Logical Analysis of Programs

Shmuel Katz and Zohar Manna
The Weizmann Institute of Science

Most present systems for verification of computer programs are incomplete in that intermediate inductive assertions must be provided manually by the user, termination is not proven, and incorrect programs are not treated. As a unified solution to these problems, this paper suggests conducting a logical analysis of programs by using invariants which express what is actually occurring in the program.

The first part of the paper is devoted to techniques for the automatic generation of invariants. The second part provides criteria for using the invariants to check simultaneously for correctness (including termination) or incorrectness. A third part examines the implications of the approach for the automatic diagnosis and correction of logical errors.

Key Words and Phrases: logical analysis, invariants, program verification, correctness, incorrectness, termination, automatic debugging

CR Categories: 3.66, 4.42, 5.24

Parts of Sections 1 and 2 are based on a paper presented at the Third International Joint Conference on Artificial Intelligence, Stanford, Calif., August 1973 [14], and Section 4 is based on a paper presented at the International Conference on Reliable Software, Los Angeles, Calif., April 1975 [15].

Authors' addresses: S. Katz, IBM Research Center, Technion, Haifa, Israel; Z. Manna, Artificial Intelligence Project, Stanford University, Stanford, CA 94305.

Introduction

In recent years considerable effort has been devoted to the goal of proving (or "verifying") that a given computer program is partially correct—i.e. that if the program terminates, it satisfies some user-provided *input/output specification*. Floyd [8] suggested a method for proving partial correctness of flowchart programs which has been shown amenable to mechanization (see e.g. [17, 5, 27]). However, existing implementations are incomplete in that they are not oriented toward incorrect programs: their declared goal is to prove that a correct program really is correct. If a program is not verified, it is unclear whether the program is erroneous or whether a proper proof has simply not been discovered.

Floyd [8] also suggested a method for proving termination based on properties of well-founded sets. Although this traditional method is a most general and elegant way to prove termination, it is qualitatively different from the method for partial correctness, and thus the two are difficult to combine. Unfortunately the method is also not suitable for proving nontermination of a program which does not halt.

We suggest conducting logical analysis of programs using "invariant assertions" which express the actual relationships among the variables of the program. These "invariant assertions" differ from Floyd's programmer-supplied "inductive assertions" in that they are generated directly from the program text. In our conception, the invariants are independent of the output specification of the program and reflect what is actually happening during the computation, as opposed to what is supposed to be happening. Thus our invariants can be used either to verify the program with respect to its specifications or to prove that the program cannot be verified (i.e. contains an error). In addition, these invariants enable us to integrate proofs of termination and nontermination into our logical analysis. Invariants can also be used to debug an incorrect program, i.e. to diagnose the errors and to modify the program.

The existing implementations of Floyd's method for proving partial correctness are actually not fully automatic, since the user must provide the appropriate inductive assertions. This deficiency has been recognized and there has recently been a substantial effort to generate the inductive assertions automatically (for example, [3, 7, 10, 14, 23, 24, and 28]). Essentially, generating invariant assertions is a similar task. We therefore devote a large part of this paper to presenting our tech-

niques for the automatic (or semiautomatic) generation of invariants.

We actually intend that whenever new invariants have been produced, all invariants generated up to that point will be used to check simultaneously for (a) partial correctness, (b) termination, (c) incorrect results, and (d) nontermination. If correctness ((a) and (b)) has been established, an attempt may be made to optimize the program through a fundamental revision of the program statements, based on the invariants. If incorrectness ((c) or (d)) has been established, an attempt is made to automatically debug the program, i.e. to diagnose and correct the errors in a systematic manner, again using the invariants. If neither correctness nor incorrectness can be established, we attempt to generate additional invariants and repeat the process. Assertions ("comments") supplied by the programmer may or may not be correct, and therefore are considered to be just promising candidate invariants. As a last resort, it may nevertheless be possible to take a more radical approach and use the invariants for modifying the program so that correctness is guaranteed, taking the calculated risk of modifying an already correct program.

In the following sections we first present the techniques of automatic invariant generation, an algorithmic approach in Section 1 and a heuristic approach in Section 2. Then in Section 3 we describe the applications of the invariants for proving correctness (including termination) or incorrectness. In Section 4 we outline the practical implications of the invariants for automatic debugging. The other implications, such as for optimization, are discussed briefly in the Conclusion. The Conclusion also includes some bibliographical remarks.

Preliminaries

The programs treated in this paper are written in a simple flowchart language with standard arithmetic operators over the domain of real or integer numbers. We assume a flowchart program P with input variables x, which do not change during execution, and program variables y, which do change during execution and whose final values constitute the output of the program. In addition we are given an *input predicate* $\phi(x)$, which restricts the legal input values, and an *output predicate* $\psi(x, y)$, which indicates the desired relationship between the input and output values.

For convenience we consider *blocked programs*. That is, we assume the program is divisible into (possibly nested) "blocks" in such a way that every block has at most one top-level loop (in addition to possible lower-level loops which are already contained in inner blocks). The blocks we consider have one entrance and may have many exits. Algorithms for identifying such blocks can be found in [1]. Every "structured program," e.g. program without **goto** statements (see [4]), can be decomposed into such blocks.

The block structure allows us to treat the program by first considering inner blocks (ignoring momentarily that they are included in outer blocks) and then working outwards. Thus for each block we can consider its top-level loop using information we have obtained from the inner blocks.

The top-level loop of a block can contain several branches, but all paths around the loop must have at least one common point. For each loop we will choose one such point as the *cutpoint* of the loop.

We use *counters* attached to each block containing a loop as an essential tool in our techniques. Since each loop has a unique cutpoint, we associate a counter with the cutpoint of the loop. The counter is initialized before entering the block so that its value is zero upon first reaching the cutpoint, and is incremented by 1 exactly once somewhere along the loop before returning to the cutpoint. There are many locations where the initialization of the counters could be done. The two extreme cases 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). In the continuation, we will assume a *local* initialization of counters, since our experience has been that this is generally the most convenient choice.

The counters will play a crucial role both for generating invariants and for proving termination. They will be used both to denote relations among the number of times various paths have been executed and to help express the values assumed by the program variables. It should be noted that it is unnecessary to add the counters physically to the body of the program. Their location can merely be indicated, since their behavior is already fixed.

It is sometimes convenient to add auxiliary cutpoints at the entrance and exit of a block. In addition, we always add a special cutpoint on each arc immediately preceding a HALT statement. Such cutpoints will be called *haltpoints* of the program.

Our first task is to attach an appropriate invariant assertion $q_i(\bar{x}, \bar{y})$ to each cutpoint i. We first define our terms.

A predicate $q_i(x, y)$ is said to be an *invariant assertion* (or *invariant* for short) *at cutpoint i w.r.t.* $\phi(x)$ if for every input $\bar{a}$ such that $\phi(\bar{a})$ is true, whenever we reach point i with $y = \bar{b}$, then $q_i(\bar{a}, \bar{b})$ is true. An invariant at i is thus some assertion about the variables which is true for the current values of the variables each time i is reached during execution.

For a path α from cutpoint i to cutpoint j, we define $R_\alpha(x, y)$ as the condition for the path α to be traversed,

and $r_\alpha(\bar{x}, \bar{y})$ as the transformation in the $\bar{y}$ values which occurs on path α. A set S of cutpoints of a program P is said to be *complete* if, for each cutpoint i in S, all the cutpoints on any path from START to i are also in S.

We now state a sufficient condition (proven in [20]) for showing that assertions ("candidate invariants") are actually invariants.

LEMMA A. *Let S be a complete set of cutpoints of a program P. Assertions $\{q_i(\bar{x}, \bar{y}) \mid i \in S\}$ will be a set of invariants for P w.r.t. ϕ if*

(a) *for every path α from the START statement to a cutpoint j (which does not contain any other cutpoint)*[1]:

$$\forall \bar{x}[\phi(\bar{x}) \wedge R_\alpha(\bar{x}) \supset q_j(\bar{x}, r_\alpha(\bar{x}))], \text{ and}$$

(b) *for every path α from a cutpoint i to a cutpoint j (which does not contain any other cutpoint):*

$$\forall \bar{x}\forall \bar{y}[q_i(\bar{x}, \bar{y}) \wedge R_\alpha(\bar{x}, \bar{y}) \supset q_j(\bar{x}, r_\alpha(\bar{x}, \bar{y}))].$$

For the initial segment of a program shown in Figure 1, assertions $q_1(\bar{x}, \bar{y})$ and $q_2(\bar{x}, \bar{y})$ will be invariants at cutpoints 1 and 2 respectively if

(a) $\forall \bar{x}[\phi(\bar{x}) \supset q_1(\bar{x}, g(\bar{x}))]$,

(b) $\forall \bar{x}\forall \bar{y}[q_1(\bar{x}, \bar{y}) \wedge \sim t(\bar{x}, \bar{y}) \supset q_1(\bar{x}, f(\bar{x}, \bar{y}))]$,
$\forall \bar{x}\forall \bar{y}[q_1(\bar{x}, \bar{y}) \wedge t(\bar{x}, \bar{y}) \supset q_2(\bar{x}, h(\bar{x}, \bar{y}))]$.

Note that the input predicate $\phi(\bar{x})$, which depends only on $\bar{x}$ (variables not changed during execution), is automatically an invariant of any cutpoint of the program, and does not need any further justification.

Lemma A is slightly misleading, because it implies that a full-fledged set of assertions is provided at a complete set of the cutpoints and that these are checked simultaneously. In fact, the invariants will be added one after another until the needs of the logical analysis have been met.

At every stage of the invariant generating process, a situation as in Figure 2 will apply for each block. At cutpoints L, N, and M, invariants $p(\bar{x}, \bar{y})$, $q(\bar{x}, \bar{y})$, and $s(\bar{x}, \bar{y})$ respectively will already have been proven. However, we also will have promising candidates for invariants $p'(\bar{x}, \bar{y})$, $q'(\bar{x}, \bar{y})$, and $s'(\bar{x}, \bar{y})$, which we have so far been unable to prove to be invariants. These candidates could originate as comments given by the user or, as in the case of $s'(\bar{x}, \bar{y})$, from the output specification, which we automatically designate a candidate at the haltpoints. As indicated in Section 2, additional candidates may be generated during this process.

For a block of the form given in Figure 2, we concentrate on developing invariants at cutpoint N on the loop. For the auxiliary cutpoint M, the invariants are generated by "pushing forward" any invariant obtained at N. Thus, if at any stage an invariant $q(\bar{x}, \bar{y})$ has been

[1] Note that the $\bar{y}$ values are not defined at the START statement, and that they are initialized by constants or functions of $\bar{x}$ along a path from START. Thus, R_α and r_α for such a path are really only functions of $\bar{x}$, and not of $\bar{y}$.

Fig. 1. An initial segment of a program.

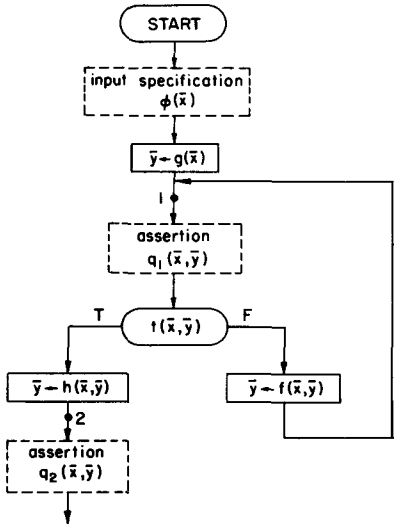
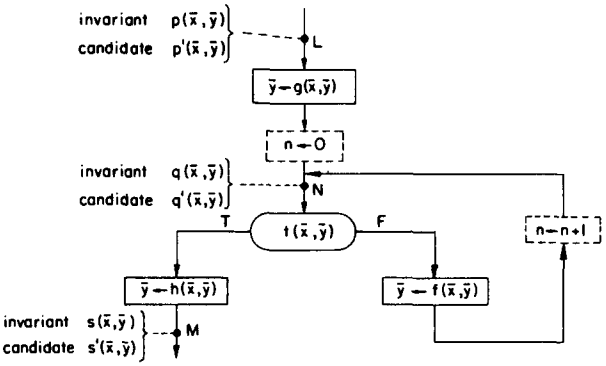


Fig. 2. A block containing a single-path loop.



established at N, we automatically can take as an invariant at M any $s(\bar{x}, \bar{y})$ satisfying

$$\forall \bar{x}\forall \bar{y}[q(\bar{x}, \bar{y}) \wedge t(\bar{x}, \bar{y}) \supset s(\bar{x}, h(\bar{x}, \bar{y})].$$

In order to establish that a candidate $q'(\bar{x}, \bar{y})$ is actually an invariant at N, it follows from Lemma A that we must show

(i) $\forall \bar{x}\forall \bar{y}[p(\bar{x}, \bar{y}) \supset q'(\bar{x}, g(\bar{x}, \bar{y}))]$,

and

(ii) $\forall \bar{x}\forall \bar{y}[q(\bar{x}, \bar{y}) \wedge q'(\bar{x}, \bar{y}) \wedge \sim t(\bar{x}, \bar{y}) \supset q'(\bar{x}, f(\bar{x}, \bar{y}))]$.

It must be emphasized that special care should be taken in case of failure in an attempt to establish that a candidate is an invariant. For example, suppose that q_1' and q_2' are candidates for invariants at the cutpoint N and that both q_1' and q_2' satisfy condition (i). It is entirely possible that neither q_1' nor q_2' satisfies condition (ii) individually, but that $q_1' \wedge q_2'$ does satisfy condition (ii), and therefore is an invariant. This phenomenon, i.e. that it is impossible to show a weak property but it is

possible to show a stronger one, is typical in mathematical proofs by induction. The explanation is that although we must show a stronger property on the right of the implication, we are also provided with a stronger inductive hypothesis on the left of the implication.

In Sections 1 and 2 we present techniques for discovering invariants. These techniques were originally designed with an automatic implementation in mind. However, they are in fact also useful for finding invariants by humans. For simplicity of presentation, we consider the single block of Figure 2. We will distinguish between two general approaches to producing invariants:

(1) the *algorithmic approach* in which we obtain guaranteed invariants $q(\bar{x}, \bar{y})$ at N directly from the assignments and tests of the loop (using also the entry invariant $p(\bar{x}, \bar{y})$ at L), and

(2) the *heuristic approach* in which we obtain a new candidate $q'(\bar{x}, \bar{y})$ for an invariant at N from already established invariants and old candidates which we have not yet been able to prove to be invariants.

1. Generation of Invariants: Algorithmic Approach

We present first the algorithmic approach for generating invariants. We distinguish between invariants derivable from the assignment statements and those based primarily on the test statements. The input predicate $\phi(\bar{x})$ and the fact that a counter is always a non-negative integer will be used as "built-in" invariants whenever convenient.

1.1. Generating invariants from assignment statements. We observe that assignment statements which are on the same path through the loop must have been executed an identical number of times whenever the cutpoint is reached. Thus the counter n of the cutpoint can be used to relate the variables iterated. We denote by $y(n)$ the value of y the $(n+1)$-th time the cutpoint is reached since the most recent entrance to the block (assuming a local initialization of the counters). Thus $y(0)$ indicates the value of y the first time the cutpoint is reached.

We use a self-evident fact as the basis for generating invariants: for $\bar{x}$ such that $\phi(\bar{x})$ is true and for each path α around the loop, we have

$$(1)\quad R_\alpha(\bar{x}, \bar{y}(n-1)) \supset \bar{y}(n) = r_\alpha(\bar{x}, \bar{y}(n-1)) \quad \text{for } n \geqslant 1.$$

That is, if values $\bar{y}(n-1)$ occurred at the cutpoint, and a path α around the loop is then followed (that is, $R_\alpha(\bar{x}, \bar{y}(n-1))$ is true), then the next values of $\bar{y}$ at the cutpoint (i.e. $\bar{y}(n)$) will be the result of applying r_α to $\bar{y}(n-1)$.

In practice, if there is only a single path around the loop such as in the block of Figure 2, it is usually easier to ignore the path-condition R_α, and find invariants which satisfy the stronger condition

$$(2)\quad \bar{y}(n) = r_\alpha(\bar{x}, \bar{y}(n-1)) \text{ for } n \geqslant 1.$$

Considering (2) for each component of $\bar{y}$, we have a set of recurrence equations, one for each y_j. We now attempt to express as many as possible of these equations in *iterative form*, e.g. as

(a) $y_j(n) = y_j(n-1) + g_j(\bar{x}, \bar{y}(n-1))$ or

(b) $y_j(n) = y_j(n-1) \cdot g_j(\bar{x}, \bar{y}(n-1))$.

Such forms are desirable because they can often be solved to obtain

$$(a')\quad y_j(n) = y_j(0) + \sum_{i=1}^{n} g_j(\bar{x}, \bar{y}(i-1)) \text{ or}$$

$$(b')\quad y_j(n) = y_j(0) \cdot \prod_{i=1}^{n} g_j(\bar{x}, \bar{y}(i-1)).$$

There are two ways to obtain invariants at a cutpoint from equations of the form (a') or (b'). First, it may be possible to express

$$\sum_{i=1}^{n} g_j(\bar{x}, \bar{y}(i-1)) \text{ or } \prod_{i=1}^{n} g_j(\bar{x}, \bar{y}(i-1))$$

as only a function of $\bar{x}$ and n, not containing any elements of $\bar{y}(i-1)$. We then have an assertion which relates $y_j(n)$, $y_j(0)$, $\bar{x}$, and n. Second, if there is a relation between

$$\sum_{i=1}^{n} g_l(\bar{x}, \bar{y})(i-1)) \text{ and } \sum_{i=1}^{n} g_k(\bar{x}, \bar{y}(i-1)),$$

or between

$$\prod_{i=1}^{n} g_l(\bar{x}, \bar{y}(i-1)) \text{ and } \prod_{i=1}^{n} g_k(\bar{x}, \bar{y}(i-1)),$$

then we can use this relation to connect $y_l(n)$ and $y_k(n)$. Once we have relations which are true for all $n \geqslant 0$, with all variables in the form $y_i(n)$, we can simplify by replacing each $y_i(n)$ by y_i, obtaining an invariant which may still contain occurrences of $y_i(0)$.

Whenever possible, known information from the entry invariant $p(\bar{x}, \bar{y})$ may be used to obtain $\bar{y}(0)$. When the variables are initialized immediately before entering the loop, $p(\bar{x}, \bar{y})$ will indicate the exact values of $\bar{y}(0)$. However, even when this is not the case, $p(\bar{x}, \bar{y})$ may often contain valuable information about $\bar{y}(0)$.

It is important to note that any predicate obtained as above, say from (a') or (b'), is not simply a candidate for an invariant, *but is actually an invariant.* This is because substituting the correct initial value in place of $\bar{y}(0)$ ensures that the relation obtained is true the first time the cutpoint is reached, and the use of $r_\alpha(\bar{x}, \bar{y})$ in obtaining the recurrence equations ensures that the relation is true subsequent times the cutpoint is reached.

Recall that the transformation from the recurrence equation (1) to (2) was made under the assumption that there was a single path around the loop as in Figure 2. The above discussion can easily be extended to the case of a loop with several possible paths—by using **if-then-else** expressions. For example, considering

the loop of Figure 3, with two paths around the loop, eq. (1) becomes

$$\sim t_1(\bar{x},\bar{y}(n-1)) \wedge \quad t_2(\bar{x},\bar{y}(n-1)) \supset \bar{y}(n) = f_1(\bar{x},\bar{y}(n-1))$$

$$\sim t_1(\bar{x},\bar{y}(n-1)) \wedge \sim t_2(\bar{x},\bar{y}(n-1)) \supset \bar{y}(n) = f_2(\bar{x},\bar{y}(n-1)).$$

These can be combined into one statement, as

$$\sim t_1(\bar{x},\bar{y}(n-1)) \supset [\textbf{if } t_2(\bar{x},\bar{y}(n-1)) \textbf{ then } \bar{y}(n) = f_1(\bar{x},\bar{y})(n-1)) \textbf{ else } \bar{y}(n) = f_2(\bar{x},\bar{y}(n-1))].$$

Since $t_1(\bar{x},\bar{y})$ controls the exit from the block, and does not affect the choice between the two paths around the loop, it can be ignored, as before, giving the stronger condition

$$(3)\ \textbf{if } t_2(\bar{x},\bar{y}(n-1)) \textbf{ then } \bar{y}(n)=f_1(\bar{x},\bar{y}(n-1)) \textbf{ else } \bar{y}(n)=f_2(\bar{x},\bar{y}(n-1)).$$

Equations of this form can then be put in iterative form, and treated just like equations of form (2).

1.2. Generating invariants from tests. So far we have concentrated on generating invariants from assignment statements, and the tests have merely been an obstacle which had to be overcome. Now we will show how the tests can be an aid to allow extracting additional invariants from the loop.

Suppose the block has the paths $\alpha_1, \alpha_2, \ldots, \alpha_k$ ($k \geqslant 1$) from the cutpoint N around the loop back to N. Again we shall use an obvious fact: whenever N is reached during execution, either it is the first visit at the point for the present entrance to the block, or control was previously at N and one of the paths $\alpha_1, \ldots, \alpha_k$ was followed, i.e. the block was not exited. Letting n be the counter of the block, this can be written more precisely as

$$(4)\ n = 0 \vee [R_{\alpha_1}(\bar{x},\bar{y}(n-1)) \vee R_{\alpha_2}(\bar{x},\bar{y}(n-1)) \vee \ldots \vee R_{\alpha_k}(\bar{x},\bar{y}(n-1))].$$

The above claim (4) is clearly always true at N. By expressing $\bar{y}(n-1)$ in terms of $\bar{y}(n)$—using the recurrence equations given by (1)—and adding known information about $\bar{y}(0)$, we can often simplify (4). Again, if we obtain relations which are true for all $n \geqslant 0$, and all variables are expressed as $y_i(n)$, we can remove the parameter n to obtain an invariant. We can also use known invariants at N, in particular those generated from assignment statements, in order to help simplify.

We demonstrate some of the above techniques on a program. Note that at this point we make no claim about whether this program is correct.

Example A. The program[2] A of Figure 4 is intended to divide x_1 by x_2 within tolerance x_3, where x_1, x_2, and x_3 are real numbers satisfying $0 \leqslant x_1 < x_2$ and

[2] This program is based on Wensley's division algorithm [29]. Note that we use a vector assignment notation, where, for example, $(y_1, y_4) \leftarrow (y_1 + y_2, y_4 + y_3/2)$ means that $y_1 \leftarrow y_1 + y_2$ and $y_4 \leftarrow y_4 + y_3/2$ simultaneously.

Fig. 3. A block containing a loop with two paths.

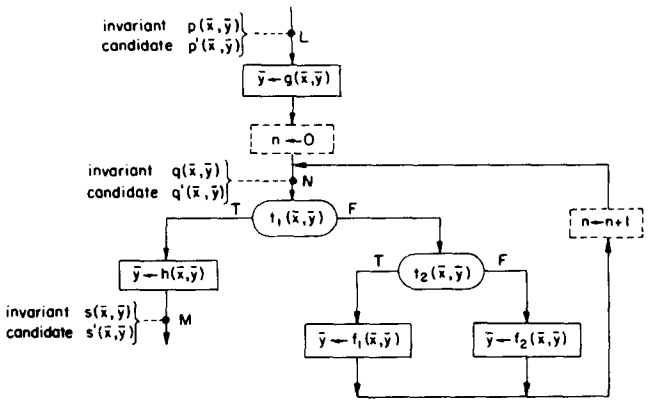
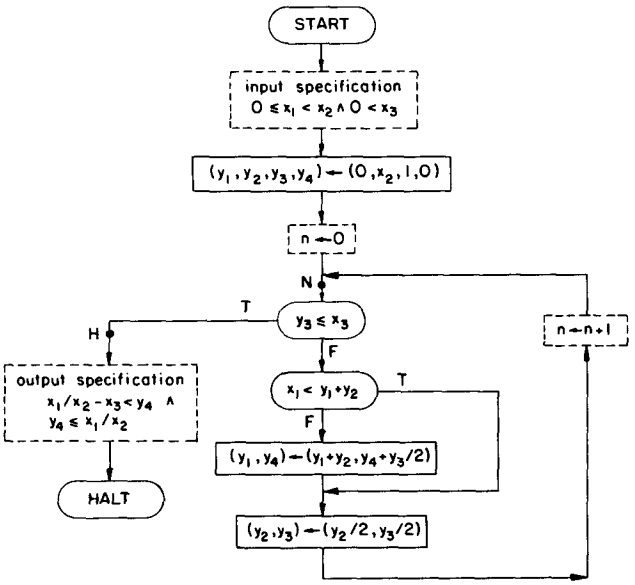


Fig. 4. Program A. Real division within tolerance.



$0 < x_3$. Thus the final value of y_4 is supposed to satisfy $x_1/x_2 - x_3 < y_4 \leqslant x_1/x_2$ at the haltpoint H. For clarity we have explicitly added the counter n to the program. There are two paths around the loop from the cutpoint N back to N: the right path following the T-branch from the test $x_1 < y_1 + y_2$, and the left path following the corresponding F-branch. By using (1) we have for each path:

right path:

$$[y_3(n-1) > x_3 \wedge x_1 < y_1(n-1) + y_2(n-1)] \supset [y_1(n) = y_1(n-1) \wedge y_2(n) = y_2(n-1)/2 \wedge y_3(n) = y_3(n-1)/2 \wedge y_4(n) = y_4(n-1)],$$

left path:

$$[y_3(n-1) > x_3 \wedge x_1 \geqslant y_1(n-1) + y_2(n-1)] \supset [y_1(n) = y_1(n-1) + y_2(n-1) \wedge y_2(n) = y_2(n-1)/2 \wedge y_3(n) = y_3(n-1)/2 \wedge y_4(n) = y_4(n-1) + y_3(n-1)/2]$$

Since the assignments to y_2 and y_3 are not affected by which path is used, we may ignore the path conditions, obtaining

$$y_2(n) = y_2(n-1)/2 \wedge y_3(n) = y_3(n-1)/2.$$

Both of these are in the iterative form (b) and may be solved to yield

$$y_2(n) = y_2(0) \cdot \prod_{i=1}^{n} 1/2 \wedge y_3(n) = y_3(0) \cdot \prod_{i=1}^{n} 1/2.$$

Since it is clear that $y_2(0) = x_2$ and $y_3(0) = 1$ at N and that $\prod_{i=1}^{n} 1/2 = 1/2^n$, we have (dropping the parameter n) the invariants

(A1) $y_2 = x_2/2^n$ at N and
(A2) $y_3 = 1/2^n$ at N.

These may be combined to yield the additional invariant

(A3) $y_2 = x_2 \cdot y_3$ at N.

For variables y_1 and y_4, we apply the techniques used to obtain eq. (3). Ignoring the exit test $y_3 \leq x_3$ and expressing the effect of the branching by using **if-then-else**, the resulting recurrence relations are

$$y_1(n) = \textbf{if } x_1 < y_1(n-1) + y_2(n-1) \textbf{ then } y_1(n-1) \textbf{ else } y_1(n-1) + y_2(n-1)$$

$$y_4(n) = \textbf{if } x_1 < y_1(n-1) + y_2(n-1) \textbf{ then } y_4(n-1) \textbf{ else } y_4(n-1) + y_3(n-1)/2.$$

Both of these are in iterative form and we can obtain the summations

$$y_1(n) = y_1(0) + \sum_{i=1}^{n} [\textbf{if } x_1 < y_1(i-1) + y_2(i-1) \textbf{ then } 0 \textbf{ else } y_2(i-1)]$$

$$y_4(n) = y_4(0) + \sum_{i=1}^{n} [\textbf{if } x_1 < y_1(i-1) + y_2(i-1) \textbf{ then } 0 \textbf{ else } y_3(i-1)/2].$$

We will use the invariant (A3), that $y_2 = x_2 \cdot y_3$ at N, in order to bring the two summations to an identical form. Substituting $x_2 \cdot y_3(i-1)$ for $y_2(i-1)$ in the **else** part of the equation for $y_1(n)$, factoring out x_2, and dividing by 2 inside the summation and multiplying by 2 outside, we obtain

$$y_1(n) = y_1(0) + 2x_2 \cdot \sum_{i=1}^{n} [\textbf{if } x_1 < y_1(i-1) + y_2(i-1) \textbf{ then } 0 \textbf{ else } y_3(i-1)/2].$$

We have expressed $y_1(n)$ and $y_4(n)$ in terms of the same summation, which thus can be used to connect these two variables. Substituting $y_1(0) = 0$ and $y_4(0) = 0$, we obtain

$$y_1(n)/(2x_2) = \sum_{i=1}^{n} [\textbf{if } x_1 < y_1(i-1) + y_2(i-1) \textbf{ then } 0 \textbf{ else } y_3(i-1)/2] = y_4(n).$$

Thus we have the invariant

(A4) $y_1 = 2x_2 \cdot y_4$ at N.

We now turn to eq. (4), using the tests of the loop to generate additional invariants. We have the fact

$$n = 0 \vee [y_3(n-1) > x_3 \wedge x_1 < y_1(n-1) + y_2(n-1)] \quad \ldots \text{right path} \vee [y_3(n-1) > x_3 \wedge x_1 \geq y_1(n-1) + y_2(n-1)] \quad \ldots \text{left path.}$$

For each path we now use the equations for $\bar{y}(n)$ obtained from (1) in order to express $\bar{y}(n-1)$ in terms of $\bar{y}(n)$. For the right path we will use the fact that $y_1(n) = y_1(n-1)$, $y_2(n) = y_2(n-1)/2$, and $y_3(n) = y_3(n-1)/2$, while for the left path we will use the fact that $y_1(n) = y_1(n-1) + y_2(n-1)$ and $y_3(n) = y_3(n-1)/2$. These substitutions will yield

$$n = 0 \vee [2y_3(n) > x_3 \wedge x_1 < y_1(n) + 2y_2(n)] \quad \ldots \text{right path} \vee [2y_3(n) > x_3 \wedge x_1 \geq y_1(n)] \quad \ldots \text{left path.}$$

Removing the parametrization in terms of n, and separating the term involving y_3, we have the two new invariants at N,

$$[n = 0 \vee 2y_3 > x_3] \wedge [n = 0 \vee x_1 < y_1 + 2y_2 \vee x_1 \geq y_1].$$

To obtain stronger invariants, we can check whether the $n = 0$ case is subsumed in the other alternatives. The left conjunct may not be so reduced and we have the invariant

(A5) $n = 0 \vee 2y_3 > x_3$ at N.

The $n = 0$ possibility in the right conjunct can easily be seen to be included in the other possibilities, since $y_1(0) = 0$ and $x_1 \geq 0$ imply that $x_1 \geq y_1(0)$. Thus we have the invariant

(A6) $x_1 < y_1 + 2y_2 \vee x_1 \geq y_1$ at N.

Note that invariant (A6) is a disjunction of the form $p \vee q$. This disjunction actually reflects the effect of taking the right path or the left path, respectively, around the loop. □

2. Generation of Invariants: Heuristic Approach

We now describe several heuristic techniques which suggest promising candidates for invariants. There is no guarantee that the candidates produced are actually invariants, and they must be checked (using Lemma A).

It is important to notice that when we are unable to

establish that a candidate is an invariant, it should be saved to retry later. The first reason for retrying the candidate is that in the meantime we may have established independently additional invariants such that the extended set of invariants along with the candidate satisfy Lemma A. A second reason is that additional "related" candidates may have been generated and that, due to the "induction phenomenon" mentioned after Lemma A, we now can prove the candidate in conjunction with the additional candidates even though we could not prove it alone.

It should be clear that before an automatic system for generating invariants is practical, strong guidance must be provided for the application of the following heuristics, since, applied blindly, they could result in too many irrelevant candidates. Here we merely state some of the various possibilities in order to give the flavor of this approach.

2.1. Strengthening existing invariants. Whenever we have established an invariant at a cutpoint i which is a disjunction of the form

$$p_1 \vee p_2 \vee \cdots \vee p_k \quad (k \geqslant 2),$$

we try to see whether any subdisjunction (in particular, each p_j alone) is itself an invariant at i. In the previous section, we actually used this approach when we eliminated the $n = 0$ alternative to obtain the invariant (A6).

2.2. Weakening existing candidates. Suppose we have at i a candidate which is a conjunction of the form

$$p_1 \wedge p_2 \wedge \ldots \wedge p_k \quad (k \geqslant 2),$$

and we have failed to prove that it is an invariant at i. One natural heuristic is to try a subconjunction (in particular, possibly each p_j alone) as a "new" candidate. Note that the failure to prove $p_1 \wedge p_2 \wedge \ldots \wedge p_k$ an invariant says nothing about whether its subconjunctions are invariants. Theoretically, any nonempty subconjunction is a legitimate candidate and should be checked independently.

For the next three heuristics, we refer back to Figure 2.

2.3. Pushing candidates backwards. Let us assume that $p(x,y)$ is an established invariant at L and $q'(\bar{x},\bar{y})$ is a candidate invariant at N. If the inductive step around the loop has been shown to establish $q'(x,y)$ at N, then the only difficulty could be that $p(x,y)$ did not imply $q'(x,g(x,y))$. We then try

$$p'(\bar{x}, \bar{y}) : p(\bar{x}, \bar{y}) \supset q'(\bar{x}, g(\bar{x}, \bar{y}))$$

as a new candidate at L. This will "fix" the problem with $q'(\bar{x}, \bar{y})$ but of course we must now prove $p'(\bar{x}, \bar{y})$ an invariant at L. Note that in any case $p'(\bar{x}, \bar{y})$ must be an invariant at L if we are to succeed in showing that $q'(\bar{x}, \bar{y})$ is an invariant at N and in this sense is the "weakest" possible precondition for the base case of the induction for $q'(\bar{x}, \bar{y})$.

A similar technique can also be used to generate candidates at N:

Let us assume that $q(\bar{x}, \bar{y})$ is an established invariant at N and $s'(\bar{x}, \bar{y})$ is a candidate invariant at M. Since $s'(\bar{x}, \bar{y})$ is reached only from N, the reason we were not able to prove it an invariant must be that $q(\bar{x}, \bar{y}) \wedge t(\bar{x}, \bar{y})$ did not imply $s'(\bar{x}, h(\bar{x}, \bar{y}))$. Thus we would like to find a candidate $q'(\bar{x}, \bar{y})$ at N such that

$$(5)\quad [q(\bar{x}, \bar{y}) \wedge q'(\bar{x}, \bar{y}) \wedge t(\bar{x}, \bar{y})] \supset s'(\bar{x}, h(\bar{x}, \bar{y})).$$

Among the many possible choices of $q'(\bar{x}, \bar{y})$ which satisfy this condition are

$$q'(\bar{x}, \bar{y}) : [q(\bar{x}, \bar{y}) \wedge t(\bar{x}, \bar{y})] \supset s'(\bar{x}, h(\bar{x}, \bar{y})) \text{ or}$$
$$q'(\bar{x}, \bar{y}) : s'(\bar{x}, h(\bar{x}, \bar{y})).$$

This first possibility is, just as above, the "weakest" possible assertion which satisfies (5), while the second is the "strongest" possible. As a very useful third alternative to the above suggestions, the transitivity of certain inequality or equality relations can suggest a candidate which takes into account the known information from $q(\bar{x}, \bar{y})$ and $t(\bar{x}, \bar{y})$. For example, if we need a q' such that $q' \wedge B < C \supset A < C$ where A, B, and C are any terms, the relation $A \leqslant B$ is a natural candidate for q'.

Any candidate for $q'(\bar{x}, \bar{y})$ obtained from formula (5) must be checked. Unfortunately, there are no clear-cut criteria for finding a $q'(\bar{x}, \bar{y})$ which will be easy to prove. If we fail to show some candidate $q'(\bar{x}, \bar{y})$ an invariant at N, clearly some "weaker" version may nevertheless succeed. On the other hand, because of the "induction phenomenon" it is quite possible that a "stronger" candidate $q'(\bar{x}, \bar{y})$ actually could be more easily proven.

Note that this process could also be used for the path around the loop, adding $q''(\bar{x}, \bar{y})$ as a new candidate at N so that we are able to prove $q'(\bar{x}, f(\bar{x}, \bar{y}))$. Again, this has the effect of transferring the "burden of proof" from $q'(\bar{x}, \bar{y})$ to $q''(\bar{x}, \bar{y})$.

2.4. Pushing invariants forward. Assuming that $p(x,y)$ is an established invariant at L, a straightforward heuristic is to try to find a candidate $q'(x,y)$ at N such that

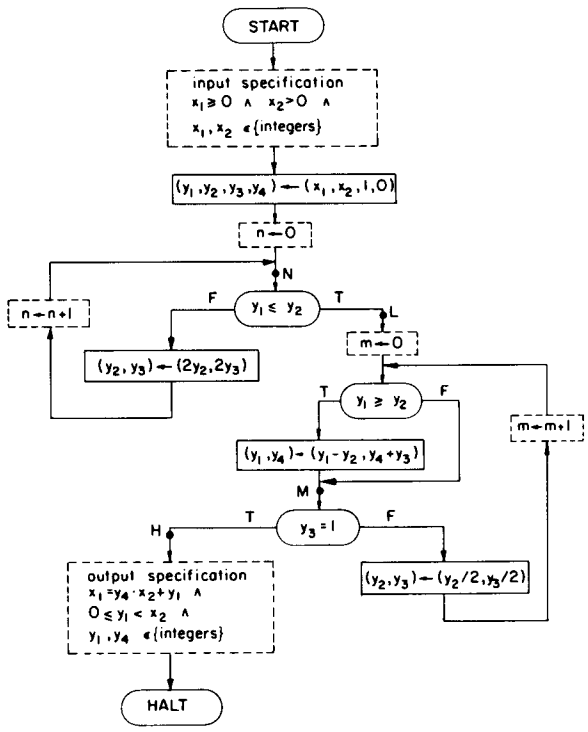
$$p(\bar{x}, \bar{y}) \supset q'(\bar{x}, g(\bar{x}, \bar{y})).$$

The above equation ensures that the first time N is reached, $q'(\bar{x}, \bar{y})$ is true. Of course, in order to complete the proof that $q'(\bar{x}, \bar{y})$ is an invariant, the corresponding formula for the path around the loop must be considered.[3]

Note that immediately after every assignment $y_i \leftarrow f(\bar{x}, \bar{y})$ where $f(\bar{x}, \bar{y})$ does not include y_i itself, we know that $y_i = f(\bar{x}, \bar{y})$ is an invariant. Also, after every test $t(\bar{x}, \bar{y})$ we can add the invariant $t(\bar{x}, \bar{y})$ on the T-branch, and $\sim t(\bar{x}, \bar{y})$ on the F-branch. Such invariants can also

[3] This is actually the method indicated in the preliminaries for obtaining invariants at cutpoint M at the exit of the loop.

Fig. 5. Program B. Hardware integer division.



be pushed forward to generate useful candidates at the cutpoints.

2.5. Bounding variables. One often useful type of candidate for $q'(\bar{x},\bar{y})$ at N involves finding upper or lower bounds for the variables, expressed only in terms of *constant expressions* with respect to the block. That is, the bounds contain only constants, input variables, or other program variables which are unchanged inside the loop of the block.

Suppose that by considering $f(\bar{x}, \bar{y})$ and the invariant $q(\bar{x}, \bar{y})$ at N, we are able to identify a variable y_j which is either always nondecreasing (or always nonincreasing) along the path around the loop. Now we try to infer from $p(\bar{x}, \bar{y})$ an initial value $y_j(0) = E$ for y_j at N where E is a constant expression with respect to the block. If y_j is nondecreasing along the loop we can conclude that $y_j \geq E$ is an invariant at N, while if y_j is nonincreasing $y_j \leq E$ is an invariant.

A similar heuristic tries to establish that the variables maintain some data type, e.g. *integer* or *real*, during execution.

We will first illustrate the application of the heuristics in obtaining some additional invariants for the program of Example A, and then present a new example which will illustrate the possible interplay between the algorithmic and heuristic techniques.

Example A (continued). Let us consider again the program A of Figure 4. Applying heuristic 2.1 to the invariant

(A6) $x_1 < y_1 + 2y_2 \vee x_1 \geq y_1$ at N,

we check first whether $x_1 < y_1 + 2y_2$ is itself an invariant. From Lemma A, we can show that

(a) $\forall \bar{x}[0 \leq x_1 < x_2 \wedge 0 < x_3 \supset x_1 < 0 + 2x_2]$ and

(b) $\forall \bar{x} \forall \bar{y}\ [x_1 < y_1 + 2y_2 \wedge y_3 > x_3 \wedge x_1 < y_1 + y_2 \supset x_1 < y_1 + y_2]$,

$\forall \bar{x} \forall \bar{y}\ [x_1 < y_1 + 2y_2 \wedge y_3 > x_3 \wedge x_1 \geq y_1 + y_2 \supset x_1 < y_1 + y_2 + y_2]$.

Since all of the conditions are true, we have the invariant

(A7) $x_1 < y_1 + 2y_2$ at N.

For $x_1 \geq y_1$, the second disjunct of (A6), we can show that

(a) $\forall \bar{x}[0 \leq x_1 < x_2 \wedge 0 < x_3 \supset x_1 \geq 0]$,

(b) $\forall \bar{x} \forall \bar{y}[x_1 \geq y_1 \wedge y_3 > x_3 \wedge x_1 < y_1 + y_2 \supset x_1 \geq y_1]$

$\forall \bar{x} \forall \bar{y}[x_1 \geq y_1 \wedge y_3 > x_3 \wedge x_1 \geq y_1 + y_2 \supset x_1 \geq y_1 + y_2]$.

Since these conditions are all true, we have shown that the second alternative is also an invariant, i.e.

(A8) $x_1 \geq y_1$ at N.

We can combine the invariant (A4), $y_1 = 2x_2 \cdot y_4$, with (A8) to obtain an upper bound on y_4 in terms of $\bar{x}$, i.e. the invariant

(A9) $y_4 \leq x_1/(2x_2)$ at N.

This invariant will be of special use later, in Sections 3 and 4, and in practice would be generated only when a need for such a bound arises.

Now, by pushing forward to H the invariants (A1) to (A9) at N, and adding the exit test $y_3 \leq x_3$, we obtain

(A10) $y_3 \leq x_3 \wedge y_2 = x_2/2^n \wedge y_3 = 1/2^n \wedge y_2 = x_2 \cdot y_3 \wedge y_1 = 2x_2 \cdot y_4 \wedge (n = 0 \vee 2y_3 > x_3) \wedge x_1 < y_1 + 2y_2 \wedge x_1 \geq y_1 \wedge y_4 \leq x_1/(2x_2)$ at H. □

Example B. The program B shown in Figure 5 is supposed to perform integer division in a manner similar to computer hardware. For every integer input $x_1 \geq 0$ and $x_2 > 0$, we would like to have as output $y_1 = rem(x_1, x_2)$ and $y_4 = div(x_1, x_2)$, i.e. $x_1 = y_4 \cdot x_2 + y_1 \wedge 0 \leq y_1 < x_2 \wedge y_1, y_4 \in$ {integers}. This program differs from the previous example in that it contains two loops, one after the other. The upper block, with counter n and cutpoint N, consists of a simple loop, while the lower block, with counter m and cutpoint M, consists of a loop with two paths. For convenience, we have added an additional cutpoint L between the blocks.

Our strategy will be to gather initially as many invariants as possible at N. The algorithmic techniques will be used to directly generate invariants at N, and then some of the heuristics presented above will be used to suggest additional invariants. We then push the invariants forward to cutpoint L, so that we have as many invariants as we can when the second block is first reached. Then we will employ the algorithmic tech-

niques to generate invariants at M. Finally, we use heuristic techniques based on the invariants at L and M and the candidates implied by the output specification at H to generate additional invariants at M. We will not go into the problem of which heuristic rule to use first, but simply indicate how some candidates, which will indeed be useful invariants, can be found by using various heuristics.

Applying the algorithmic techniques for finding invariants at N, we obtain the equations

$$y_2(n) = y_2(0)\cdot\prod_{i=1}^{n} 2 = y_2(0)\cdot 2^n = x_2\cdot 2^n \text{ at } N,$$

$$y_3(n) = y_3(0)\cdot\prod_{i=1}^{n} 2 = y_3(0)\cdot 2^n = 2^n \text{ at } N.$$

Thus we can obtain the invariants

(B1) $y_2 = x_2\cdot 2^n \wedge y_3 = 2^n$ at N.

These can be combined to give

(B2) $y_2 = x_2\cdot y_3$ at N.

By pushing forward the information in $\phi(\bar{x})$ and the initial assignments (using heuristic 2.4), we get the additional invariants

(B3) $y_1 = x_1 \wedge y_4 = 0 \wedge y_1, y_2, y_3, y_4 \in \{\text{integers}\}$ at N.

Using heuristic 2.5, we note that y_2 and y_3 are always increasing around the loop, and since $y_2(0) = x_2$ and $y_3(0) = 1$ at N, we obtain the invariants

(B4) $y_2 \geqslant x_2 \wedge y_3 \geqslant 1$ at N.

Note that (B4) could also be obtained directly from (B1) using the implicit invariant $n \geqslant 0$.

Using the T-branch of the test $y_1 \leqslant y_2$ and pushing forward to L the invariants at N, we have the invariants

(B5) $y_2 = x_2\cdot 2^n \wedge y_3 = 2^n \wedge y_2 = x_2\cdot y_3 \wedge y_1 = x_1$
$\wedge\ y_4 = 0 \wedge y_1, y_2, y_3, y_4 \in \{\text{integers}\} \wedge y_2 \geqslant x_2$
$\wedge\ y_3 \geqslant 1 \wedge y_1 \leqslant y_2$ at L.

Generating invariants directly from the statements of the lower block, we first have the relations

$$y_2(m) = y_2(0)/2^m \text{ at } M, \quad y_3(m) = y_3(0)/2^m \text{ at } M.$$

Using the invariants $y_2 = x_2\cdot 2^n$ and $y_3 = 2^n$ from (B5) to establish $y_2(0)$ and $y_3(0)$ at M, we obtain the invariants

(B6) $y_2 = x_2\cdot 2^n/2^m \wedge y_3 = 2^n/2^m \wedge y_2 = x_2\cdot y_3$ at M.

Using the same technique for y_1 and y_4, we obtain the recurrence relations

$$y_1(m) = y_1(m-1) + [\textbf{if } y_1(m-1) \geqslant y_2(m-1) \textbf{ then } -y_2(m-1) \textbf{ else } 0] \quad \text{at } M,$$

$$y_4(m) = y_4(m-1) + [\textbf{if } y_1(m-1) \geqslant y_2(m-1) \textbf{ then } y_3(m-1) \textbf{ else } 0] \quad \text{at } M.$$

Writing these equations as a summation, then using (B6) to replace the occurrence of $-y_2(m-1)$ by $-x_2\cdot y_3(m-1)$ and factoring out $-x_2$, we obtain

$$y_1(m) = y_1(0) - x_2\cdot\sum_{i=1}^{m} [\textbf{if } y_1(i-1) \geqslant y_2(i-1) \textbf{ then } y_3(i-1) \textbf{ else } 0] \quad \text{at } M,$$

$$y_4(m) = y_4(0) + \sum_{i=1}^{m} [\textbf{if } y_1(i-1) \geqslant y_2(i-1) \textbf{ then } y_3(i-1) \textbf{ else } 0] \quad \text{at } M.$$

Simplifying, we get

(6) $y_1(m) - y_1(0) = -x_2\cdot(y_4(m) - y_4(0))$ at M.

We will again use invariants from (B5) at L, namely $y_1 = x_1$ and $y_4 = 0$, to establish $y_1(0)$ and $y_4(0)$ at M. There are two possible paths from L to M. If the right branch is used, clearly $y_1(0) = x_1 \wedge y_4(0) = 0$ at M. On the other hand, if the left branch is taken, the additional invariants from (B5), $y_1 \leqslant y_2$ and $y_2 = x_2\cdot y_3$ at L, together with the fact that $y_1 \geqslant y_2$ along this path, yield that $y_1 = y_2$ and $y_3 = y_2/x_2 = y_1/x_2 = x_1/x_2$, and therefore after the assignments $y_1 \leftarrow y_1 - y_2$ and $y_4 \leftarrow y_4 + y_3$ we have that $y_1(0) = 0$ and $y_4(0) = x_1/x_2$ at M. Substituting both possibilities for $y_1(0)$ and $y_4(0)$ into the above equation (6), we obtain in both cases $y_1(m) - x_1 = -x_2\cdot y_4(m)$. Thus we have the invariant

(B7) $x_1 = y_4\cdot x_2 + y_1$ at M.

Turning to the tests, following eq. (4) we have

$m = 0$
$\vee\ [y_3(m-1) \neq 1 \wedge y_1(m-1) \geqslant y_2(m-1)/2]$ . . . left path
$\vee\ [y_3(m-1) \neq 1 \wedge y_1(m-1) < y_2(m-1)/2]$ . . . right path.

We try to substitute using the recurrence equations for the left path in $y_3(m-1) \neq 1 \wedge y_1(m-1) \geqslant y_2(m-1)/2$, and the recurrence equations for the right path in $y_3(m-1) \neq 1 \wedge y_1(m-1) < y_2(m-1)/2$. For the left path, we have the recurrence relations

$$\left.\begin{aligned} y_1(m) &= y_1(m-1) - y_2(m-1)/2 \\ y_2(m) &= y_2(m-1)/2 \\ y_3(m) &= y_3(m-1)/2, \end{aligned}\right\} \quad \text{. . . left path}$$

and for the right path we have

$$\left.\begin{aligned} y_1(m) &= y_1(m-1) \\ y_2(m) &= y_2(m-1)/2 \\ y_3(m) &= y_3(m-1)/2. \end{aligned}\right\} \quad \text{. . . right path.}$$

Using these equations we obtain

$m = 0$
$\vee\ [2y_3(m) \neq 1 \wedge y_1(m) \geqslant 0]$ . . . left path
$\vee\ [2y_3(m) \neq 1 \wedge y_1(m) < y_2(m)]$ . . . right path.

Equivalently, we can write

(7) $[m=0 \vee 2y_3(m) \neq 1] \wedge [m=0 \vee y_1(m) \geqslant 0 \vee y_1(m) < y_2(m)]$.

For the first conjunct, we can eliminate the $m=0$ alternative because by (B5) we have $y_3(0) \geqslant 1$ at M, and therefore $2y_3(0) \neq 1$ at M. We have therefore the invariant

(B8) $2y_3 \neq 1$ at M.

For the second conjunct of (7), we can eliminate the $m=0$ alternative because $y_1(0)$ at M is either 0 or x_1, and thus $y_1(m) \geqslant 0$ is true when $m=0$. We have the invariant

(B9) $y_1 \geqslant 0 \vee y_1 < y_2$ at M.

So far we have used only the algorithmic techniques on the lower block, and have directly generated invariants (B6), (B7), (B8), and (B9) at M. Now we illustrate how some of the heuristic methods could be applied in order to obtain additional candidates.

Turning to heuristic 2.1, we consider each disjunct of (B9) separately. It is straightforward to show that both $y_1 \geqslant 0$ and $y_1 < y_2$ are invariants at M, i.e. we may add

(B10) $y_1 \geqslant 0 \wedge y_1 < y_2$ at M.

Using heuristic 2.4, we push forward to M the invariants in (B5), and among the candidates obtained is $y_2 = x_2 \cdot y_3$. We actually already directly discovered $y_2 = x_2 \cdot y_3$ in (B6). However, we also have the candidate $y_3 \geqslant 1$. Since, using the invariant $y_3 = 2^n/2^m$ from (B6), we can prove that for both paths around the loop

$\forall \bar{y}[y_3 \geqslant 1 \wedge y_3 = 2^n/2^m \wedge y_3 \neq 1 \supset y_3/2 \geqslant 1]$,

we have the new invariant

(B11) $y_3 \geqslant 1$ at M.

In turn this can be used, along with invariants $y_2 = x_2 \cdot 2^n/2^m$ and $y_3 = 2^n/2^m$ from (B6), to show that the candidate $y_1, y_2, y_3, y_4 \in \{\text{integers}\}$ (also obtained by pushing forward from L) is an invariant, i.e.

(B12) $y_1, y_2, y_3, y_4 \in \{\text{integers}\}$ at M.

Observe that if we had used heuristic 2.3 to push the given output specification at H backwards to M at an early stage, we could have obtained the important candidates for invariants $x_1 = y_4 \cdot x_2 + y_1$, $0 \leqslant y_1$, $y_1, y_4 \in \{\text{integers}\}$, and $y_1 < x_2$ directly by this method. As shown, the first three candidates are indeed invariants at N, while any attempt to establish the fourth candidate $y_1 < x_2$ will fail.

Now, by pushing forward to H the invariants (B6) to (B12) at M, adding the exit test $y_3 = 1$, and simplifying, we obtain the invariants

(B13) $y_3 = 1 \wedge y_2 = x_2 \wedge n = m \wedge x_1 = y_4 \cdot x_2 + y_1 \wedge 0 \leqslant y_1 < x_2 \wedge y_1, y_2, y_3, y_4 \in \{\text{integers}\}$ at H. □

3. Correctness and Incorrectness

As indicated in the Introduction, invariants may be used to prove correctness or incorrectness of a program. In order to place these properties into their proper framework, we first present some basic definitions and lemmas (which follow [20]; see also [21]).

(a) A program P *terminates over* $\phi(x)$ if for every input $\bar{a}$ such that $\phi(\bar{a})$ is true, the execution reaches a HALT statement.

(b) A program P is *partially correct w.r.t.* $\phi(x)$ *and* $\psi(x, y)$ if for every input $\bar{a}$ such that $\phi(\bar{a})$ is true, whenever the program terminates with some $\bar{b}$ as the final value of y, $\psi(\bar{a}, \bar{b})$ is true.

(c) A program P is *totally correct w.r.t.* $\phi(x)$ *and* $\psi(x, y)$ if for every input $\bar{a}$ such that $\phi(\bar{a})$ is true, the program terminates with some $\bar{b}$ as the final value of y and $\psi(\bar{a}, \bar{b})$ is true.

We are interested in proving that a program is either totally correct (*correct*) or not totally correct (*incorrect*). We introduce termination and partial correctness because together they are equivalent to total correctness, and, as we shall see, for a proof technique based on invariants it is easier to prove these two properties separately rather than to prove total correctness directly.

The Lemmas B–D and B′–D′ (Table I) use the invariants $\{q_h(x, y)\}$ at the haltpoints to provide criteria for proving termination, partial correctness, total correctness, and their negations. For clarity we have used an informal abbreviated notation. Lemma B, for example, should be stated as:

LEMMA B. *A program P terminates over* ϕ *if and only if for every set of invariants* $\{q_h(x, y)\}$ *and every input* $\bar{x}$ *such that* $\phi(\bar{x})$ *is true, there exists a haltpoint h such that* $\exists y[q_h(\bar{x}, y)]$ *is true.*

PROOF. If the program terminates, then for every input $\bar{x}$ satisfying $\phi(\bar{x})$ some haltpoint h must be reached and y will naturally have some value $\bar{b}$ at h. Then, by the definition of an invariant, for every set of invariants, $q_h(\bar{x}, \bar{b}_0)$ must be true, i.e. $\exists y[q_h(\bar{x}, y)]$ is true.

In order to prove the Lemma B in the other direction, we introduce the notion of a *minimal invariant* at cutpoint i, denoted by $m_i(x, y)$. A minimal invariant $m_i(\bar{a}, \bar{b})$ is true for some input $\bar{a}$ satisfying $\phi(\bar{a})$ and for some $\bar{b}$ *if and only if* during execution with input $\bar{a}$ the cutpoint i is reached with $y = \bar{b}$. Thus $m_i(x, y)$ denotes the exact domain of the y values which occur at i during execution of the program with input $\bar{x}$.[4]

Now we assume that for every set of invariants and every $\bar{x}$ such that $\phi(\bar{x})$ is true, there exists a haltpoint h such that $\exists y[q_h(\bar{x}, y)]$ is true. This is also true for the set of minimal invariants. By the definition of minimal invariant, since there exists a y such that

[4] Note that from its definition $m_i(\bar{x}, y)$ always exists as a predicate; for our purposes it is irrelevant how this predicate is expressed.

$m_h(x, y)$ is true, that y value actually occurs during execution at the haltpoint h, i.e. h must be reached, and the program must therefore terminate. □

The other lemmas may be proved by using similar arguments.

The six lemmas of Table I can be divided into two groups. The first group, Lemmas B′, C, and D′, are expressed in terms of the existence of a single set of invariants $\{q_h(x, y)\}$ (an "$\exists\bar{q}$ formula"). They therefore may be used to prove nontermination, partial correctness, and incorrectness, respectively, by demonstrating a set of invariants which satisfies the appropriate formula. The techniques of Sections 1 and 2 can be used to produce such a set of invariants. Lemmas B, C′, and D, on the other hand, are expressed in terms of every possible set of invariants $\{q_h(x, y)\}$ (a "$\forall\bar{q}$ formula"), and may not be used directly with our techniques.

Since total correctness is expressed by a $\forall\bar{q}$ formula, we try to prove this property by showing partial correctness and termination separately. Lemma C uses an $\exists\bar{q}$ formula, and therefore can be used to prove partial correctness. This lemma in fact represents "Floyd's method" [18] for proving partial correctness. The problem of termination, however, remains since it is expressed in terms of a $\forall\bar{q}$ formula. Termination must therefore be treated by other means, which will be discussed at the end of this section.

Incorrectness, on the other hand, is expressed by an $\exists\bar{q}$ formula, and therefore can be proven directly by our techniques, using Lemma D′. Note that the formula of this lemma can be expressed alternatively as

$$\exists\bar{q}\exists x\forall h\forall y[\sim q_h(x, y) \vee \sim\psi(x, y)],$$

i.e. for some input x either the program does not terminate, or the final result is incorrect.

We first illustrate the use of Lemma C for proving partial correctness.

Example B (continued). We would like to show that program B of Figure 5 is partially correct w.r.t.

$$\phi(x) : x_1 \geq 0 \wedge x_2 > 0 \wedge x_1, x_2 \in \{\text{integers}\}, \text{ and}$$
$$\psi(x, y) : x_1 = y_4 \cdot x_2 + y_1 \wedge 0 \leq y_1 < x_2 \wedge y_1, y_4 \in \{\text{integers}\}.$$

Using invariants (B1) to (B4) at N, (B5) at L, and (B6) to (B12) at M, we have established the invariants (B13) at H (the only haltpoint of the program). Since (B13) contains the invariants

$$x_1 = y_4 \cdot x_2 + y_1 \wedge 0 \leq y_1 < x_2 \wedge y_1, y_4 \in \{\text{integers}\},$$

we clearly have that

$$\forall x\forall y[q_H(x, y) \supset \psi(x, y)].$$

Thus, by Lemma C, program B is partially correct w.r.t. ϕ and ψ.

Note that (B13) actually contains additional information about the final values of the variables, namely that

$$y_3 = 1 \wedge y_2 = x_2 \wedge n = m$$

at the haltpoint H. □

Thus to prove partial correctness, we merely exhibit the invariants at the haltpoints which fulfill Lemma C. On the other hand, in order to prove incorrectness we must provide, in addition to appropriate invariants, an input value x_0 satisfying $\phi(x_0)$ such that the formula in Lemma D′ is true. We would like to develop candidates for x_0 in a systematic manner, similar to the way invariants were generated in Sections 1 and 2. For this reason, it is desirable to find a predicate $\phi'(x)$ which specifies a nonempty subset of the legal inputs for which the program is incorrect, rather than merely demonstrating the incorrectness for a single x. That is, to establish incorrectness we prove that for some $\phi'(x)$,

$$\forall x[\phi'(x) \supset \phi(x)] \wedge$$
$$\exists x\phi'(x) \wedge \exists\bar{q}\forall x\forall h\forall y[\phi'(x) \wedge q_h(x, y) \supset \sim\psi(x, y)].$$

In general, a proof which establishes incorrectness for a large set of input values is also more useful for the diagnosis and correction of the logical errors than an incorrectness proof for a single input value (see Section 4).

We will develop candidates for $\phi'(x)$ by starting with $\phi(x)$ and adding conjuncts (restrictions) to $\phi(x)$ one after another as the need arises. Thus $\phi'(x) \supset \phi(x)$ will be guaranteed true. In case there are several alternative restrictions at some stage of the process, we prefer adding the weakest possible, so that $\phi'(x)$ will allow maximal freedom in choosing additional restrictions later. At each stage, it is, of course, necessary to demonstrate that $\phi'(x)$ is satisfiable.

Note that all invariants which have been proven for $\phi(x)$ will remain true for any $\phi'(x)$ specifying a subset of $\phi(x)$. Moreover, at each stage of the process we now may discover additional invariants which are true for every x satisfying $\phi'(x)$ but are not necessarily true for every x satisfying $\phi(x)$.

Example A (continued). An attempt to prove the partial correctness of program A (Figure 4) will not succeed. Although the invariants (A10) at H can be used to establish $y_4 \leq x_1/x_2$ since

$$\forall x\forall y[y_4 \leq x_1/(2x_2) \supset y_4 \leq x_1/x_2],$$

we are unable to establish $x_1/x_2 - x_3 < y_4$. Thus we turn to incorrectness, trying to show that for some $\phi'(x)$ which specifies a nonempty subset of the legal inputs, and for some invariants $q_H(x, y)$ at H, we have

$$\forall x\forall y[\phi'(x) \wedge q_H(x, y) \supset x_1/x_2 - x_3 \geq y_4].$$

We first could try to show that the program is incorrect for *every* legal input x, i.e. to let $\phi'(x)$ be $\phi(x)$ itself. Such an attempt will fail. To find a candidate $\phi'(x)$, we notice that the "desired" conjunct is $y_4 \leq x_1/x_2 - x_3$, and that the invariant $y_4 \leq x_1/(2x_2)$ at H of (A10) also

Table I. Applications of the Invariants $\{q_h(\bar{x}, \bar{y})\}$.

LEMMA B. P terminates over ϕ if and only if
$\forall\bar{q}\forall\bar{x}\exists h\exists\bar{y}[q_h(\bar{x}, \bar{y})]$.

LEMMA C. P is partially correct w.r.t. ϕ and ψ if and only if
$\exists\bar{q}\forall\bar{x}\forall h\forall\bar{y}[q_h(\bar{x}, \bar{y}) \supset \psi(\bar{x}, \bar{y})]$.

LEMMA D. P is (totally) correct w.r.t. ϕ and ψ if and only if
$\forall\bar{q}\forall\bar{x}\exists h\exists\bar{y}[q_h(\bar{x}, \bar{y}) \wedge \psi(\bar{x}, \bar{y})]$.

LEMMA B'. P does not terminate over ϕ if and only if
$\exists\bar{q}\exists\bar{x}\forall h\forall\bar{y}[\sim q_h(\bar{x}, \bar{y})]$.

LEMMA C'. P is not partially correct w.r.t. ϕ and ψ if and only if $\forall\bar{q}\exists\bar{x}\exists h\exists\bar{y}[q_h(\bar{x}, \bar{y}) \wedge \sim\psi(\bar{x}, \bar{y})]$.

LEMMA D'. P is incorrect w.r.t. ϕ and ψ if and only if
$\exists\bar{q}\exists\bar{x}\forall h\forall\bar{y}[q_h(\bar{x}, \bar{y}) \supset \sim\psi(\bar{x}, \bar{y})]$.

where
$\forall\bar{q}$ means "for every set of invariants $\{q_h(\bar{x}, \bar{y})\}$."
$\exists\bar{q}$ means "there exists a set of invariants $\{q_h(\bar{x}, \bar{y})\}$."
$\forall\bar{x}$ means "for every input $\bar{x}$ such that $\phi(\bar{x})$ is true."
$\exists\bar{x}$ means "there exists an input $\bar{x}$ such that $\phi(\bar{x})$ is true."
$\forall h$ means "for every haltpoint h."
$\exists h$ means "there exists a haltpoint h."

provides an upper bound on y_4 in terms of $\bar{x}$. This suggests using the transitivity of inequalities to find an $r(\bar{x})$ such that

$$[y_4 \leq x_1/(2x_2) \wedge r(\bar{x})] \supset y_4 \leq x_1/x_2 - x_3 .$$

The "most general" candidate for $r(\bar{x})$ is clearly $x_1/(2x_2) \leq x_1/x_2 - x_3$, or equivalently,

$$r(\bar{x}) : x_3 \leq x_1/(2x_2).$$

The trial $\phi'(\bar{x})$ will therefore be $\phi(\bar{x}) \wedge r(\bar{x})$, i.e.

$$\phi'(\bar{x}) : 0 \leq x_1 < x_2 \wedge 0 < x_3 \wedge x_3 \leq x_1/(2x_2).$$

From the development of $\phi'(\bar{x})$, it is obvious that $y_4 \leq x_1/x_2 - x_3$ is an invariant at H for every $\bar{x}$ satisfying $\phi'(\bar{x})$. Thus to establish incorrectness it only remains to show that $\phi'(\bar{x})$ is satisfiable. Since we may first choose any x_1 and x_2 such that $0 \leq x_1 < x_2$, and then choose any x_3 such that $0 < x_3 \leq x_1/(2x_2)$, the satisfiability of $\phi'(\bar{x})$ is obvious. □

Recall that we have not yet provided a practical method for proving termination. The difficulty arose from the fact that Lemma B of Table I requires proving a "$\forall\bar{q}$ formula." Therefore we clearly need a special method for proving termination.

The traditional method suggested by Floyd in [8] involved choosing a well-founded set $(W, \succ)$, where $\succ$ is a partial ordering having the property that there is no infinitely descending chain of elements from W, $w_1 \succ w_2 \succ \ldots$. For every cutpoint i, one must find a partial function $u_i(\bar{x}, \bar{y})$ which maps the elements of the variables' domain into W, and an invariant $q_i(\bar{x}, \bar{y})$ which serves to restrict the domain of u_i. A proof of termination requires showing that each time control moves from cutpoint i to cutpoint j (along a path which includes no other cutpoints and which is a part of some loop), $u_i(\bar{x}, \bar{y}) \succ u_j(\bar{x}, \bar{y})$. Intuitively, since by definition there is no infinitely decreasing chain of elements in any well-founded set, the proof implies that no execution path of the program can be infinitely long.

The use of Floyd's method entails choosing the appropriate well-founded set $(W, \succ)$, the functions $\{u_j(\bar{x}, \bar{y})\}$, and the invariants $\{q_j(\bar{x}, \bar{y})\}$. We will suggest an alternative method for proving termination which will be strongly oriented toward the use of invariants, so that we may take advantage of the techniques of Sections 1 and 2. We present the method briefly.

As explained in the Preliminaries section, it is assumed that we can divide the given program into blocks in such a way that every block has only one top-level loop (in addition to possible "lower-level" loops already contained in inner blocks). We treat the innermost blocks first, and work outwards. Thus for each block we can consider only its top-level loop (with a unique cutpoint), assuming its inner blocks are known to terminate.

We suggest proving termination of a block with cutpoint i and counter n (assuming that the inner blocks terminate) by finding invariants which will imply that n is absolutely bounded from above at i. That is, $n \leq c_i$ at i for some constant c_i. Therefore the cutpoint cannot be reached infinitely many times during computation. Note that it is actually sufficient to show $a_i(\bar{x}, n) \leq b_i(\bar{x})$ where $a_i(\bar{x}, n)$ is an integer-valued function monotonic in n (i.e. if n increases in value, so does $a_i(\bar{x}, n)$). We therefore state

LEMMA E (termination). *A program P terminates if and only if there exists a set of invariants $\{q_j\}$ and functions $\{a_j\}$ and $\{b_j\}$ such that for every block B with cutpoint i and counter n,*

$$(8)\ \forall\bar{x}\forall\bar{y}\forall n[q_i(\bar{x}, \bar{y}, n) \supset a_i(\bar{x},n) \leq b_i(\bar{x})],$$

where $a_i(\bar{x}, n)$ is an integer-valued function monotonic in n.

The practical importance of the above Lemma E is that we may use invariants which link n to the program variables to derive directly the appropriate functions a_i and b_i. Recall that in such programs, we have the "built-in" invariant that n is a strictly increasing nonnegative integer. We shall use these properties in our

examples without explicit indication. Although n and $a_i(x, n)$ are integers, $b_i(x)$ and the program variables y need not be integers, and this technique is perfectly applicable to programs with real numbers, strings, etc. Lemma E can be proven formally by reduction to Floyd's method.

One can weaken the termination condition (8) of Lemma E in several different ways. For example, we can often generate $R(x, y)$, the union of the conditions for following a path from i to i in B. We may then use it in proving that the counter is bounded, since if $R(x, y)$ is false, the loop will terminate anyway. Another possibility is to use in a_i and b_i all those variables of y (and counters), denoted by y', which are not changed in B. Thus it actually suffices to prove the weaker condition

$$(9)\quad \forall x \forall y \forall n[q_i(x, y, n) \wedge R(x, y) \supset a_i(x,y',n) \leq b_i(x,y')].$$

Example A (continued). Consider again Program A of Figure 4. From $\phi(x)$ and invariant (A2) we note that $0 < x_3 \wedge y_3 = 1/2^n$ is an invariant at N. Thus, since

$$\forall x \forall y \forall n[0 < x_3 \wedge y_3 = 1/2^n \wedge y_3 > x_3 \supset 2^n < 1/x_3]$$

is true, it follows by Lemma E that the program terminates over $\phi(x)$. □

Example B (continued). Consider Program B of Figure 5. Using the known invariant (B1), $y_2 = x_2 \cdot 2^n$ at N, and $\phi(x)$ we obtain

$$\forall x \forall y \forall n[x_2 > 0 \wedge y_2 = x_2 \cdot 2^n \wedge y_2 < y_1 \supset 2^n < y_1/x_2].$$

Since y_1 is unchanged in the upper block, it follows by (9) that the upper block terminates.

For the lower block we use the invariants (B6) and (B11), $y_3 = 2^n/2^m \wedge y_3 \geq 1$ at M, and obtain

$$\forall x \forall y \forall n \forall m[y_3 = 2^n/2^m \wedge y_3 \geq 1 \supset 2^m \leq 2^n].$$

Since n is unchanged in the lower block, the termination of this block also follows by (9). □

The reader should not be misled into assuming that proving termination is always as trivial as it seems here. The method of Lemma E is examined in greater detail (and presented with some nontrivial examples) in [16].

Note that the method of Lemma E, as well as Floyd's original method, is useful only for showing termination. If we want to prove nontermination, both are impractical (again, *all* possible q_i's must be checked). Thus Lemma B′ should be used.

4. Automatic Debugging

In this section we suggest a method for debugging based on the invariants generated from the program. The technique we describe uses the invariants and information about how they were generated in order to modify the program systematically (and, at least potentially, automatically). For a more complete presentation of this particular aspect of logical analysis, along with an assessment of the remaining difficulties, see [15].

As explained in the Introduction, failure to prove correctness still leaves us unable to decide whether the program is actually correct (but, despite all our efforts, we are unable to prove it so), or the program is really incorrect (and we should not waste more time trying to prove it correct). Two differing philosophical approaches to automatic debugging can be applied as soon as we are unable to prove (total) correctness of a program.

Following what may be termed the *conservative approach*, we would insist on a proof of incorrectness before proceeding to modify the program. This is a reasonable view, and, as will be indicated below, a proof of incorrectness can aid in debugging. The method presented for proving incorrectness of programs was motivated by this approach.

However, proofs of incorrectness are often difficult to obtain, in particular for subtle errors, since the needed $\phi'(x)$ (of inputs leading to incorrectness) must be demonstrated. Thus an alternative to the conservative approach, a *radical approach*, can also be justified. In this approach, we will "fix" the program so that a proof of correctness is guaranteed to succeed, even without having proven that the original program is incorrect. In effect, under this approach we modify a program we merely *suspect* of being incorrect, taking the risk of modifying an already correct program.

The basic debugging technique using invariants is common to both approaches. We shall first describe the technique as it is used under the radical approach. The slight differences which arise if the conservative approach has been used (i.e. if a proof of incorrectness is available) are pointed out later in this section. At the end of the section we briefly compare the two approaches.

For simplicity we will again deal with a simplified model: a single block having no inner blocks, with a cutpoint L at the entrance, N inside the loop, and M at the exit, as in Figures 2 or 3. In addition to the candidates produced and invariants proven for each cutpoint during the process of invariant generation, we assume candidates $s''(x, y)$ at M which would guarantee partial correctness of the program were they actually invariants. For the case in which M is a haltpoint, $s''(x, y)$ would naturally be the output specification itself.

In order to effectively use the invariants for debugging, it is necessary to record in an *invariant table* all the information required to establish each invariant, e.g. the rule applied, and precisely how the program statements and/or other invariants were used in its derivation. In general there will be an entire invariant table

associated with each cutpoint. However, there is usually an essential difference in the complexity of the table for cutpoints on a loop, like N, and for those not on a loop, like M. All of the invariants at M, for example, will be obtained simply by "pushing forward" either invariants at N, or the exit condition of the block. Thus below we concentrate on the more interesting case of the invariant table at N.

For clarity, we will use a more pictorial representation for the invariant table at N and arrange the invariants generated in the form of a directed acyclic graph (dag). We use terminology similar to that of trees, talking about the "ancestors" or "descendants" of an invariant, and of moving "up" or "down" the graph. For this reason we refer to the graph as an *invariant tree*. We will have invariants from previous blocks given in $p(x, y)$, the initial assignment statements of the block, and the statements of the loop at the top of the tree. Each invariant $q(x, y)$ at N is the descendant of the loop statements, initial assignments, and other invariants used to establish $q(x, y)$.

By examining such an invariant tree, we can see both how a desired change in any given statement will affect the various invariants, and (conversely) how a desired change in an invariant can be achieved by changing statements.

The basic steps in correcting the program are as follows (again referring to Figures 2 or 3):

1. Using the heuristic methods of Section 2, such as 2.3, generate candidates for invariants $q''(x, y)$ at N which would allow proving the candidates $s''(x, y)$ at M to be invariants, and thus would allow proving partial correctness.[5] It is also possible to generate candidate exit tests $t'(x, y)$ or candidate exit functions $h'(x, y)$ which would guarantee partial correctness along with the existing invariants at N. In the continuation, we discuss changing only the invariants at N, although similar considerations apply to changing the exit test or exit function.

2. Find actual invariants $q(x, y)$ in the invariant tree which are "similar" to those candidates $q''(x, y)$ which guarantee correctness. The precise definition given to "similarity" will have a direct influence on the kinds of errors which may be corrected, and there are obviously many possibilities. We here define two predicates to be *similar* if they differ only in constant (nonzero) coefficients of variables, a constant term, or other minor perturbations in the relation involved, such as $<$ in place of $\leq$. When we have succeeded in finding invariants $q(x, y)$ in the tree similar to candidates $q''(x, y)$, the candidates will be called the *goal candidates* at N, and denoted $q^*(x, y)$.

3. Attempt to replace $q(x, y)$ by the similar goal candidates $q^*(x, y)$, moving up the tree and modifying the ancestors of $q(x, y)$ so that the new $q^*(x, y)$ will be derived rather than the former $q(x, y)$.

4. When a *statement* has been modified in order to allow deriving a goal candidate, inspect (by moving down the tree) the effect of the modification on all other invariants derived from it. This is necessary in order to ensure that no other part of the proof of partial correctness or the proof of termination are disturbed. The inspection could require making additional "compensatory" changes in other statements, or abandoning a possible change.

Example A (continued). Consider once again program A of Figure 4. The invariant tree for the program is shown in Figure 6. For simplicity, we have merely listed the number of the rule which was applied to obtain each invariant, rather than including more information. A brief review of the generation of invariants for this example (in Sections 1 and 2) should make the tree clear (except for (A11), which should be momentarily ignored). We have added the "termination" and "partial correctness" boxes at the bottom of the tree to emphasize which statements and invariants were used to prove termination (with bound $2^n < 1/x_3$) and partial correctness (w.r.t. $y_4 \leq x_1/x_2$). Recall that we were unable to prove partial correctness for $x_1/x_2 - x_3 < y_4$, the first conjunct of the output specification. In order to demonstrate the radical approach, we momentarily ignore the fact that in Section 3 we actually have proven this program incorrect.

The problematic part of the output specification, $x_1/x_2 - x_3 < y_4$, is automatically a candidate for an invariant at H. Using heuristic 2.3, we can generate candidates for invariants at N based on the candidate at H (assuming temporarily that the exit test $y_3 \leq x_3$ is correct). The strongest candidate at N is $x_1/x_2 - x_3 < y_4$ itself. We may also use the transitivity of inequalities with $x_1/x_2 - x_3 < y_4$ and the exit condition $y_3 \leq x_3$ to suggest another natural candidate. We need a $q(x, y)$ such that

$$q(x, y) \wedge y_3 \leq x_3 \supset x_1/x_2 - y_4 < x_3,$$

and see easily that the most general $q(x, y)$ which will do this is $x_1/x_2 - y_4 < y_3$, or $x_1/x_2 < y_3 + y_4$.

Naturally, if either of these candidates could be proven to be an invariant at N, the program already would have been proven correct. Now we turn to the invariant tree in order to modify the program so that a correctness proof is possible. We look for invariants already in the tree which are similar to the above candidates, and also try to combine existing invariants into new ones similar to the candidates.

For the candidate $x_1/x_2 - x_3 < y_4$, we find no similar invariant. For the second candidate, $x_1/x_2 < y_3 + y_4$, we may combine (A3), (A4), and (A7), giving

$$(10)\quad [y_2 = x_2 \cdot y_3 \wedge y_1 = 2x_2 \cdot y_4 \wedge x_1 < y_1 + 2y_2] \supset x_1/x_2 < 2y_3 + 2y_4,$$

[5] The possibility that the program is partially correct but nonterminating will not be treated in our discussion; actually it would lead to a correcting process similar to that described here.

i.e. we have the new invariant

(A11) $x_1/x_2 < 2y_3 + 2y_4$ at N.

This is similar to the candidate, which we now will refer to as the goal candidate

(A11)* $x_1/x_2 < y_3 + y_4$ at N.

We have thus found a place to "hang" the candidate on the tree, and now must adjust the ancestors of (A11) (i.e. (A3), (A4), or (A7)) so that (A11)* will be derived instead. By examining eq. (10), it is not difficult to see that two of the most direct modifications among the many possibilities are

(a) leave (A3) and(A4) unchanged, but change
(A7) $x_1 < y_1 + 2y_2$ to (A7)′ $2x_1 < y_1 + 2y_2$; or

(b) leave (A7) unchanged, but change
(A3) $y_2 = x_2 \cdot y_3$ to (A3)′ $2y_2 = x_2 \cdot y_3$ and
(A4) $y_1 = 2x_2 \cdot y_4$ to (A4)′ $y_1 = x_2 \cdot y_4$.

Possibility (*a*) will be considered first. The invariant tree shows that (A7) was derived from the invariant

(A6) $x_1 < y_1 + 2y_2 \vee x_1 \geq y_1$ at N,

by using heuristic 2.1 to strengthen the invariant. To obtain (A7)′, we will first modify (A6) to

(A6)′ $2x_1 < y_1 + 2y_2 \vee h(x, y)$

where $h(x, y)$ is the part of (A6)′ not of interest to us at the moment. By tracing back through the derivation of (A6) (which used the algorithmic rule 1.2), the left alternative of (A6) can be seen to originate as

(i) $x_1 < y_1(n-1) + y_2(n-1)$ ... from the test $x_1 < y_1 + y_2$, using the right path

(ii) $y_1(n) = y_1(n-1)$ ... from the fact that y_1 is unchanged along the right path

(iii) $y_2(n) = y_2(n-1)/2$ ... from the assignment $y_2 \leftarrow y_2/2$.

These clearly were combined to yield the alternative $x_1 < y_1 + 2y_2$. To obtain $2x_1 < y_1 + 2y_2$ instead, we replace (i) by $2x_1 < y_1(n-1) + y_2(n-1)$, i.e. change the test statement $x_1 < y_1 + y_2$ to $2x_1 < y_1 + y_2$. This suggested change was built to yield an acceptable left alternative of (A6)′. Checking $2x_1 < y_1 + 2y_2$ alone, we may conclude that with this suggested change (A7)′ is indeed an invariant, and thus, so is the goal (A11)*.

We must now check whether any other vital invariants are affected. From the tree it is clear that the only effect could be on the right alternative of (A6) and its descendants. Using the new test statement, it is easy to see that the left path leads to

$2x_1 \geq y_1(n-1) + y_2(n-1)$ ... from the test $2x_1 < y_1 + y_2$, using the left path

$y_1(n) = y_1(n-1) + y_2(n-1)$ ... from the assignment $y_1 \leftarrow y_1 + y_2$ on the left path.

These clearly combine to yield $2x_1 \geq y_1(n)$, so that $h(x, y)$ is $2x_1 \geq y_1$, and we have the invariant

(A6)′ $2x_1 < y_1 + 2y_2 \vee 2x_1 \geq y_1$ at N.

Examining the descendants of (A6)′, we can see that (A8) must be replaced by

(A8)′ $2x_1 \geq y_1$ at N,

which is an invariant of the modified program. In turn, this combined with (A4) will yield the invariant

(A9)′ $y_4 \leq x_1/x_2$ at N.

Thus we also have the invariant $y_4 \leq x_1/x_2$ instead of $y_4 \leq x_1/(2x_2)$ at H.

However, this invariant serves just as well as the original $y_4 \leq x_1/(2x_2)$ to guarantee partial correctness for the output specification $y_4 \leq x_1/x_2$. Thus the suggested correction leads to the goal (A11)* and does not disturb any other aspect of the proof of correctness, i.e. the modified program is guaranteed correct. In Figure 7 we show the invariant tree at N of the modified program, which is totally correct. Thus, to summarize:

Replace the test $x_1 < y_1 + y_2$ *by* $2x_1 < y_1 + y_2$.

Possibility (*b*) for achieving the goal (A11)* will now be considered, i.e. we would like to replace (A3) and (A4) by (A3)′ and (A4)′, respectively (again referring to the original invariant tree of Figure 6). We immediately note that since (A3) is an ancestor of (A4), any change in (A3) will influence (A4). The invariant (A4) was obtained by bringing two summations involving **if-then-else** to an identical form, so that y_1 and y_4 could be connected. If during the manipulations of the relations, $2y_2 = x_2 \cdot y_3$ is used for substitution instead of $y_2 = x_2 \cdot y_3$, the new (A4) becomes exactly $y_1 = x_2 \cdot y_4$, i.e. the (A4)′ we require. Thus if we can change (A3) to (A3)′, we "automatically" have changed (A4) to (A4)′.

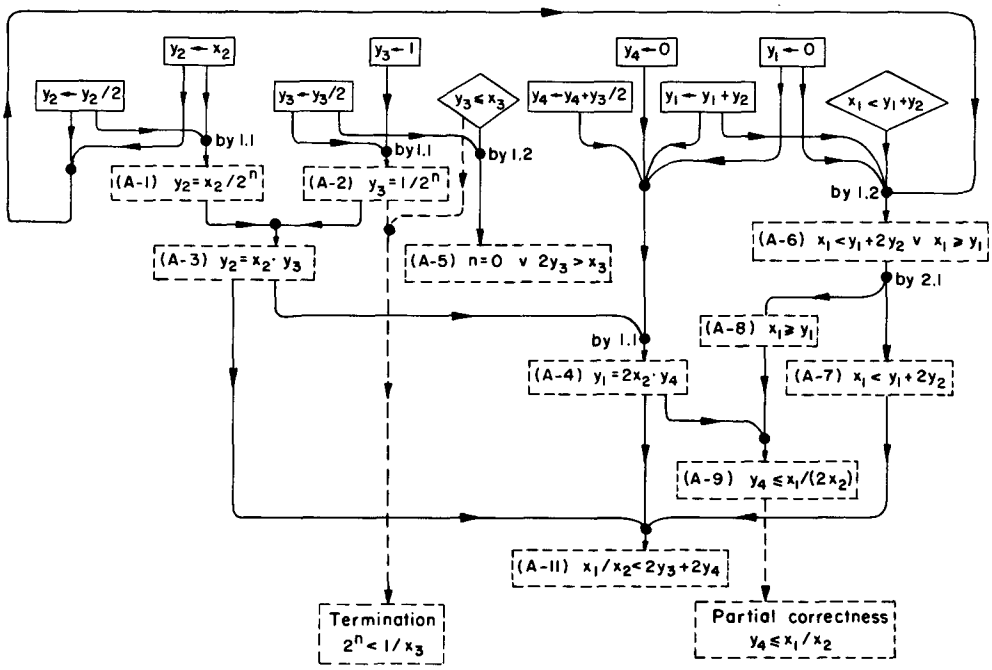
Examining the invariant tree, it is clear that we may achieve (A3)′ by changing either (A1) or (A2), i.e. either

(A1) $y_2 = x_2/2^n$ to (A1)′ $y_2 = x_2/2^{n+1}$ or

(A2) $y_3 = 1/2^n$ to (A2)′ $y_3 = 2/2^n$.

Since $y_2 = y_2(0)/2^n$ and $y_2(0) = x_2$, the first possibility can be achieved by letting $y_2(0) = x_2/2$, i.e. by changing the initialization $y_2 \leftarrow x_2$ to $y_2 \leftarrow x_2/2$. Now we check the possible effect of this change on other invariants. This initialization was used to establish (A6) and (A7) the first time N is reached, but the new initialization also does the same job. Tracing other paths down from this suggested change, we see that (A4) was used to establish $y_4 \leq x_1/(2x_2)$ at H. However, the new (A4)′, $y_1 = x_2 \cdot y_4$, may still be combined with (A8),

Fig. 6. The invariant tree for cutpoint N of program A.



$y_1 \leq x_1$, to show that $y_4 \leq x_1/x_2$ at N and thus at H. Therefore this change is also safe, and we have

Replace the initialization $y_2 \leftarrow x_2$ *by* $y_2 \leftarrow x_2/2$.

The change in (A2), from $y_3 = 1/2^n$ to $y_3 = 2/2^n$, is also easy to achieve, since $y_3 = y_3(0)/2^n$. Thus we set $y_3(0) = 2$ instead of $y_3(0) = 1$, i.e. change the initialization $y_3 \leftarrow 1$ to $y_3 \leftarrow 2$. This change will slightly affect the termination, but the counter n can now be bounded by $2^n < 2/x_3$. Again (A4)′ can be shown not to disturb the correctness for $y_4 \leq x_1/x_2$. Thus a third safe change is

Replace the initialization $y_3 \leftarrow 1$ *by* $y_3 \leftarrow 2$. □

So far in this section, we have ignored the possibility that we have already proven the program incorrect. Now we briefly consider how a proof of incorrectness can aid in the automatic debugging process described above.

We assume that when unable to prove correctness, the conservative approach was followed and a proof of incorrectness was produced. Although the existence of this proof has surprisingly little effect on the basic debugging technique, it can be of some aid. Clearly, any change in the program, which is intended to correct the error, must change at least one of the invariants used in the incorrectness proof. Thus the paths up the tree from the goal candidate can be restricted to those which will influence invariants from the proof of incorrectness. This is valuable because one of the difficulties with the use of the tree is the need for further guidance in the selection of likely paths.

Moreover, it is often possible to discover the smallest change in an invariant which will invalidate the proof of incorrectness. This then becomes a new source of candidates; such a candidate would not guarantee correctness, but at least would ensure that the existing incorrectness proof could no longer be applied.

Example A (continued). Let us review the proof of incorrectness of program A of Figure 4. We used the invariant $y_4 \leq x_1/(2x_2)$ at H (one of the invariants of (A10)) to find an $r(\bar{x})$ such that

$$[y_4 \leq x_1/(2x_2) \wedge r(\bar{x})] \supset y_4 \leq x_1/x_2 - x_3 .$$

This suggested taking $r(\bar{x}) : x_1/(2x_2) \leq x_1/x_2 - x_3$, since

$$(11)\ [y_4 \leq x_1/(2x_2) \wedge x_1/(2x_2) \leq x_1/x_2 - x_3] \supset y_4 \leq x_1/x_2 - x_3.$$

This $r(\bar{x})$ then led to

$$\phi'(\bar{x}) : 0 \leq x_1 < x_2 \wedge 0 < x_3 \wedge x_1/(2x_2) \leq x_1/x_2 - x_3$$

which was then simplified and shown to be satisfiable.

Since $y_4 \leq x_1/(2x_2)$ at H was the only invariant used in the proof of incorrectness and was obtained directly from the invariant (A9), $y_4 \leq x_1/(2x_2)$ at N, it follows that any correction of the program *must* change invariant (A9).

If we analyze the above incorrectness proof more closely, we can obtain some additional information about *how* invariant (A9) must be changed. We will try to find a new invariant in place of (A9) so that following the framework of the proof given, an unsatisfiable $\phi'(\bar{x})$ would result, thereby invalidating the proof of incorrectness. Since the terms y_4 and $x_1/x_2 - x_3$ are

fixed by the desired output specification, the only term of eq. (10) which can naturally be replaced is $x_1/(2x_2)$. We thus look for a term P such that $y_4 \leq P$ and

$$\phi'(\bar{x}) : 0 \leq x_1 < x_2 \wedge 0 < x_3 \wedge P \leq x_1/x_2 - x_3$$

is unsatisfiable. Simplifying the desired $\phi'(\bar{x})$, we see that we need $0 \leq x_1 < x_2 \wedge 0 < x_3 \supset \sim(P \leq x_1/x_2 - x_3)$, i.e.

$$0 \leq x_1 < x_2 \wedge 0 < x_3 \supset x_1/x_2 - P < x_3 .$$

If we let P be $(ax_1)/x_2$, for any $a \geq 1$, the above implication is clearly true, so the $\phi'(\bar{x})$ is unsatisfiable. Thus if we change the relevant invariant at H from $y_4 \leq x_1/(2x_2)$ to any $y_4 \leq (ax_1)/x_2$, where $a \geq 1$, the existing proof of incorrectness will not work. Since the invariant $y_4 \leq x_1/(2x_2)$ at H was obtained directly from invariant (A9) at N, we have a class of candidate invariants

(A9)′ $y_4 \leq (ax_1)/x_2$ for any $a \geq 1$ at N.

In general, such additional knowledge is valuable in restricting the possible alternatives which must be explored in the invariant tree. Note that all of the alternative changes previously found for the program "coincidentally" change (A9) to $y_4 \leq x_1/x_2$ by changing other invariants. □

Let us briefly compare the two approaches.

Because we guarantee correctness, the "radical" approach of modifying without first proving incorrectness is not as dangerous as it might seem. In fact, the only objection would seem to be that in the case of a program which actually was originally correct, the efficiency of execution may be reduced in a modified (also correct) version. From our experience with hand simulations, we believe that *if* we are able to find goal candidates similar to the invariants, the program is very likely incorrect, and it is worthwhile to follow the radical approach without first proving incorrectness.

However, for programs with a large number of errors (or a small number of very gross errors), it is unlikely that the required similarity will be found. "Gross" errors could actually be defined as those which lead to invariants completely irrelevant to a proof of the specification. In such a case, the radical approach will fail, but the conservative approach has a good chance of at least partial success. For a grossly incorrect program, a proof of incorrectness will generally be very easy to find, and the technique of invalidating the proof may even lead to a correction.

In any case, the proof of incorrectness would be a valuable aid to the user, even if an automatic correction could not be made. It provides what could be called *logical diagnostics* about the program. From the conjuncts of the output specification which were contradicted, the general effect of the error is obtained. From $\phi'(\bar{x})$, the user obtains a class of inputs for which the program is incorrect. Most importantly, from the invariants directly used in the proof of incorrectness, the user can identify the problematic relations in the program.

Conclusion

In this paper we have presented an overview of how invariants can be produced and used. The basic concept of an invariant is, of course, not new (e.g. [8, 12]). The term invariant has also been used previously, for example in [13]. We have, however, tried to present a new perspective which shifts emphasis from the limited task of verifying a correct program to the more general framework of logical analysis. From our perspective, invariants are "independent entities" which can be used for more than one purpose, only one of which is proving partial correctness when possible.

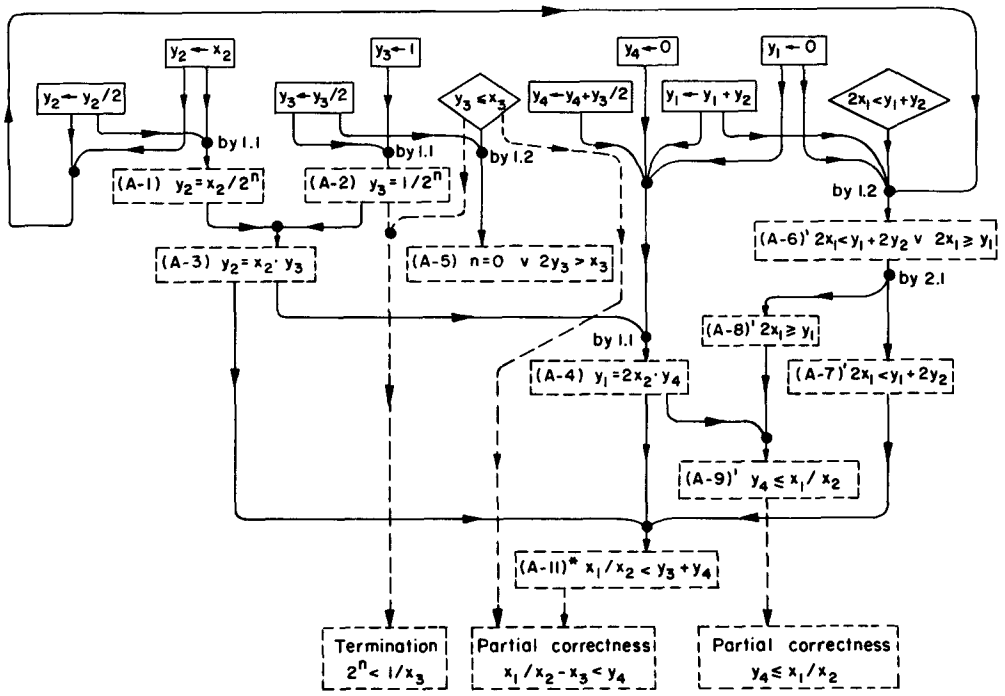
Numerous improvements and refinements are clearly possible to the invariant-generating techniques presented. In particular, it is necessary to further guide the heuristics in Section 2, so that they will not be applied indiscriminately. For example, only when the need for an invariant involving certain variables has become evident, should candidates involving those variables be generated.

The general problem of finding an algorithm to generate invariants for any program is unsolvable. Programs clearly exist with relations among the variables based implicitly on deep mathematical theorems which could not conceivably be rediscovered by any general invariant-generating algorithm.

In a practical implementation, the user would be encouraged to provide his own ideas about what the intermediate invariants should be ("comments"), and these will automatically be considered as candidates for invariants. The system could also ask the user to provide suggestions as the need arises for invariants involving specific problematic variables with unclear relationships at a certain cutpoint. We expect that a reasonably sophisticated system based on the techniques presented here, with some aid from the user whenever necessary, could produce sufficient invariants to conduct the logical analysis of some nontrivial programs.

Several other efforts have been made to attack the problem of finding inductive assertions which prove partial correctness. The earliest work is by Floyd (private communication, 1967), and Cooper [2]. Elspas [7] was the first to consider using recurrence relations. Wegbreit [28] has developed independently some rules similar to our heuristic approach, and a method using a "weak interpretation" of the program. These have been implemented by German [10]. In [14] the authors suggested additional heuristics to treat arrays. Grief and Waldinger [11] also described a method for generating assertions which moves backwards from the output specification. There is much current activity in finding new techniques for generating inductive assertions; for example, [3, 23, and 24].

Fig. 7. The invariant tree for cutpoint N of the modified program A (correction #1).



The idea of adding variables, such as counters, to the program in order to facilitate proofs of partial correctness or termination is not new. Knuth [19] uses a "time clock" incremented at every statement in order to prove termination. Elspas et al. [6] also discuss how such counters can be used to prove termination. Other related works on termination are those of Cooper [2], Maurer [22], and Sites [26].

The possibility of using a program verifier to debug programs is first discussed informally by King [18]. Sussman [25] stresses the importance of systematically eliminating bugs in the context of program synthesis. An attempt to establish incorrectness by finding counterexamples was outlined by Floyd [9] as part of his proposed system for interactive program writing. In our presentation we have basically considered the debugging of a program with a single loop. For more complicated programs, with multiple loops, additional research problems present themselves. What we have introduced here is clearly just a first step toward the use of invariants in debugging.

For the sake of completeness, a few additional application areas of logical analysis using invariants are mentioned below.

The area of *program optimization* is one natural application. Once a program has been proven correct and the "vital" invariants used in the proof identified, those invariants can be used to optimize the program. The basic idea is to maintain the vital invariants and exit conditions, or their equivalents, thereby guaranteeing the continued correctness of the program. However, the way in which the invariants and exit conditions are computed would be changed in order to increase the efficiency of the code produced. Using invariants can systematize the optimization process because "overzealous" optimizations which introduce errors are prevented. Flexibility is also increased because we do not restrict ourselves in advance to specific transformations and because it is easy to identify extraneous computation.

The problem of *modification*, where an existing (presumably correct) program is given a new input or output specification, differing only slightly from the original, can be reduced to error correction. The old program must be "corrected" to meet the new specification. Knowledge of the invariants allows this to be done without falling into the familiar pitfall of making some unchanged part of the specification untrue in the course of changing the original program (see [25]).

Finally, it should be noted that the techniques presented provide information on the (time) *complexity* and behavior of the given program. For example, in proving termination by showing counters bounded, we actually obtain upper bounds on the number of times the loops may be executed. It would be natural to also consider lower bounds on the counters immediately after exit from the block and to obtain automatically more sophisticated estimates of the total time required for each loop.

Acknowledgment. We are indebted to Ed Ashcroft, Nachum Dershowitz, Bernard Elspas, Stephen Ness, Tim Standish, and Richard Waldinger for their critical reading of the manuscript.

Received July 1974; revised January 1975.

References

1. Allen, F.E. A basis for program optimization. Proc. IFIP Cong. 71, Vol. 1, North-Holland Pub. Co., Amsterdam, 1971, pp. 385–390.
2. Cooper, D.C. Programs for mechanical program verification. *Machine Intelligence* 6, American Elsevier, New York, 1971, pp. 43–59.
3. Caplain, M. Finding invariant assertion for proving programs. Proc. Int. Conf. on Reliable Software, Los Angeles, Calif., April 1975.
4. Dahl, O.J., Dijkstra, E.W., and Hoare, C.A.R. *Structured Programming*. Academic Press, New York, 1972.
5. Deutsch, L.P. An interactive program verifier. Ph.D. Th., Dept. of Computer Sci., U. of California, Berkeley, June 1973.
6. Elspas, B., Levitt, K.N., and Waldinger, R.J. An interactive system for the verification of computer programs. Research Rep., SRI, Menlo Park, Calif., Sept. 1973.
7. Elspas, B. The semiautomatic generation of inductive assertions for proving program correctness. Research Rep., SRI, Menlo Park, Calif., July 1974.
8. Floyd, R.W. Assigning meaning to programs. Proc. Symp. in Appl. Math., Vol. 19, J.T. Schwartz (Ed.), Amer. Math. Soc., Providence, R.I., 1967, pp. 19–32.
9. Floyd, R.W. Towards interactive design of correct programs. Proc. IFIP Cong., Vol. 1, North-Holland Pub. Co., Amsterdam, 1971, pp. 7–10.
10. German, S. M. A program verifier that generates inductive assertions. B.A. Th., Harvard U., May 1974.
11. Greif, I., and Waldinger, R. A more mechanical heuristic approach to program verification. Proc. Int. Symp. on Programming, Paris, April 1974, pp. 83–90.
12. Hoare, C.A.R. An axiomatic basis of computer programming. *Comm. ACM 12*, 10 (Oct. 1969), 576–580, 583.
13. Hoare, C.A.R. Proof of a program: FIND. *Comm. ACM 14*, 1 (Jan. 1971), 39–45.
14. Katz, S.M., and Manna, Z. A heuristic approach to program verification. Proc. 3rd Int. Conf. on Artificial Intelligence, Stanford U., Aug. 1973, pp. 500–512.
15. Katz, S.M., and Manna, Z. Towards automatic debugging of programs. Proc. Int. Conf. on Reliable Software, Los Angeles, Calif., April 1975, pp. 143–155.
16. Katz, S.M., and Manna, Z. A closer look at termination. *Acta Informatica*, to appear.
17. King, J. A program verifier. Ph.D. Th., Dep. of Computer Sci., Carnegie-Mellon U., Pittsburgh, Pa., 1969.
18. King, J. A verifying compiler. In *Debugging Techniques in Large Systems*, Randall Rustin (Ed.), Prentice-Hall, Englewood Cliffs, N.J. 1970, pp. 17–39.
19. Knuth, D.E. *The Art of Computer Programming, Vol. I, Fundamental Algorithms*. Addison-Wesley, Reading, Mass. 1968.
20. Manna, Z. The correctness of programs. *J. Computer and System Sci., 3*, 2 (May 1969), 119–127.
21. Manna, Z. *Mathematical Theory of Computation*. McGraw-Hill, New York, 1974.
22. Maurer, W.D. The theory and practice of algorithm verification. ERL-M315, U. of California, Berkeley, Aug. 1973.
23. Misra, J. Relations uniformly conserved by a loop. Proc. Int. Symp. on Proving and Improving Programs, Arc et Senans, France, July 1975, pp. 71–80.
24. Moriconi, M.S. Towards the interactive synthesis of assertions. Research Rep., U. of Texas at Austin, Oct. 1974.
25. Sussman, G.J. A computational model of skill acquisition. Ph.D. Th., MIT, Cambridge, Mass., Aug. 1973.
26. Sites, R.L. Proving that computer programs terminate cleanly. Ph.D. Th., Dep. of Computer Science, Stanford U., STAN-CS-74-418, May 1974.
27. Waldinger, R. and Levitt, K.N. Reasoning about programs. *Artificial Intelligence 5* (1974), 235–316.
28. Wegbreit, B. The synthesis of loop predicates. *Comm. ACM 17*, 2 (Feb. 1974), 102–112.
29. Wensley, J.H. A class of non-analytical interactive processes. *Computer J., 1* (1958), 163–167.