1 = y_2$, so that u_B is $(y_1 + y_2, 4)$ but u_C is $(y_1 + y_2, 3)$. Similarly, the path condition $y_2 \leqq y_1$ for taking the path from C to A implies that whenever this path is followed u_C is $(y_1 + y_2, 3)$ (because $y_2 > y_1$ is false), and so there is a drop to $(y_1 + y_2, 2)$. For the path around the first inner loop, from B back to B, we use the invariant $y_2 > 0$ at B to show that the function value drops because the first component always descends from $y_1 + y_2$ to y_1 even though the second component may increase from 1 to 4.

1 These functions were suggested by Martin Fürer.

2 That is, $(\alpha_1, \alpha_2) < (\beta_1, \beta_2)$ in the lexicographical ordering iff $\alpha_1 < \beta_1$ or, $\alpha_1 = \beta_1$ and $\alpha_2 < \beta_2$.

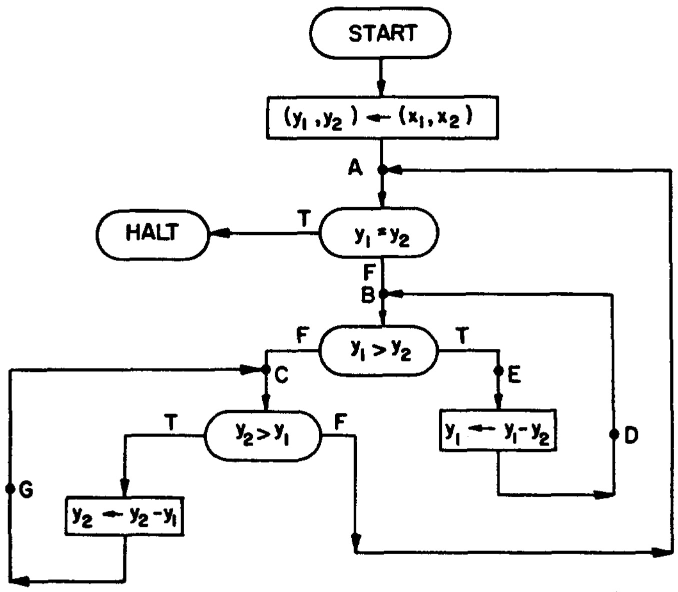

START
$(y_1, y_2) \leftarrow (x_1, x_2)$
A
HALT
T
$y_1 = y_2$
F
B
F
$y_1 > y_2$
T
C
E
T
$y_2 > y_1$
F
$y_1 \leftarrow y_1 - y_2$
D
G
$y_2 \leftarrow y_2 - y_1$

Fig. 2. *g.c.d.* program

On the other hand, choosing cutpoints E and C allows using the far simpler functions

u_E: $(y_1+y_2, 0)$

u_C: $(y_1+y_2, 1)$

where W is again the set of all pairs of non-negative integers with the lexicographical ordering.

Finally, if we choose cutpoints D and G, then

u_D: y_1+y_2,

u_G: y_1+y_2

are sufficient, where W is the set of all natural numbers. In this case, it is necessary to note that we have cut only every *possible* path around the loops. The "impossible" path around the outer loop $A-B-C-A$ (which does not execute either inner loop) is not cut; but since this cannot occur, the set of cutpoints is nevertheless adequate.

A generalized version of Floyd's method which is less sensitive to the placement of the cutpoints will now be used. We will prove that for each cutpoint i there will eventually be a drop in the value of the function at some cutpoint j along every path from i, ignoring intermediate values. The advantage of this generalization is that simpler functions can often be used, but the penalty is that more paths must be treated.

This approach enables us to prove termination by considering the cutpoints A, B, and C with u_A, u_B, and u_C being y_1+y_2[3]. In order to show a drop somewhere

3 The invariants needed are still $y_1 > O$ $y_2 > O$ at A, B and C.

along every non-simple possible path from A, we consider three cases: (a) For all paths which begin $A-B-B-\cdots$ (i.e., from A to B, and then do the first inner loop at least once), the *second* time B is reached u_B is $y_1-y_2+y_2=y_1$ (relative to y_1 and y_2 at A). This value is always smaller than y_1+y_2 because $y_2>0$. (b) For all paths which begin $A-B-C-C-\cdots$ (i.e., do not include the upper inner loop, but do at least one circuit around the lower inner loop) the second time C is reached u_C is $y_1+y_2-y_1=y_2$. This value is again smaller than y_1+y_2 because $y_1>0$. (c) Finally, we note that any path which begins $A-B-C-A-\cdots$ (i.e., from A to A, without doing either inner loop) cannot be executed because the condition for following such a path is that $y_1 \neq y_2$, $y_1 \leqq y_2$, and $y_2 \leqq y_1$ are all true, which is impossible.

Similar reasoning can be used to show an eventual drop for every path from B, and from C, thus the program must terminate. □

As a final example, we bring a more typical program, where termination is not based on any complicated tricks, and the variables which control termination are basically counters. The example illustrates how the functions $\{u_A\}$ can be chosen in the case of a more complicated nested loop structure.

Example 3 (Floyd's approach). The program in Fig. 3 computes the determinant $|X[a, b]|$ of order M, $M \geqq 1$, by Gaussian elimination. We choose the three cutpoints A, B, and C.

This program has three loops, where α is the top loop controlled by the variable a, β is the middle loop controlled by b, and γ is the bottom loop controlled by c. Loop α can be said to "dominate" β and γ because a is not changed in β or γ. Similarly, β dominates γ because b is not changed in γ. This suggests using the triples of non-negative integers with the lexicographical ordering as the well-founded set, with a leftmost component for α, a middle component for β, and a right component for γ. The functions over N^3 can be

$$u_A\colon\ (M-a,\ M+1,\ M+1),$$
$$u_B\colon\ (M-a,\ M+1-b,\ M+1),$$
$$u_C\colon\ (M-a,\ M+1-b,\ c).$$

The functions include M and $M+1$ either in order to guarantee a drop along the paths from A to B and from B to C, or to guarantee that the values are non-negative. The ordering of the components is clearly important. For example, along the path from C to B the value of the third component increases, but the second component decreases.

The invariants needed to guarantee that there will indeed be a drop from cutpoint to cutpoint, and that the values of each component are non-negative integers, are

$$q_A\colon\ 1 \leqq a \leqq M,$$
$$q_B\colon\ 1 \leqq a < M \wedge b \leqq M+1,$$
$$q_C\colon\ 1 \leqq a < M \wedge b < M+1 \wedge a \leqq c.$$

Such invariants can all be generated automatically using existing methods. □

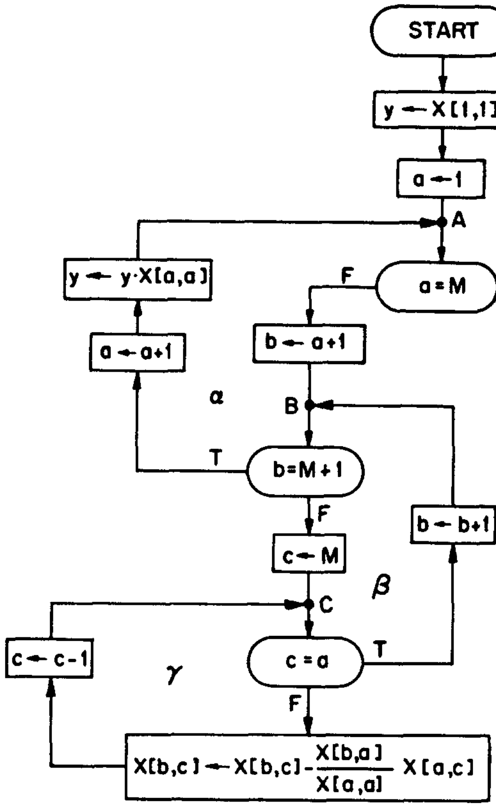


Fig. 3. Program for evaluating the determinant $z = |X|$.

2. The Loop Approach

Another solution to the problem of proving termination systematically is to rely more on the invariants than on the functions. This shift of emphasis should allow the methods we suggest to take full advantage of the progress in finding invariants automatically. In order to facilitate this process, it is also convenient to consider loops as the basic entities, rather than paths between cutpoints.

In the remainder of this paper we will assume that the loops of a program have been identified. Algorithms for this task can be found in [1] or [2]. We further assume that for the programs we treat, the loops can be enclosed in *blocks*, such that every block contains at most one top-level loop, ignoring lower-level loops which are possibly contained in inner blocks. There is exactly one entrance to each block, and one or more exits. We then associate one cutpoint with each block so that its top-level loop will be cut.

Note that a top-level loop may actually consist of several looping paths, obtained by test branching and join points, but sharing a single cutpoint.

We use *counters* as an essential tool for this technique. With each cutpoint, and thus each block with a loop, we associate a counter. The counter must be initialized before entering the block so that its value is zero upon first reaching the cutpoint, and incremented exactly once by one along the top-level loop inside the block before control returns again to the cutpoint.

There are many locations where the counters could be initialized to zero. The two extreme possibilities are of special interest: (a) the counter is initialized only once, at the beginning of the program (a "global" initialization, parameterizing the total number of times the cutpoint is reached), or (b) the counter is initialized just before entering its block (a "local" initialization, indicating the number of executions of the corresponding loop since the most recent entrance to the block).

These counters will serve a dual purpose:

(1) We may indicate the values of the program variables in terms of the counters. For example, for a cutpoint A with counter n, $y(n_0)$ indicates the value of the variable y when cutpoint A is reached with $n=n_0$. (If there is no way of reaching A with $n=n_0$, then $y(n_0)$ is undefined.)

(2) We may also denote relations among the number of times various paths have been executed. For example, an invariant $i>j$ at cutpoint A means that whenever control reaches A, the statements adjoining counter i have been executed more often than those adjoining counter j. Similarly, $i \leqq r$, for fixed r, means that the statements adjoining counter i will not be executed more than r times.

For convenience we shall assume that every invariant involving counters implicitly contains the information that they are non-negative integers.

Variables which are not part of the program but are useful and even necessary in order to prove properties of the program have been used previously by several researchers (e.g., [5]). Knuth [16] uses a 'time clock' incremented before every statement in order to prove termination. We found that such "implicit" variables are virtually indispensable whenever it is necessary to discuss how the control moves along various paths through the program. For this reason, we often use additional auxiliary counters in order to facilitate a proof of termination.

The *loop approach* depends upon the fact that a counter at cutpoint A indicates the number of times the control has passed A (as mentioned above, either globally or locally). Thus, if we are able to show for each block that its counter is absolutely bound from above at the cutpoint of the block, then the program must terminate. Proving termination becomes equivalent to finding invariants of the loop which guarantee that for each cutpoint A, its counter i has a fixed upper bound r at A[4]. In effect, for a single loop we have added counters and then adopted a particular case of Floyd's method, with W the non-negative integers and r the upper bound needed to establish that u_A: $r-i$ assumes values in N. The advantage gained is that a program to generate invariants for proving correctness may simultaneously produce invariants which are useful for proving termination by this method. In addition, invariants involving the counters are often useful for proving correctness as well.

[4] The bound r may be expressed in terms of constants and any variables which are not changed in that block.

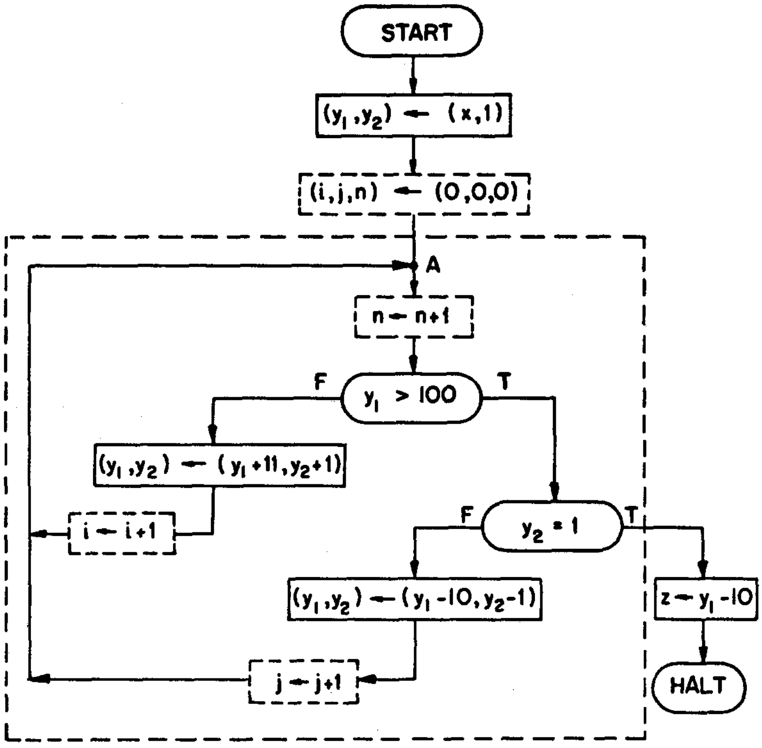


Fig. 4. The "91-function" program (with counters)

Example 1 (loop approach). We again consider the 91-program, this time proving termination by the loop approach (see Fig. 4). There is only one block which includes one top-level loop (with two alternative paths) that is cut by cutpoint A. We associate the counter n with the cutpoint. Furthermore, it is convenient to use the two additional counters i and j, as indicated in Fig. 4. We want to find invariants at A which will establish a bound on n. However, since $n=i+j$ at A, we shall first look for bounds on i and j.

The counters i and j allow us to express at cutpoint A the obvious invariant

$$y_2=i-j+1 \tag{1}$$

(because y_2 is increased by 1 each time i is increased, and decreased by 1 each time j is increased, and y_2 is initially 1). Similarly, we obtain

$$y_1=11i-10j+x. \tag{2}$$

We seek bounds on y_1 and y_2 which will allow us to bound i and j by the loop approach. We have

$$y_2\geqq 1 \tag{3}$$

(because y_2 is initially 1, is increased on the left path, and can be decreased by 1 on the right path only when its value is greater than 1).

Combining (1) and (3) we obtain the invariant

$$i \geqq j \tag{4}$$

i.e., the right path around the loop cannot be executed more often than the left.

By using (4) with (2) we obtain both

$$y_1 \geqq 11i - 10i + x \geqq i + x \tag{5}$$

and

$$y_1 \geqq 11j - 10j + x \geqq j + x \tag{6}$$

(depending on whether we substitute i for j, or j for i).

It is clear that

$$n = i + j \tag{7}$$

at A. This is a typical "structural invariant", i.e., an assertion which contains only counters and is dependent only on the structure of the graph of the flow-chart.

From (5) and (6) we may obtain $2y_1 \geqq i + j + 2x$, and by (7)

$$y_1 \geqq n/2 + x. \tag{8}$$

We now would like to bound y_1 from above. Initially we reach A with $y_1 = x \wedge y_2 = 1 \wedge n = 0$. If the left path is then taken, $y_1 \leqq 111$ after completing it, and this will then remain true at A. If the $y_1 > 100$ branch is taken initially, the program will immediately terminate. Thus at A we have

$$(y_1 = x \wedge y_2 = 1 \wedge n = 0) \vee y_1 \leqq 111. \tag{9}$$

If we let the invariant q_A be the conjunction of (8) and (9), then

$$q_A \supset [n = 0 \vee n/2 + x \leqq 111].$$

Thus the counter n is absolutely bounded at A and the program must terminate. $\square$

Example 2 (loop approach). We apply the loop approach to the *g.c.d.* program of Fig. 5. It contains one outer block with counter i and two inner blocks with counters j and k globally initialized. To prove termination of the program, we have to find bounds for i at A, j at B, and k at C.

It is not difficult to discover the invariant

$$y_1 > 0 \wedge y_2 > 0 \quad \text{at} \quad A, B \text{ and } C. \tag{1}$$

To link j with y_1, we note that y_1 and y_2 are integers and that each time j is increased by one, y_1 is decreased by at least one (because $y_2 \geqq 1$). Thus we obtain

$$y_1 \leqq x_1 - j \quad \text{at} \quad A, B \text{ and } C, \tag{2}$$

and similarly

$$y_2 \leqq x_2 - k \quad \text{at} \quad A, B \text{ and } C. \tag{3}$$

We may use (1) with (2) and (1) with (3) to conclude that

$$j < x_1 \wedge k < x_2 \tag{4}$$

throughout the computation, bounding the total number of executions of the inner loops.

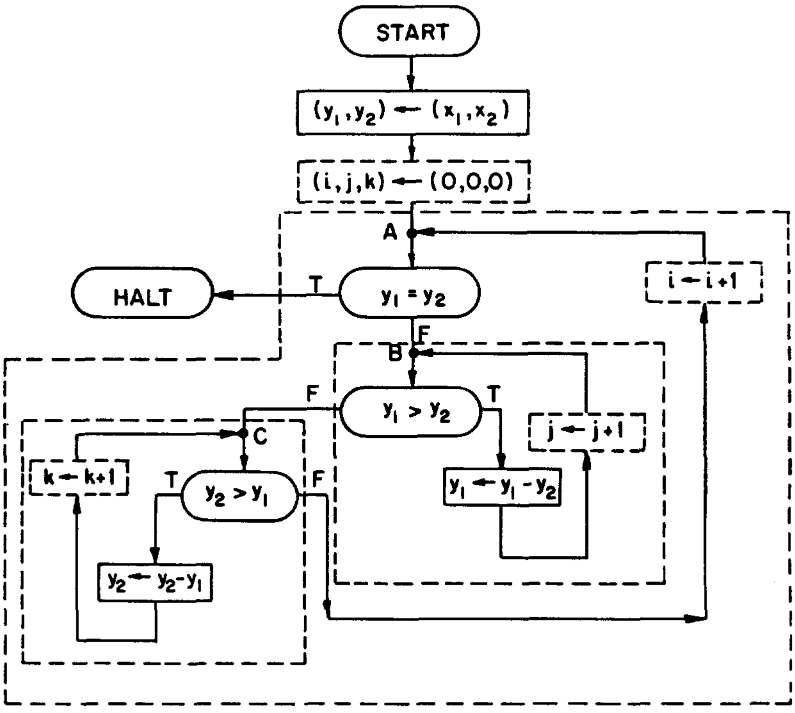


Fig. 5. *g.c.d.* program (with counters)

Since we have upper bounds on j and k, but need an upper bound on the counter i of the outer loop, we would like to show that

$$i \leqq j+k \quad \text{at} \quad A, \tag{5}$$

i.e., that each time we complete the outer loop, at least one of the inner loops has been executed on that pass. In order to establish that this is indeed an invariant, we must also show that

$$(y_1 \neq y_2 \wedge i \leqq j+k) \vee i < j+k \quad \text{at} \quad B.$$

That is, either control has arrived at B from A, so $y_1 \neq y_2 \wedge i \leqq j+k$ holds, or control was already at B and made a pass around the loop, so $i < j+k$. Similarly we must show that

$$(y_1 < y_2 \wedge i \leqq j+k) \vee i < j+k \quad \text{at} \quad C.$$

Using these assertions, it is easy to verify that in fact $i \leqq j+k$ is an invariant at A. Then clearly the outer loop must also terminate because from (4) and (5) it follows that

$$i < x_1 + x_2 \quad \text{at} \quad A. \tag{6}$$

Note that the use of the counters served to reduce the sensitivity to the placement of the cutpoints seen in Floyd's method. This is because an invariant which is true at the cutpoint of a loop, and which involves only counters and

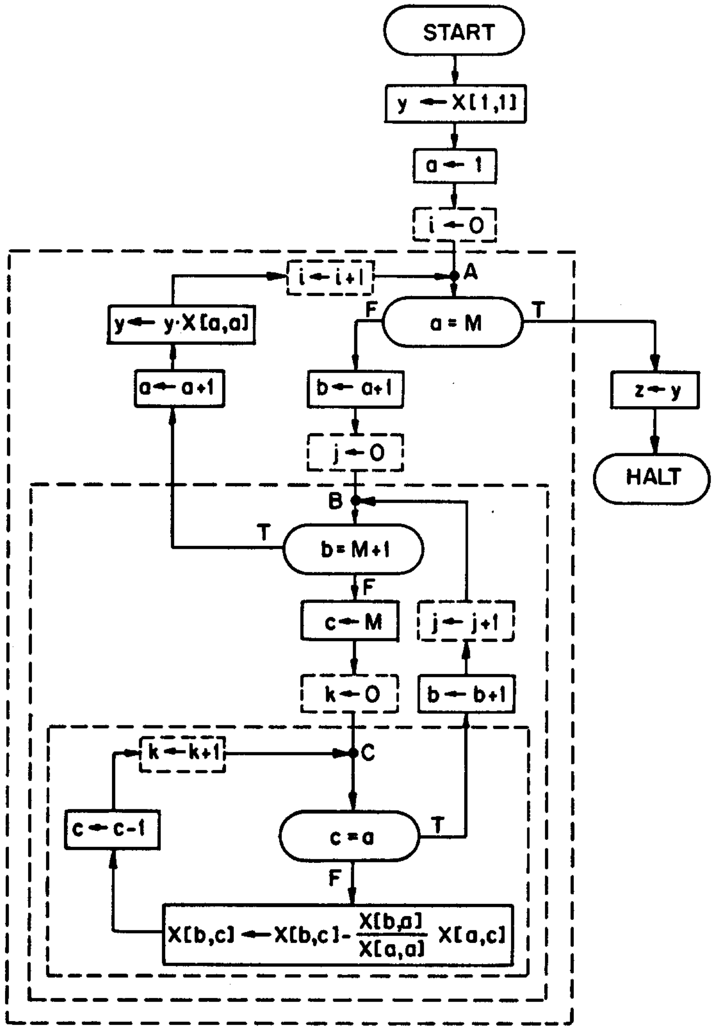


Fig. 6. Program for evaluating the determinant $z = |X|$ (with counters)

constants, is actually true anywhere on the loop, except for possible minor perturbation by a constant. □

Example 3 (loop approach). Let us consider the Gaussian elimination program of Example 3. We demonstrate the division to blocks and the (local) placement of the counters in Fig. 6. The invariants needed to bound the counters are

$$q_A\colon\ 1 \leqq a \leqq M \qquad \wedge\ i = a-1,$$
$$q_B\colon\ 1 \leqq a < M \wedge b \leqq M+1 \qquad \wedge\ i = a-1 \wedge j = b-(a+1),$$
$$q_C\colon\ 1 \leqq a < M \wedge b < M+1 \wedge a \leqq c \wedge i = a-1 \wedge j = b-(a+1) \wedge k = M-c.$$

Then clearly

$$q_A \supset i \leqq M-1,$$
$$q_B \supset j \leqq M-1,$$
$$q_C \supset k \leqq M-1,$$

proving termination by the loop approach.

Note that since the program variables are very similar to counters, the invariants connecting the variables to the counters are trivial. The nesting of the blocks and the local initialization of the counters take care of the relations between the loops which were handled in Floyd's method by using "triples". □

An important side-benefit of the loop approach lies in the added information provided on the (time) complexity and the control behavior of the given program. In proving termination by showing counters bounded, we actually have obtained upper bounds on the number of times the loops may be executed. Note that in Example 1 we also obtained the interesting information that the right path around the loop will ultimately be executed the same number of times as the left path (i.e., $i=j$ when the program terminates). Moreover, since $n=0 \vee n/2+x \leqq 111$, the loop itself will be executed no more than $\max(0, 222-2x)$ times. Similarly, in Example 2 we obtained the (rather loose) bound of x_1+x_2 on the number of executions of the outer loop, and the bounds of x_1 and x_2, respectively, for the inner loops.

It would be natural to extend this by refining the estimates and by also considering lower bounds on the counters. Although, of course, at a cutpoint *inside* the loop we may only assert that its counter i is non-negative, we can often establish a constant r', such that $r' \leqq i$ is an invariant immediately after exit from the loop.

3. The Exit Approach

Note that both in Floyd's method and in the loop approach there is not necessarily any direct reference to those tests of the program which lead out of the block. Another type of proof, which we term the *exit approach*, involves generating for each cutpoint the conditions which would lead out of the block from the cutpoint. The program will terminate for a given input if for every cutpoint either (a) such a condition will eventually hold, or (b) the cutpoint is never reached.

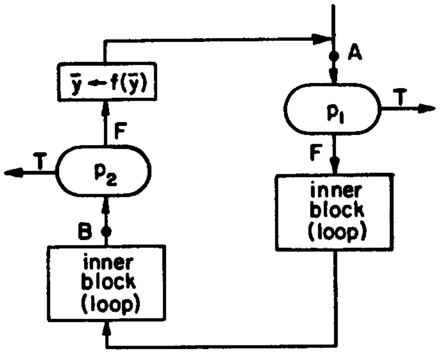
For the cutpoint A of a block with a locally initialized counter n and k exits we define the *exit condition* $R_A(\bar{x}, \bar{y}(n))$ as

$$p_1(\bar{x}, \bar{y}(n)) \vee p_2(\bar{x}, \bar{y}(n)) \vee \cdots \vee p_k(\bar{x}, \bar{y}(n))$$

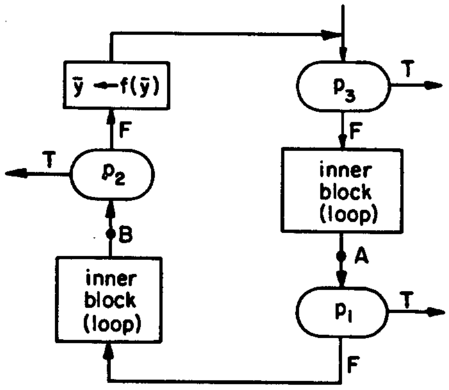
where $p_i(\bar{x}, \bar{y}(n))$ is the condition for traversing the path from the cutpoint A to the $i-th$ exit of the block. We then try to find loop invariants q_A at A such that

$$\forall \bar{x}\,[q_A \supset \exists n_0 R_A(\bar{x}, \bar{y}(n_0))].$$

This indicates that there must be a value $n_0 \geqq 0$, such that after n_0 iterations of the top-level loop of the block, one of the $p_i(\bar{x}, \bar{y}(n))$ will be true and therefore the corresponding exit path of the block will be taken.



a



b

Fig. 7a and b

For the common case of "structured blocks", i.e. blocks with a single exit, the cutpoint may be located next to the exit test and the exit condition may be generated by "forward substitution" along the path between the cutpoint and the exit. In the more general case, there may be several exits from the block, and it may be impossible to generate the entire exit condition at a given cutpoint, because of inner blocks between the cutpoint and the exit, as is the case for cutpoint A of Figs. 7a and 7b. This difficulty may often be overcome simply by choosing the location of the cutpoint with more care. For example, for the loop of Fig. 7a, the exit condition $R_B(\bar{x}, \bar{y}(n))$: $p_2(\bar{x}, \bar{y}(n)) \vee p_1(\bar{x}, f(\bar{y}(n)))$ is easy to generate at point B. When there is no way to generate the entire exit condition at a single cutpoint, as in Fig. 7b, the problem can usually be treated by using a set of cutpoints, such as A *and* B in Fig. 7b. In this case we generate for every one of these cutpoints partial exit conditions each of which would "cover" only some of the exits from the block. For the loop of Fig. 7b, the partial exit conditions are R_A: $p_1(\bar{x}, \bar{y}(n))$ and R_B: $p_2(\bar{x}, \bar{y}(n)) \vee p_3(\bar{x}, f(\bar{y}(n)))$. Then to show termination it is sufficient to prove that

$$\forall \bar{x}\{[q_A \supset \exists n_0 R_A(\bar{x}, \bar{y}(n_0))] \vee [q_B \supset \exists n_0 R_B(\bar{x}, \bar{y}(n_0))]\}.$$

Example 3 (exit approach). For the Gaussian elimination program of Fig. 6, a proof by the exit approach would use the same invariants as for the loop approach.

For the innermost block, with the exit test $c=a$, we must show that

$$q_C \supset \exists k_0\ [c(k_0)=a].$$

We use the facts that a and M are constant in the block, that the a and M are integers, and that $1 \leqq a < M$ and $c=M-k$ are invariants at C. Since $1 \leqq a < M$ implies that $\exists k_0 [M-k_0=a]$, and $c=M-k$ implies $\forall k_0 [c(k_0)=M-k_0]$ clearly

$$1 \leqq a < \mathrm{M} \wedge c=M-k \supset \exists k_0 [c(k_0)=a],$$

and therefore the innermost loop must terminate each time it is entered.

Similarly, it is not difficult to show that

$q_B \supset \exists j_0 [b(j_0)=M+1]$,

$q_A \supset \exists i_0 [a(i_0)=M]$. □

Even if we ignore the problem of generating the exit conditions, we do not consider the exit approach to be the preferable method for proving termination. The basic difficulty is that it is often unfeasible to show directly that certain values will occur during execution of the program. In the *g.c.d.* program (Fig. 2), for example, it is both difficult and unnecessary (even for correctness) to demonstrate directly that $y_1(n)=y_2(n)$ will eventually occur at point A.

The real importance of the exit approach lies in proving *non-termination*. Both Floyd's method and the loop approach are not suitable for this task. If we fail to find an appropriate set of descending functions $\{u_A\}$, or to find invariants which bound the counters, we still have not proven that it is impossible to find other, more successful, functions or invariants. However, if we are able to show that there exits some legal input value x_0 and some invariants q_A at a cutpoint A, such that

$$q_A \supset \forall n \sim R_A(\bar{x}_0, \bar{y}(n))$$

then the exit condition can never be true for execution with input x_0 and the block is proven non-terminating. A proof of non-termination could be valuable as an aid in debugging the program (see [13]).

Modified Example 2 (non-termination). The program of Fig. 8 differs from that of Fig. 5 only in that the exit test of the first inner loop is $y_1 \geqq y_2$ instead of $y_1 > y_2$. As in Example 2, it is not difficult to discover that $y_1 \geqq 0$ and $y_2 > 0$ are invariants at A, B, and C. Using $y_2 > 0$, we can prove termination of the first inner loop (by the loop approach). However, we cannot prove termination of the second inner loop only by using $y_1 \geqq 0$. The problem is clearly the possibility that $y_1=0$ at C.

Thus we try to see if there are input values such that the first inner loop could end with $y_1=0$, so that C will be reached with that value. The first time the first inner block is entered, $y_1(j)=y_1(0)-j \cdot y_2(0)$. We want to choose $y_1(0)$ and $y_2(0)$ so that $y_1(j)$ will be zero when the exit condition $y_1 < y_2$ becomes true. If we take $y_1(0)=m \cdot y_2(0)$ for some $m \geqq 1$, then cutpoint B is reached, and $y_1(j)=m \cdot y_2(0)-j \cdot y_2(0)=(m-j) \cdot y_2(0)$. In this case, since $y_1 \geqq 0 \wedge y_2 > 0$ is

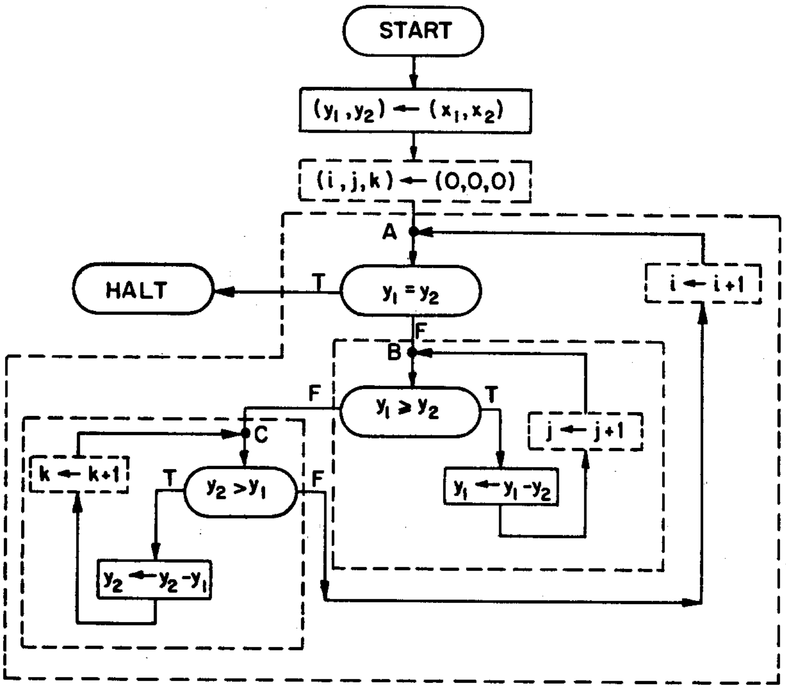

START
$(y_1, y_2) \leftarrow (x_1, x_2)$
$(i, j, k) \leftarrow (0,0,0)$
A
HALT
T
$y_1 = y_2$
F
$i \leftarrow i+1$
B
F
$y_1 \geq y_2$
T
C
$j \leftarrow j+1$
$k \leftarrow k+1$
T
$y_2 > y_1$
F
$y_1 \leftarrow y_1 - y_2$
$y_2 \leftarrow y_2 - y_1$

Fig. 8. A modified g.c.d. program

an invariant, $y_1 < y_2$ can occur only when $j = m$. Intuitively, this means that we will choose the initial value of y_1 as an exact multiple of y_2, and since y_2 is subtracted from y_1 each time the first inner loop is executed until $y_1 < y_2$, the loop will end with $y_1 = 0$. Since $y_1(0) = x_1$ and $y_2(0) = x_2$, we choose inputs $x_1 > 0$ and $x_2 > 0$ such that $x_1 = m \cdot x_2$ for some $m > 1$.

Once we have shown that C can be reached initially with $y_1 = 0$, it is easy to prove that $y_1 = 0$ and $y_2 > y_1$ will then be invariants at C, since $y_2 > 0$ is invariant and y_1 is not changed in the second inner loop.

Thus, we may conclude that in order to prove non-termination, we can choose integer inputs such that

$$x_1 > 0 \wedge x_2 > 0 \wedge x_1 = m \cdot x_2 \quad \text{for some} \quad m > 1$$

and then show that

$$q_A\colon\ y_1 = x_1 \wedge y_2 = x_2 \wedge j = 0,$$
$$q_B\colon\ y_1 = (m - j) \cdot y_2 \wedge y_1 \geqq 0 \wedge y_2 > 0,$$
$$q_C\colon\ y_1 = 0 \wedge y_1 < y_2 \wedge y_2 > 0$$

are invariants for such inputs. Clearly, cutpoint C is reached, and

$$\forall k\,[q_C \supset y_2(k) > y_1],$$

i.e., the negation of the exit condition of the second inner loop is an invariant for inputs as indicated, so the program is therefore proven non-terminating. □

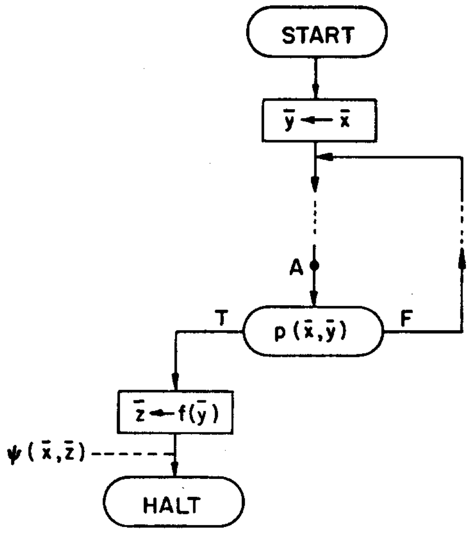


Fig. 9

4. Structural Induction

All of the methods in the previous sections prove termination independently of a proof of correctness. Burstall [3], however, has suggested an alternative which proves correctness and termination together. In this approach we show that if we assume some property p_A at a point A (in particular, the START point), we must eventually reach a point B (in particular, a HALT point) with some property q_B true. Instead of using invariants, such a claim is proven by induction on the domain of the input values (and is therefore called *structural induction*).

The notation of the exit approach is exactly suited to Burstall's method.

We denote by $\psi(\bar{x}, \bar{z})$ the desired relation between the input variables $\bar{x}$ and the output variables $\bar{z}$. Let us consider a simple program having the structure indicated in Fig. 9.

To show termination, we must prove that

$$\exists n_0 [p(\bar{x}, \bar{y}(n_0))] \quad \text{at} \quad A.$$

Similarly, to show termination and correctness w.r.t. $\psi(\bar{x}, \bar{z})$, we want to establish that

$$\exists n_0 [p(\bar{x}, \bar{y}(n_0)) \wedge \psi(\bar{x}, f(\bar{y}(n_0)))] \quad \text{at} \quad A.$$

Note that due to the way we defined $\bar{y}(n)$, if the above equation is true for n_0, then $\bar{y}(n_0)$ is defined and must actually occur at A, implying that the loop was not exited with $n < n_0$.

Example 1 (structural induction). Consider the termination and correctness of the "91-program" of Example 1 (see Fig. 4) with respect to the termination condition

$$p(x, \bar{y}(n)) \colon y_1(n) > 100 \wedge y_2(n) = 1$$

and the input-output relation

$$\psi(x, z): z = \textbf{if } x > 101 \textbf{ then } x - 10 \textbf{ else } 91.$$

For $x > 101$, the correctness, including termination, is trivial. Thus it suffices to prove that (using the initial values $y_1(0) = x$ and $y_2(0) = 1$) at A, for any integer x, $x \leq 101$:

$$\begin{aligned}&[y_1(0) = x \wedge y_2(0) = 1] \supset \\ &\quad \exists n [n \geqq 0 \wedge y_1(n) > 100 \wedge y_2(n) = 1 \wedge \\ &\qquad\qquad (y_1(n) - 10) = (\textbf{if } x > 101 \textbf{ then } x - 10 \textbf{ else } 91)].\end{aligned} \tag{1}$$

Since $x \leq 101$, this can be simplified to

$$\begin{aligned}&[y_1(0) = x \wedge y_2(0) = 1] \supset \\ &\quad \exists n [n \geqq 0 \wedge y_1(n) = 101 \wedge y_2(n) = 1].\end{aligned} \tag{2}$$

Instead of using invariants, we try to prove (2) by induction on x. However, as in many proofs by induction, it is easier to prove a more general statement, since this way the inductive hypothesis used is stronger. Generalizing 0 to a variable g and 1 to h, we try to prove at A that for any integer x, $x \leq 101$:

$$\begin{aligned}&\forall h \forall g \{[h \geqq 1 \wedge g \geqq 0 \wedge y_1(g) = x \wedge y_2(g) = h] \supset \\ &\quad \exists n [n \geqq g \wedge y_1(n) = 101 \wedge y_2(n) = h]\}.\end{aligned} \tag{3}$$

This means that if A is reached with $y_1(g) = x \wedge y_2(g) = h$ when $x \leqq 101$, $h \geqq 1$ and $g \geqq 0$, then A will eventually be reached with some $n \geqq g$ such that $y_1(n) = 101$ $\wedge\, y_2(n) = h$. Clearly (2) is a special case of (3), and $y_1(0) = x$ and $y_2(0) = 1$ actually occur at A, so proving (3) is sufficient to prove correctness and termination of the program.

We now proceed to prove (3) by using "going-down" induction on x.

Base step. $x = 101$. This is trivial: take $n = g$.

Inductive step. Assume (3) holds for every x', $x < x' \leqq 101$, and show it holds for x. We distinguish between two cases

(a) $90 \leqq x \leqq 100$

$y_1(g) = x \wedge y_2(g) = h$	(given)
$y_1(g+1) = x + 11 \wedge y_2(g+1) = h + 1$	(executing the left path, since $x \leqq 100$)
$y_1(g+2) = x + 1 \wedge y_2(g+2) = h$	(executing the right path, since $x + 11 > 100 \wedge h + 1 > 1$)
$y_1(n') = 101 \wedge y_2(n') = h$ for some $n' \geqq g + 2$	(induction, since $x < x + 1 \leqq 101$).

(b) $x<90$.

$y_1(g)=x \wedge y_2(g)=h$	(given)
$y_1(g+1)=x+11 \wedge y_2(g+1)=h+1$	(executing the left path since $x \leqq 100$)
$y_1(n')=101 \wedge y_2(n')=h+1$ for some $n' \geqq g+1$	(induction, since $x<x+11 \leqq 101$)
$y_1(n'+1)=91 \wedge y_2(n'+1)=h$	(executing the right path, since $101>100$ and $h+1>1$)
$y_1(n'')=101 \wedge y_2(n'')=h$ for some $n'' \geqq n'+1$	(induction, since $x<91$). $\square$

This method completely combines termination with correctness, and, in many cases, yields a very elegant proof. This seems especially true for iterative versions of "inherently" recursive programs. However, since it is not based on invariants, this type of proof could not take full advantage of the techniques used in existing verification systems.

Acknowledgement. We would like to thank Nachum Dershowitz, Amir Pnueli and Adi Shamir for their critical reading of the manuscript and their many helpful suggestions.

References

1. Aho, A.V., Ullman, J.D.: The theory of parsing, translation, and compiling Vol. 2. Englewood Cliffs (N.J.): Prentice Hall 1973
2. Allen, F.E.: A basis for program optimization. Proc. IFIP, Congress 71, Ljubljana, Yugoslavia. Amsterdam: North-Holland 1971, pp. 380–390
3. Burstall, R.M.: Program proving as hand simulation with a little induction. Proc. IFIP Congress 74, Stockholm, Sweden. Amsterdam: North-Holland 1974, pp. 308–312
4. Caplain, M.: Finding invariant assertions for proving programs. Proceedings of International Conference on Reliable Software. Los Angeles (Calif.) April 1975, pp. 165–171
5. Clint, M.: Program proving: coroutines. Acta Informatica **2**, 50–63 (1973)
6. Cooper, D.C.: Programs for mechanical program verification. Machine Intelligence 6. New York: American Elsevier 1971, pp. 43–59
7. Deutsch, L.P.: An interactive program verifier. Dept. of Comp. Sci., U. of Calif., Berkeley (Calif.) Ph.D. Thesis, June 1973
8. Elspas, B., Levitt, K.N., Waldinger, R. J.: An interactive system for the verification of computer programs. *SRI*, Menlo Park (Calif.), Sept. 1973
9. Elspas, B.: The semiautomatic generation of inductive assertions for proving program corretness. *SRI*, Menlo Park (Calif.), July 1974
10. Floyd, R.W.: Assigning meaning to programs. In: J.T. Schwartz (ed.): Proc. of a Symposium in Applied Mathematics, **19**. Providence (R.I.): Amer. Math. Soc. 1967, pp. 19–32
11. German, S.M., Wegbreit, B.: A synthesizer of inductive assertions. IEEE Trans. on Software Engineering, SE-1, 68–75 (1975)
12. Igarashi, S., London, R.L., Luckham, D.C.: Automatic program verification I: A logical basis and its implementation. Acta Informatica **4**, 145–182 (1975)
13. Katz, S.M., Manna, Z.: Towards automatic debugging of programs. Proceedings of International Conference on Reliable Software. Los Angeles (Calif.), April 1975

14. Katz, S.M., Manna, Z.: Logical analysis of programs. Comm. ACM, to appear (1976)
15. King, J.: A program verifier. Dept. of Comp. Sci., Carnegie-Mellon U., Pittsburgh (Pa.) Ph.D. Thesis, 1969
16. Knuth, D.E.: The Art of Computer Programing, Vol. I. Reading (Mass.): Addison-Wesley, 1968
17. Moriconi, M.S.: Towards the interactive synthesis of assertions. The University of Texas at Austin Research Report, October 1974
18. Sites, R.L.: Proving that computer programs terminate cleanly. Dept. of Computer Science, Stanford University, STAN-CS-74-418 Ph.D. Thesis, May 1974
19. Waldinger, R., Levitt, K.N.: Reasoning about programs. Artificial Intelligence **5**, 235–316 (1974)
20. Wegbreit, B.: The synthesis of loop predicates. Comm. ACM **17**, 102–112 (1974)

S. Katz
Z. Manna
Applied Mathematics Department
Weizmann Institute of Science
Rehovot, Israel