PB-233 045

PROVING THAT COMPUTER PROGRAMS TERMINATE CLEANLY

Richard L. Sites

Stanford University
Stanford, California

May 1974

BIBLIOGRAPHIC DATA SHEET	1. Report No. STAN-CS-74-418	2	PB 233 045

4. Title and Subtitle

PROVING THAT COMPUTER PROGRAMS TERMINATE CLEANLY

5. Report Date: May 1974

6.

7. Author(s)

Richard L. Sites

8. Performing Organization Rept. No. STAN-CS-74-418

9. Performing Organization Name and Address

Stanford University
Computer Science Department
Stanford, California 94305

10. Project/Task/Work Unit No.

11. Contract/Grant No.

12. Sponsoring Organization Name and Address

IBM Corporation
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, New York 10598

13. Type of Report & Period Covered: technical, May 1974

14.

15. Supplementary Notes

16. Abstracts

A system of techniques is presented for proving mechanically that a computer program terminates cleanly. In this paper, clean termination means that the program has no infi ite loops and no semantic errors - no undefined variables, no subscripts out of range, no overflows on a given computer, etc. The techniques are discussed in terms of programs expressed as flow charts, and they have wide application to high-level languages.

The work described here complements work done on program correctness, differing particularly by not requiring a description of the correctness properties of a program and by treating the running of programs on machines with finite-range arithmetic.

17. Key Words and Document Analysis. 17a. Descriptors

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement

approved for public release; distribution unlimited.

19. Security Class (This Report) UNCLASSIFIED

21. No. of Pages 146

20. Security Class (This Page) UNCLASSIFIED

22. Price 4.75

Dedication

This thesis is dedicated to a certain place in a cow pasture behind the Stanford campus; a Hill without whom this thesis would never have been written. May all schools have the foresight to preserve such places for the lost souls who will need them.

Richard L. Sites
Dedicated January 11, 1973

Acknowledgments

The single most important factor in the completion of this thesis has been Don Knuth's willingness to read and extensively annotate early drafts of this and related papers. He has been such an outstanding thesis advisor for me that I could enjoy doing another thesis under him, just so I could learn how to be a good thesis advisor myself. I also am grateful for the support of Bob Floyd and Ben Wegbreit on my reading committee.

Financial assistance for the most difficult year of my thesis work was provided by the Fannie and John Hertz Foundation, and was offered for subsequent years in spite of my progress reports; I thank the Foundation for its long-range view and particularly for alleviating one of the stresses of completing a thesis. I also thank Hewlett-Packard, Inc. for its indirect support during the final year of this work.

The previous work of others whom I drew upon for technical support is acknowledged in references throughout the text.

Jim Dulev, Don Knuth, John Walters, Phyllis Winkler, and representatives of the Fannie and John Hertz Foundation provided continuing moral support. I thank them all.

Finally, heartfelt thanks to Susan Phoebe Watts, whose encouragement started my pursuit of a Ph.D. in the first place. May she find similar encouragement.

Table of Contents

Note to the reader

I have tried to structure this thesis so that it can be read at many different levels of detail. You have already passed the first level, the title. I have tried to write the introductory chapter so that you can see what the rest of the thesis is and is not about and how this work is different from others. Hopefully, after reading the introduction, you will have enough information to decide whether to read the rest. At the third level of detail, each chapter begins with a summary of its content. If your interests are very specific, this should allow you to skip the bulk of some chapters. The chapter summaries end with the symbol ⊗ . The fourth level is Appendix A. All of the examples in it should be readable if you have read just the chapter summaries. The rest of the thesis is at the fifth level of detail. For yet more detail, read all the references, their references, etc. (Proof of termination of the last step is left to you.)

For a quick reading, I would suggest the following order: Chapter 3, Example 1 in Appendix A, all the chapter summaries, then Example 10 in Appendix A. For reference purposes, on pages 138-139 there is an index, and on pages 66-68 there is a summary of the points covered in the Appendix A examples.

Chapter 0. Introduction

This thesis discusses techniques for proving that a computer program terminates cleanly -- that it always terminates and does so without encountering any semantic errors -- overflows, out-of-range subscripts, etc. In contrast with others' work on rigorous proofs of program correctness, this work only tangentially examines what a program does; the emphasis is on proving that whatever it does, a program always terminates normally. ⊗

Proof of clean termination is not an end in itself. Rather, it is a well-defined subgoal in convincing oneself that a program works reliably. Proving that a program does not "blow up" in the middle does not in any way say that the program correctly produces useful results; it just says that whatever the program does, it will eventually come to a normal end. For a large class of programs, it is useful to run a set of test cases to demonstrate that the program goes through its intended motions for at least those test cases, then to try to prove that the program terminates cleanly in order to discover anomalies that the test cases missed. The proof will pick up problems like:

(1) Degenerate cases of some data structure which the program did not anticipate and which result in, say, the use of a zero subscript in an array whose legal bounds are 1:100 .

(2) Degenerate cases where some loop exits before iterating at all, leaving some variables undefined (never assigned to) on exit.

(3) A programmer's assumption that, say, N is always positive, when in fact there is not an explicit test for this, and the program loops indefinitely if N = 0 .

(4) Use of uninitialized variables, which could make the program non-deterministic.

(5) Calculations on a small (say 16-bit) machine which could easily produce integer overflows and hence invalid results.

For some programs, this process is not very useful. For example, in a matrix inversion routine almost every arithmetic operation could produce an overflow or underflow, so the attempted proof of clean termination will fail miserably, flagging almost every statement as a possible place for an unclean termination.

For other programs, proof of clean termination may actually be an end in itself, as in certain real-time programs or operating system subsystems, where it may be all right for the program (or subsystem) to give wrong answers occasionally, but it would be disastrous if the program got in an infinite loop and impacted the operation of the rest of the system.

Proof of clean termination is a valuable tool because it is a well-defined problem which lends itself to being done almost entirely mechanically, with very little help from a user. Unlike rigorous proofs of correctness, which require the user to supply a carefully-constructed set of assertions about the program's behavior, proofs of clean termination can use mechanically generated assertions: based on each operator in the program, it is possible to synthesize a set of assertions about semantic errors, and based on some flow analysis of the program, it is possible to synthesize a set of assertions stating that each loop terminates. Attempting to prove these assertions then usually has the effect of finding that some of them aren't true and

hence suggesting to the user bugs to be fixed or an appropriate set of restrictions for the program's data. The user can then either change the program, or add tests to the program to detect data that can't be handled (and if detected, return a clean indication or message), or run the program as it stands, knowing that it will blow up in a possibly obscure way for some sets of inputs.

In contrast to work on algorithm correctness, the system described here deals explicitly with programs which fail because of finite-range arithmetic. In this regard, see London's certification of the algorithm TREESORT3 [London 1970b], in which he states "... it is possible and appropriate to certify algorithms with a proof of correctness. This certification would be in addition to, or in many cases instead of, the usual certification [by testing]", and Sites's certification of the program TREESORT3 [Sites 1974], in which he notes that the program can fail to sort large arrays because of an overflow in the subscript calculations, in spite of London's proof of correctness. The same issue is pointed out in London's reply to Redish [Redish 1971]. As minicomputers and microcomputers with small word lengths proliferate, the restrictions of finite-range arithmetic will become more important.

In contrast to work on partial correctness, the system described here deals explicitly with proof of termination. In this regard, see King's proof of partial correctness of a simple division program [King 1969], and the same Example 2 in Appendix A of this thesis, in which it is noted that King's proof of partial correctness includes the case of division by zero, for which the program loops indefinitely.

At this point, I will summarize the major limitations and results of the work described in subsequent chapters.

Limitations:

(1) There is no computer implementation of the techniques.

(2) Calculations with floating-point numbers are not handled, although Chapter 6 includes some discussion of the problems that would be involved.

(3) Recursion and asynchronous events are not handled.

(4) The system in fact requires a minimal amount of program annotation to be supplied by the user -- descriptions of the bounds of arrays passed to procedures, and descriptions of the intended structuring of linked lists and trees.

Results:

(1) The analysis of a program is based on an algorithm for the forward propagation of information while visiting the nodes of a program's flow graph in a fixed order. The last time a node is visited, all the assertions associated with it are either proved, disproved, or the theorem prover gives up. Proved assertions need not concern the user, disproved assertions represent definite bugs or hidden restrictions, and the remaining assertions represent possible problems on which the user should focus his attention.

(2) A second result is a set of techniques for untangling loops and eliding tests, an extension of the interval analysis and compiler optimization techniques of Cocke, Allen, et al. [Allen 1970] [Allen and Cocke 1972]. The technique for finding paths along which a test can be elided is important in the automatic synthesis of lexicographic orderings for proving termination of complex loops.

(3) Techniques are presented for proving the termination of some loops which do not lend themselves to mapping into monotonically

decreasing sequences, such as some search-for-equality loops and circularly-linked-list loops.

(4) Procedures, parameters (both name and value), read statements, and arrays are all explicitly treated.

(5) Specific programs which have been proved to terminate cleanly include TREESORT3 [Floyd 1964] [London 1970b] [Sites 1974]; SELECT, an algorithm for finding medians [Floyd and Rivest 1973] [Sites 1974]; an iterative version of McCarthy's 91 function [Manna et al. 1972]; and some of King's examples [King 1969] (see Examples 1-9 in Appendix A). Hand simulation of these proof techniques uncovered a hidden restriction in TREESORT3 and a simple bug in Knuth's Algorithm 2.3.3A [Knuth 1973b]. Preliminary work on this thesis included hand simulation of some of the techniques on a wide variety of programs: a list reversal routine, a symbol table search routine, Knuth's program for Dijkstra's inversion problem [Knuth 1973a], a floating-point calculation [Fritsch et al. 1973], a hash search routine [Brent 1973], and a list move routine [Reingold 1973].

In brief, proof of clean termination is a mechanical process, requiring little effort from the human user, which can do much of the tedious work of examining a program's behavior in all possible degenerate cases, for all possible sets of input data, and either report to the user an assurance that the program is free of an important class of errors, or report pieces of the program or sets of inputs which may fail. This process can be applied to programs for which we have no way of even expressing what it means for the program to be rigorously "correct".

Chapter 0. Introduction

Chapter 1. Flow Graph Processing

This chapter discusses preliminary modifications to the flow graph of a program to make its loop structure more tractable. The modifications consist of putting all loops in leading test form and inserting a "loophead" node at the beginning of each loop. Copies may be made of some nodes in the flow graph, either because of node splitting during interval analysis [Allen and Cocke 1972], or because of permuting the nodes in a loop to bring an exit test to the front of the loop. The nodes in the modified flow graph are then ordered so that when a node is encountered in subsequent processing, all of its predecessors (and any loops containing them but not the current node) will have already been processed.

For programs which have already been put in while format (perhaps using techniques described in [Ashcroft and Manna 1972]), the processing described in this chapter can be skipped, except for the insertion of "loophead" nodes and ordering the nodes. ⊗

In this paper, we shall view all programs as flow graphs consisting of nodes and directed arcs. Our flow graphs have seven kinds of nodes: binary test, assignment, START, HALT, PROCEDURE, RETURN, and CALL. The last three aren't strictly necessary, but they make the discussion of subroutines easier. All high-level flow-of-control constructs are mapped into tests and assignments. Thus, Algol 60 FOR loops are mapped into leading tests and explicit assignments to the control variable, Fortran DO loops are mapped into following tests, and CASE statements are mapped into a series of tests (instead of a single multiple-exit

test). An eighth kind of node, the LOOPHEAD node, will be discussed a little later.

We shall assume that, in forming the flow graph, any necessary variable renaming has been done so that all names are unique and we do not have to deal with scope rules. Blocks and scope rules would have to be handled in a more complicated way if the system described here were to be redesigned to analyze recursive procedures. For our purposes, input/output statements could be modeled in the flow graph with assignments to/from the variables read or written. Complicated input/output semantics can be modeled with assignments to auxiliary variables representing, for example, device position.

The nodes in our flow graphs are connected by directed arcs. Test nodes have two arcs leaving them (exit arcs); HALT and RETURN nodes have no exit arcs; all other nodes have one exit arc. START and PROCEDURE nodes have no entry arcs; all other nodes have one or more entry arcs.

A complete flow graph for a program and its sub-procedures consists of a set of disjoint graphs, one for each procedure or main program. The graph for the main program contains exactly one START and one HALT node; the graphs for the sub-procedures each contain one PROCEDURE and one RETURN node. The limitation to a single RETURN node is somewhat arbitrary, but allows us to describe one set of exit conditions for a procedure, instead of describing a different set of conditions for each RETURN.

We accept general flow graphs of the type described above as input; but to find, analyze, and eventually prove the termination of the loops in a program, we need to modify the input flow graph to

put it in a more constrained form. The operations described below are to be performed on each of the disjoint graphs, representing one procedure each.

First, we perform interval analysis with node splitting [Allen 1970] [Allen and Cocke 1972], [Cocke and Schwartz 1970], which forces each loop in the graph to have exactly one entry node, so that we can analyze the manipulations within a loop in terms of unique initial entry conditions. A graph with multiple-entry loops, such as the one in Figure 1.1 is changed into a reducible graph by node splitting, which makes copies of some of the nodes of a graph so that the new graph has fewer multiple-entry loops. Node splitting would change the graph in Figure 1.1 to that in Figure 1.2.

Arcs which go from a node within an interval to the interval head node are called latchback arcs; they represent branches back to the beginning of a loop. In any interval which has latchback arcs and whose interval head is not already a loophead node, we now replace the interval head node, A , with a pair of nodes: a LOOPHEAD node and A . We reroute A's original entry arcs to the LOOPHEAD node, add an arc from the LOOPHEAD to A , and leave all of A's exit arcs intact, as in Figure 1.3. The LOOPHEAD node serves to identify the top (beginning) of a loop and provides us a convenient place to attach loop termination assertions.

In analyzing a loop, we are interested both in its branches back to the top of the loop (its latchback arcs) and in its loop exit arcs, which cannot lead back to the LOOPHEAD node (without going through the LOOPHEAD node of a containing loop). We are interested in the loop

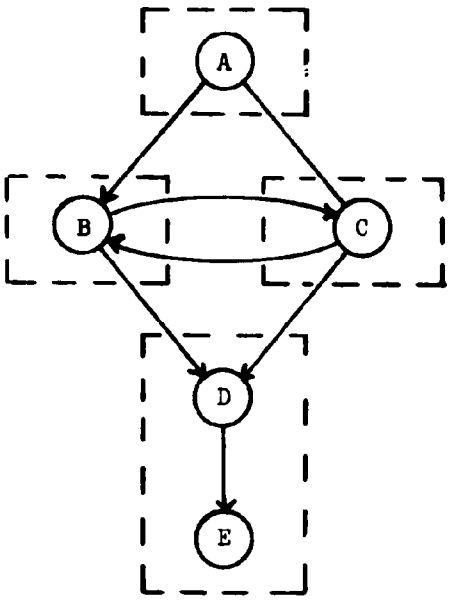

Figure 1.1. An irreducible graph, with rectangles showing its partition into intervals. The loop BC has multiple entry nodes, making its analysis difficult.

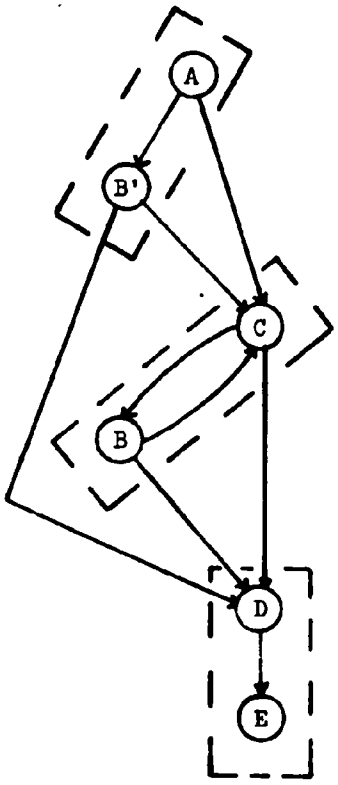

Figure 1.2. Node split version of the graph in Figure 1.1, in which the loop BC now has a single entry node, C .

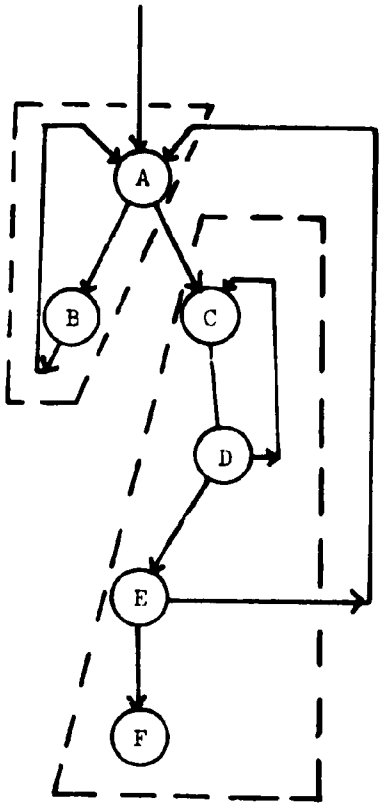

Figure 1.3a. A flow graph with its two intervals indicated by dashed lines.

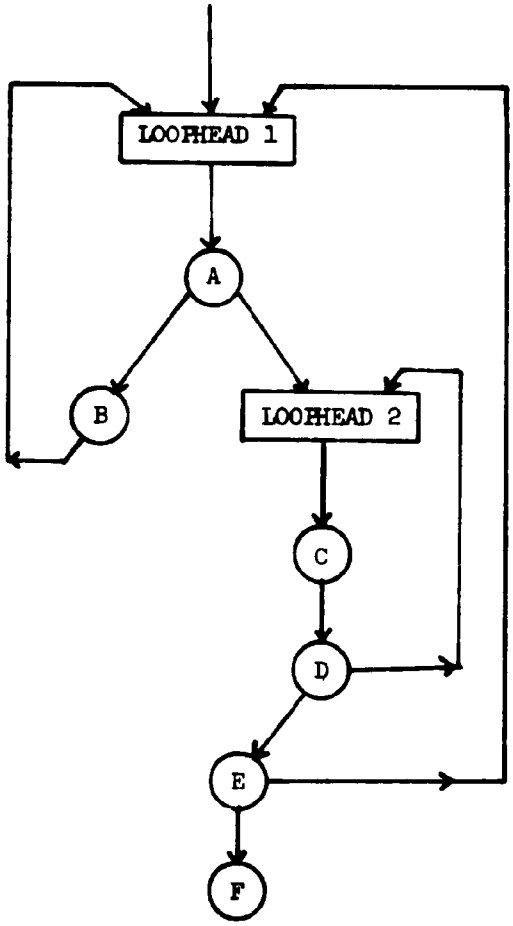

Figure 1.3b. The same flow graph after inserting LOOPHEAD nodes.

exit arcs because one way of proving that the loop terminates is to prove that an exit arc must eventually be taken as the program executes.

We may find that two or more loops in a program have a common beginning node and interval analysis indicated only a single loop, as in Figure 1.4a. To detect and clear up this situation, we in general need to modify each loop so that every path around the loop goes through an exit test (a test node which has a loop exit arc leaving it). We make a separate, contained, loop out of any paths which do not exit directly, as in Figure 1.4b. More formally, if breaking one arc leaving a TEST node breaks the only path from that node which eventually latches back to the top of the loop, then the other arc leaving the TEST node is a loop exit arc and that TEST node is an exit test. [Also see Appendix A, Example 10.]

In analyzing the effects of loops (described in Chapter 4), we may find it convenient to permute the nodes inside each loop (Figure 1.5) so that all the exit tests are at the top of the loop, thus making it easier to consider the degenerate case of zero iterations. If a loop has multiple exit tests, this modification is not always possible, so the best we can do is permute the loop so that one of the exit tests is at the top. [See Appendix A, Examples 4, 7, and 10.]

One final step in the preliminary processing of the flow graphs is to order the nodes so that when we later examine them one at a time to gather information and prove assertions, all of the appropriate predecessor nodes will have been already examined. We use the following rules to order the nodes:

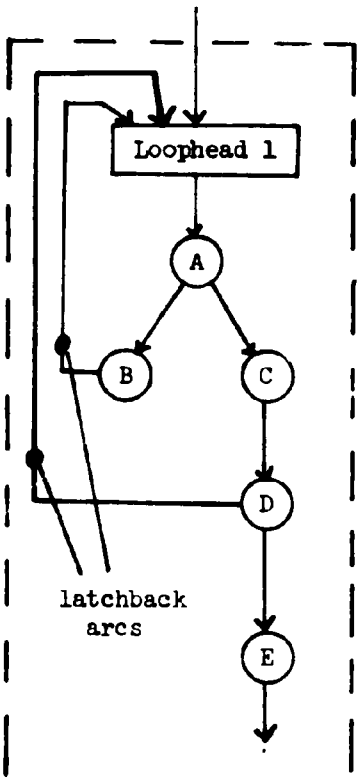

Figure 1.4a. A single loop as seen by interval analysis. The interval is indicated by dashed lines and may contain more nodes below E .

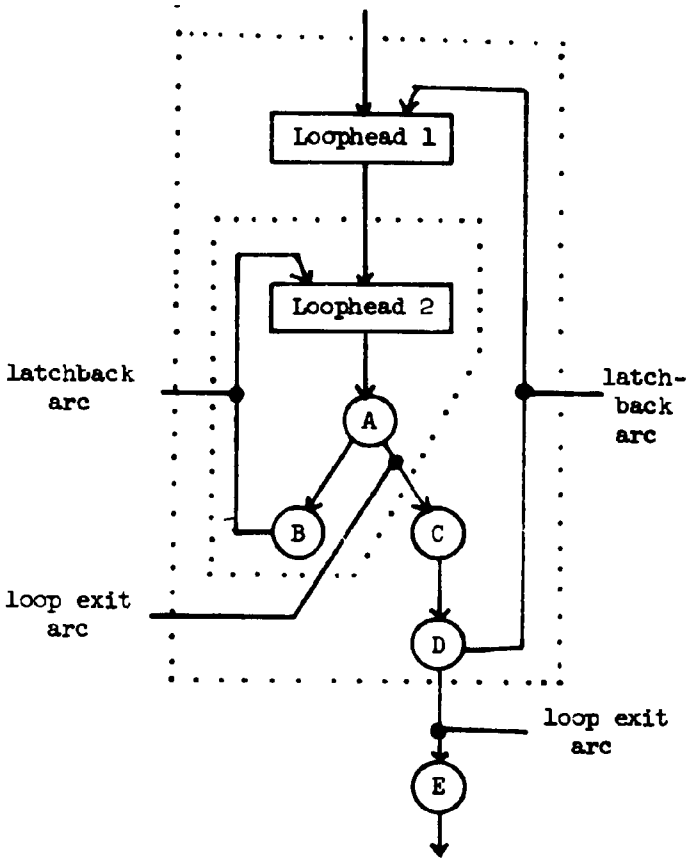

Figure 1.4b. The same graph after forcing each path around a loop to go through an exit test. The two termination issues of getting to node C and getting to node E are separated now into two different loops, indicated by their loophead nodes and by dotted lines. Note that, in contrast to intervals, node E and its successors are not in the loops. [See also Appendix A, Example 10.]

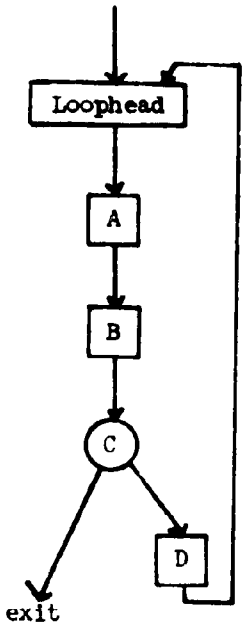

Figure 1.5a. A loop without leading exit tests.

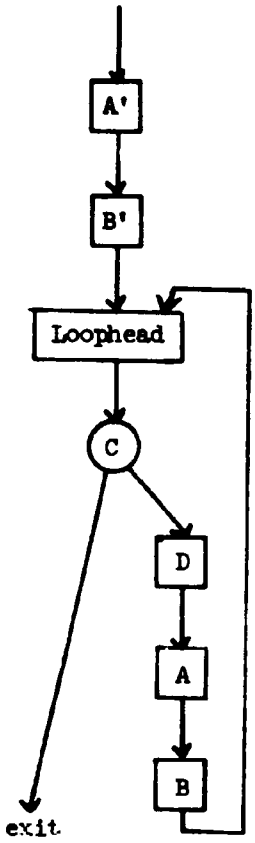

Figure 1.5b. The same loop permuted so that the exit test is at the top.

(1) Reduce each loop in the program to a single node.

(2) Topologically sort [Knuth 1973b, p. 258] the nodes in the reduced graph, using the directed arcs as the ordering.

(3) For each node in the reduced graph which represents a loop, topologically sort the nodes within the loop, ignoring all latchback arcs, then insert these nodes in the main topological ordering as a single group, so that all the nodes in the loop precede any nodes which followed the loop in the reduced ordering.

(4) Apply Step 3 until all loops have been expanded.

A discussion of this ordering and its properties appears in [Earnest et al. 1972].

Chapter 2. Generation of Semantic Error Assertions

This chapter discusses the generation of assertions which state that "no semantic error occurs if the following node is executed". This is a very local, operator-driven process. These assertions are attached to each of the entry arcs for the node, as in Floyd's original description of the inductive assertion technique [Floyd 1967]. Semantic errors occur whenever an operation gives an undefined result, as specified in the language definition or in a set of implementation restrictions for a particular compiler/computer combination. The examples are given in terms of Algol 60 programs running on a machine which gives undefined results for underflow/overflow, assignment or any other use of uninitialized values, subscripts out of range, etc. The machine is also assumed to perform mathematically correct comparisons of, say, i and j even when j-i would overflow/underflow. Machines (such as the CDC 6600) which violate this last assumption are discussed below, and in [Sites 1974].

Assertion generation for value parameters is straightforward, but name parameters are handled strictly according to the copy rule, making a separate copy of a procedure for each call.

The symbols I_{min} and I_{max} are introduced as notations for the smallest and largest representable integers on the target machine. The symbol ω is introduced to denote the undefined value. ⊗

After forming a modified flow graph, as described in Chapter 1, we attach to its arcs various assertions stating that the operations in each node are well-defined. For each node in the flow graph, we

mechanically form a set of assertions describing restrictions on the program variables which must be true upon entry to the node in order for each operation in the node to produce well-defined results. We then attach this set of assertions to each of the entry arcs for that node.

In most of the examples which follow, we shall assume that programs are written in Algol 60 and are run on a compiler/computer system which has the following implementation restrictions.

1. No real numbers.
2. Integer overflow. The binary operations $i+j$, $i-j$, $i \times j$, and $i \div j$, give the mathematically correct result if and only if i and j have defined values and the result is in the range I_{min} to I_{max} inclusive; otherwise the result is undefined. Division by zero produces a result outside of the range I_{min} to I_{max}. It is assumed that $I_{min} < 0$ and $I_{max} > 0$. As an example, for the PDP-8 with 12-bit 2's complement integer arithmetic, $I_{min} = -2048$, $I_{max} = +2047$.

   A program can be analyzed using only symbolic values for I_{min} and I_{max}, in which case we may be able to state maximum and minimum values for them, respectively, drawn from the values of the smallest and largest integer constants in the program. Alternately, a program can be analyzed with only loose bounds on I_{min} and I_{max}, such as $I_{min} \leq -1000$, $I_{max} \geq 1000$. This will save some work in checking that the small integer constants often encountered in programs are within the representable range. Alternately, the exact values of I_{min} and I_{max} for some

particular machine can be supplied, in order to answer the question, "Will this program generate any overflows when run on this particular machine?" Most of the examples below assume $I_{min} \leq -1000$ and $I_{max} \geq 1000$.

3. Representable constants. All integer constants must be in the range I_{min} to I_{max} inclusive.

4. No use of uninitialized variables, including simple assignments. No right-hand-side expression is allowed to use an uninitialized variable. In particular, the operation $i := j$ will assign the value of j to i if and only if j has a defined value; otherwise a semantic error occurs. It is possible to write programs which violate this restriction and still give meaningful results, but more often a violation of this condition indicates an error which is best caught as soon as possible.

Algol 60 semantics for local variables starting out undefined at the beginning of a block are modelled by putting into the flow graph, at the start of each block, special assignments of the undefined value, ω , to each local variable. The program proper is not allowed to use ω .

5. Mathematically correct comparison. The relations $i < j$, $i \leq j$, $i > j$, $i \geq j$, $i \neq j$, $i = j$ produce the proper value <u>true</u> or <u>false</u>, even in cases where $j-i$ would produce an overflow. For a machine which does not have this property, such as the CDC 6600, programs must be transformed so that every comparison is done as a subtraction and a sign test. All such subtractions will then be checked for overflow in the normal way. Two

representations of zero are allowed if the implementation gives identical results for each.

These restrictions are in addition to those specified in the Algol 60 Report [Naur 1963], such as requiring each subscript to be within the declared bounds of an array.

The examples presented here do not directly address the issues of a program executing in a given amount of memory or a given amount of time. The only guarantees about space and time are that both requirements are finite: the memory required is finite because no recursion is allowed, and because the bounds for individual arrays are limited by I_{min} and I_{max} ; the time required is finite if all loops are proved to terminate.

Typical assertions generated are:

Node	Assertion generated
A[i] := j+k	$j \neq \omega \wedge k \neq \omega \wedge I_{min} \leq j+k \leq I_{max} \wedge$ $i \neq \omega \wedge A_\ell \leq i \leq A_u$. (A_ℓ and A_u are the lower and upper bounds for the array A .)
i < j+5	$i \neq \omega \wedge j \neq \omega \wedge I_{min} \leq 5 \leq I_{max} \wedge I_{min} \leq j+5 \leq I_{max}$. (Since I_{min} is assumed to be < 0 , the condition $I_{min} \leq 5$ is clearly true.)
i := j	$j \neq \omega$.
A[i] := A[i]+1	$i \neq \omega \wedge A_\ell \leq i \leq A_u \wedge A[i] \neq \omega \wedge I_{min} \leq 1 \leq I_{max} \wedge I_{min} \leq A[i]+1 \leq I_{max} \wedge i \neq \omega \wedge A_\ell \leq i \leq A_u$. (The last two terms come from the left-hand A[i] .)

Chapter 2. Semantic Errors

Standard techniques can be used to simplify the assertions, including removing terms which are clearly true, removing duplicate terms, and removing terms which are implied by other terms ($i < 5 \wedge i < 8 \wedge i \neq 12$ reduces to $i < 5$). One way to remove redundant terms mechanically from a set T of n terms is to eliminate any term for which the theorem

$$\{T - t_i\} \supset t_i$$

is true. ($\{T - t_i\}$ represents the set of all terms except t_i.)

In most cases, the generation of semantic error assertions is quite straightforward, but some complications arise in handling procedure calls. Arguments passed to value parameters are treated like the right-hand side of an assignment statement at the point of call, i.e., the argument expression must be well-defined when evaluated before the call. In contrast, procedures with name parameters must be handled strictly according to the copy rule, making a unique copy of the procedure for each call and logically substituting the body of the procedure for the CALL node. This use of the copy rule is one way to reflect properly the side effects which can result from tricky use of name parameters, but is also a reason that we do not handle recursion.

Procedures with array arguments have the problem that the procedure does not specify the legal lower and upper bounds for subscripts. Either of two strategies can be adopted for generating and proving assertions about subscripts for such arrays: symbolic names like A_ℓ and A_u can be used in all the assertions, and the proof techniques can try to push back to the entry point of the procedure any assertions (restrictions) which must be true on entry

in order to avoid subscript range errors; alternately, the programmer can supply an extra statement to the proof system, describing the bounds for each such array. If the programmer has definite assumptions about array bounds in his mind, it is better to state them to the proof system. Not doing so forces the system to try to synthesize equivalent information, a much harder process.

Chapter 3. Generation of Loop Termination Assertions

This chapter describes the generation of assertions which are true if and only if the loops in a program terminate after a finite number of iterations. For many practical cases, the assertions generated lend themselves to direct proof. For loops which have obscure reasons for termination, the assertions have equally obscure reasons for being true (of course, in general, proving loop termination is theoretically unsolvable; we shall not solve the halting problem here). For many loops which do not terminate, the corresponding assertions can be proven definitely false and the user alerted to the bug, perhaps with a counterexample.

The basic form of the assertions generated is, "There exists a k such that on the k-th iteration of the loop, one of the exit arcs will be taken." For many loops involving monotonic expressions in their exit tests, or simple searches, or movement through a linked list, these assertions are easy to prove. ⊗

Loop termination assertions are harder to generate than semantic error assertions because the goal is much more abstract. For semantic errors, the assertions generated are a straightforward function of the source language definition and compiler/computer implementation restrictions. For loop termination, however, synthesizing an appropriate assertion may well be harder than proving it true.

Generation of termination assertions can be "driven" by a variety of goals. One technique is to insert a counter in each loop and assert that the count is bounded; however, such a statement doesn't lend itself to direct proof -- having a counter doesn't give any insight into its behavior. Another technique is to require all loops to

be FOR loops or DO loops in which the step and limit are evaluated exactly once and the iteration variable cannot be changed inside the loop; such loops terminate by definition (if a zero step is prevented).

In between these extremes, we need to find a strategy for generating assertions which are related to the intended reasons for loop termination that the programmer had in his mind when he wrote the loop. Without searching for these reasons, we will have a hard time mechanically proving the termination of subtle loops whose termination properties may be perfectly clear to a human. In unannotated programs, the best evidence we have for the intended termination of loops is in their exit tests. For a given loop to terminate, one of its exit tests eventually must be satisfied (i.e., branch to a loop exit arc). Often the tests themselves present the reason for loop termination, while sometimes the preceding logic (which sets the values of the variable(s) in the test) embodies the reason for termination.

For example, in a loop such as:

```
while ℓ < r do
    if p(ℓ) then ℓ := ℓ+1
           else r := r-1
```

where p is an unspecified predicate, the exit test $\ell < r$ provides us with the proper driving goal: prove $r-\ell$ is monotonically decreasing. If we try to prove that the loop terminates because either ℓ or r is monotonic, we will fail; the relevant monotonic expression involves both r and ℓ and appears only in the exit test.

As a second example, consider the loop:

Chapter 3. Loop Termination

```
comment this program is a subset of an example in [Ashcroft and
        Manna 1972];
t := true;
while t do
     begin
     if q(x) then
          begin
          x := b(x);
          if s(x) then
               x := c(x)
          else begin
               x := f(x);
               t := false
               end
          end
     else begin
          x := g(x);
          t := false
          end
     end.
```

Here, the exit test of t offers no direct enlightenment, but as we shall see in Chapter 4, the flow graph for this loop will be mechanically modifed by test elision so that the manipulations of t are ignored, the assignments t := false are immediately followed by branches out of the loop, and the assignment x := c(x) is immediately followed by a branch to the test if q(x) ... as in this modified program:

```
loop:  if q(x) then
           begin
           x := b(x);
           if s(x) then
                begin
                x := c(x);
                goto loop
                end
           else begin
                x := f(x);
                goto exit
                end
           end
       else begin
           x := g(x);
           goto exit
           end
exit:  ...
```

Thus q(x) and s(b(x)) become exit tests, and we are now more directly confronting the reasons for the loop's termination.

For loops with leading tests, such as those we tried to form by the manipulations described in Chapter 1, it is straightforward to generate an assertion that there exists a $k \geq 1$ such that on the k-th iteration of the loop, an exit arc will be taken. For the original while t ... loop above, the assertion would be:

$$\exists k \geq 1 \text{ s.t. } \sim t_k$$

where the subscript k indicates "the value of the variable at the beginning of the k-th iteration," i.e., the value of a variable at the LOOPHEAD node, before any nodes inside the loop have been executed the k-th time. The termination assertion for the modified loop above would be:

Chapter 3. Loop Termination

$$\exists k \geq 1 \quad \text{s.t.} \quad \sim q(x_k) \vee [q(x_k) \wedge \sim s(b(x_k))] \quad .$$

Note that we describe the exit test $s(x)$ in terms of x_k, the value of x at the top of the loop, as modified by the assignment $x := b(x)$. In general, a multiple-exit loop may have exit tests which are preceded by enough computation that the values of the variables in the exit test cannot be described in terms of the values at the top of the loop. In this case, we will have to abandon the top-of-the-loop bindings and state an assertion like:

$$\exists k \geq 1 \quad \text{s.t.} \quad \sim q(x_k) \vee \sim s(x'_k)$$

where the primed x signifies the value of x upon entry to the test node $s(x)$, in the middle of the k-th iteration of the loop. All we are really doing is delaying the analysis of the behavior of x'_k until we actually try to prove the assertion true. This is appropriate, since we may find that the stronger theorem

$$\exists k \geq 1 \quad \text{s.t.} \quad \sim q(x_k)$$

is true, or we may find that the flow graph for the loop (and hence the termination assertion) is completely changed during the information-gathering and proving process described in Chapter 4. [A simple multiple-exit loop is in Appendix A, Example 4.]

While assertions such as those above can be mechanically generated from any loop, it is in general an unsolvable problem to prove that the assertions are true. However, a small variety of techniques based on monotonic expressions, finite sets, and searches can prove the termination of most loops encountered in practical programs. Also, this strategy of generating a $\exists k \ldots$ assertion

sometimes allows a proof system to state that a loop definitely never terminates. If the final i := i+1 were left out of the loop:

```
while A[i] > 0 do
    begin
    ...
    i := i+1
    end
```

and no other statements inside the loop changed the value of i or A[i], then the loop termination assertion,

$$\exists k \geq 1 \text{ s.t. } A_k[i_k] \leq 0$$

could be shown to be invariant over k, and the quantifier dropped:

$$A[i] \leq 0 \ .$$

If this assertion is true, the loop terminates immediately; if it is false, the loop never terminates.

The next chapter discusses proofs of the mechanically generated semantic error and loop termination assertions. You may want to review Appendix A, Example 1, at this point.

Chapter 4. Proofs

This chapter is the heart of the thesis; it describes an algorithm for examining the nodes of a flow graph in forward topological order (detailed in Chapter 1), and at each node (1) trying to prove all its entry assertions, (2) performing extra processing for LOOPHEAD and TEST nodes, and (3) developing the given information for its exit arcs (to be used in subsequent proofs). In trying to prove an assertion, there can be five answers: a) true; b) false; c) maybe, but more information will be known later; d) maybe, but a refinement of the given information is available; e) or just plain maybe. In the last case, the user will have to decide if the program contains a bug or if the proof system just isn't powerful enough.

When a LOOPHEAD node is encountered, a first pass is made through all of the nodes in the loop gathering recurrence relations about how the values of the variables at the beginning of the k+1st iteration are related to values at the beginning of the k-th iteration. Then an induction routine tries to describe the set of values each variable takes on during all iterations. Finally, a second pass is made through the loop, proving assertions and processing nodes in the normal way. When a TEST node is encountered, an attempt is made to elide the test: to prove that along some entry path(s), the test is either always true or always false. If such a path is found, then it is separated from other paths (perhaps causing node splitting), and re-routed around the test. The topology of the flow graph is then re-analyzed. This sometimes has the effect of mechanically synthesizing an appropriate lexicographic ordering on a pair of variables, when a single loop is changed into a pair of nested loops.

Chapter 4.1. Proofs

After the entry assertions and the node itself have been processed, the new given information for the exit arcs is synthesized. This synthesis involves merging the entry given information, the entry assertions, and the results of tests, then modifying this information to reflect any assignments inside the node.

There is no backtracking in the node processing algorithm, but some nodes are visited more than once: a) Since two passes are made through each loop, a node inside a nest of n loops will be visited 2^n times, although only n+1 of these visits will do any work. b) If a test is elided, the graph is re-analyzed and re-processed from the beginning of the outermost loop containing that test.

The notation " $\rightarrow\varepsilon$ " is introduced to specify an initial subset of an ordered set. [See Appendix B for a summary of the procedure for processing nodes.]

⊗

Given a modified flow graph with assertions attached, as described in Chapters 1-3, we will now process the nodes one at a time, proving assertions and developing information for the proofs at later nodes. Starting with the graph for the outermost procedure, we examine each node in topological order, performing the following operations on it as we go.

1. Prove Assertions

First, we try to prove the assertions on the entry arc(s). If the node is a LOOPHEAD node, we temporarily ignore the assertions on the latchback arc(s), and just treat the initial entry arc(s). Each arc has attached to it two sets of information: the given information developed on exit from the predecessor node, and the assertions to be proved. (The given information for START and PROCEDURE nodes is null.)

Chapter 4.1. Proofs

We simply call a theorem prover for each assertion on an arc, asking it to prove

given $\supset$ assertions .

The possible answers true, false, and maybe are explained in detail below:

a) If the answer is true, then we mark the assertion true and never bother proving it again.

b) If the answer is false, then the program contains a definite error. At this point, we can state to the user that the assertion was false and go on, but we can often be more helpful than that. First, the theorem prover may have supplied a counterexample, a set of values for program variables which make the assertion false. In this case, we tell the user the counterexample. Second, a false assertion may be an indication of an error much earlier in the program, so it would be helpful (but entirely optional) for us to "push back" the assertion as far as possible toward the start of the program. In moving such false assertions toward the start of the program, we may find related assertions moved to a common point from many different nodes of the program. In this case, we can give a single error message, instead of "discovering" the same bug in, say, three different places. To the extent that this merging of related or identical false assertions is successful, we also guide the user to the most appropriate place in the program to fix the error. If an assertion is false on the very first iteration of a loop, then we may be able to move it outside of the loop entirely, thus directly indicating an error in loop initialization, not (necessarily) in the inductive properties of the loop. [See Appendix A, Examples 1 and 2.] Any false assertions which are pushed

all the way back to a START or PROCEDURE node represent entry restrictions for the whole routine, and should be both documented and explicitly tested. Thus, although this movement of false assertions is not logically required, it enables our system to encourage a programming style which includes explicit, executable tests for all entry conditions, perhaps coupled with the printing of a user's error message and the returning of an "undefined over the given inputs" value for the result of a function. Alternately, we may encourage a style which extends the meaning of a procedure to include all possible inputs, thus removing the restrictions. In either case, the user is encouraged to make his program more reliable without his engaging in a tedious and often incomplete analysis of degenerate cases.

c) If the answer from the theorem prover is maybe, but we are on the first, information-gathering pass around a loop (using dummy bindings of variables), then we simply reserve judgment until the second pass. It would be possible to attempt no theorem proving at all during the first pass through a loop, but that has the general effect of delaying the discovery of information and lemmas which are useful in analyzing the loop. So, as a somewhat arbitrary choice, we try proving all assertions on the first pass through a loop, dropping those for which we are successful, and trying again on the second pass for the others.

d) If the answer from the theorem prover is maybe, but the given information has come from a merging of several different paths and is marked "a possibly useful refinement of this information is available", then we can break the proof down into several cases, for different paths leading to the node being processed. A "refinement" mark is

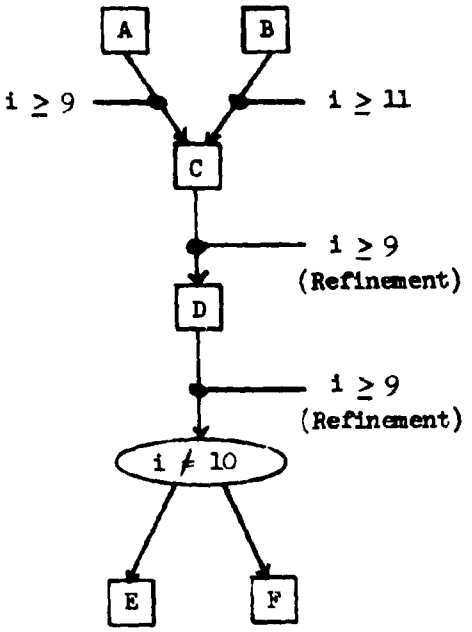

Figure 4.1a. At node C, the two relations about i are merged by taking the one implied by both, the weaker: $(i \geq 11) \supset (i \geq 9)$ and $(i \geq 9) \supset (i \geq 9)$.

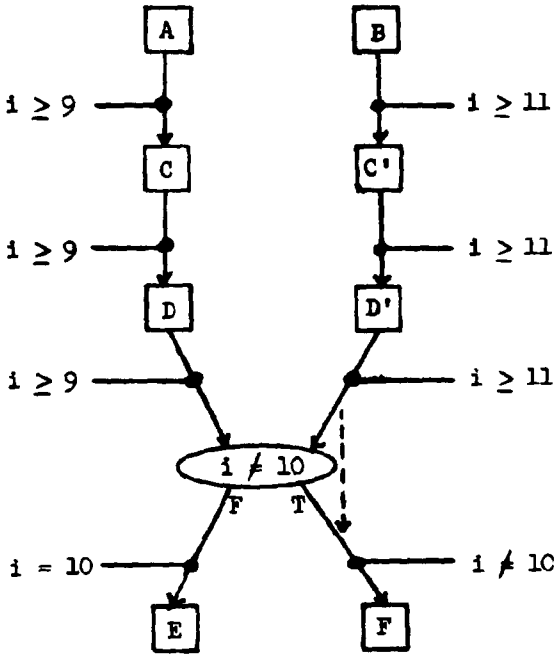

Figure 4.1b. Because the refinement is useful at the test, nodes C and D have been split to separate the two paths. The dotted arrow indicates the subsequent elision of the test.

Figure 4.1. Example of node splitting to separate paths associated with a useful refinement of <u>given</u> information.

created when two arcs in the flow graph merge and they contain different given information, as described in detail later in this chapter. If a refinement of the given information exists, and we can prove the assertion in question conclusively true or false for some of the cases in the refinement, then we make separate paths for those cases in the flow graph, possibly making copies of some nodes, as shown in Figure 4.1. [Also see Appendix A, Examples 8 and 10.]

e) If the answer from the theorem prover is maybe, then either the program contains an error or our proof system isn't good enough to discover that the theorem is in fact true.

We have covered the five cases involved in proving assertions on entry arcs. We now look at the processing of the node itself.

2a. LOOPHEAD Nodes

If we are examining a LOOPHEAD node, then we have just reached the beginning of a loop. To prove the various assertions inside the loop, we need to synthesize the ranges of possible values that all variables can take on in the body of the loop. Essentially, if we can describe the complete set of values that a variable takes on at the loophead node, be it the first iteration or the k-th, then we are in a good position to prove all of the assertions inside the loop which depend on this set of values.

Our method for discovering the ranges of variables in a loop is to take one pass through the nodes in the loop symbolically developing the value of each variable after one iteration of the loop in terms of the value of all variables at the beginning of that iteration. For

example, starting with the symbolic bindings (2) in Figure 4.2, one pass through the loop body gives the following recurrence relations:

$$
\begin{aligned}
(3) \qquad & i_k \leq 100 \ \wedge \\
& i_{k+1} = i_k + 11 \ \wedge \\
& j_{k+1} = j_k + 1 \ \wedge \\
& n_{k+1} = n_k \ .
\end{aligned}
$$

We then feed these induction relations and the set of initial entry relations (1) to an induction routine, which synthesizes the complete set of values that each variable takes on at the loophead node. The synthesized sets of values for i , j , and n at the LOOPHEAD node would then be:

$$
\begin{aligned}
(4) \qquad & i_0 \neq \omega \ \wedge \ i_0 \leq i \ \wedge \\
& 0 \leq j \ \wedge \\
& n = i_0 - 1 \ ,
\end{aligned}
$$

where i_0 represents the value of i at the READ statement. Note that it is wrong to deduce that

$$i \leq 111$$

at the LOOPHEAD node. This is only true after going around the loop one or more times, but is not true on the first iteration if the value read in for i is, say, 347 . As discussed later in this chapter, the relations for i and j would actually be marked "a refinement exists", so that the two cases of first iteration and subsequent ones could be distinguished later if necessary. The details of the loop induction routine will be discussed later in this chapter.

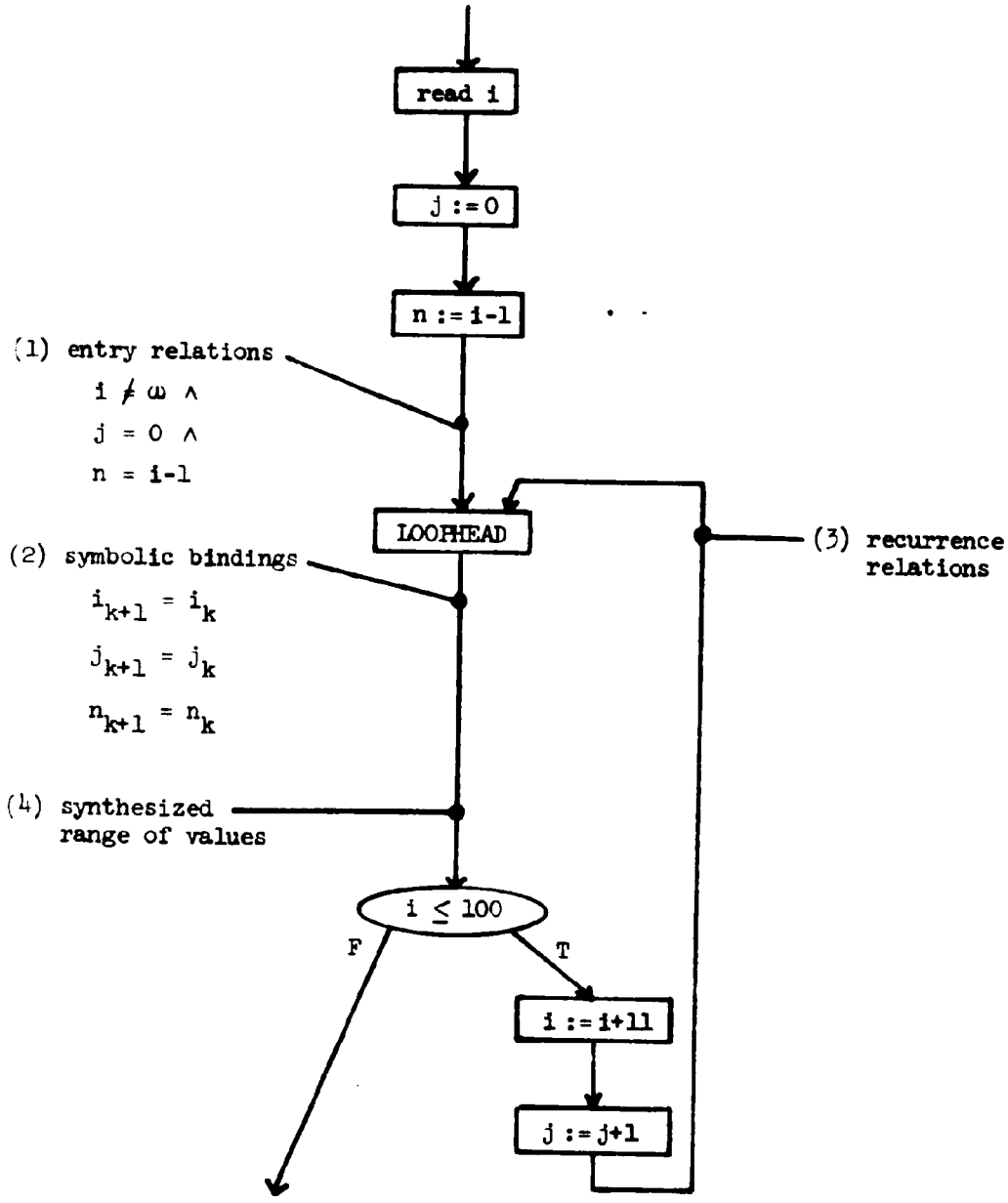

Figure 4.2. Sample loop for showing loop induction information.

After the initial pass around the loop and call of the induction routine, we attach the synthesized relationships and set of variable values to the exit arc of the LOOPHEAD node as given information for subsequent nodes. We than take a second pass around the loop, processing nodes and proving assertions in the normal way, proving the assertions on the latchback arcs just before processing nodes which topologically follow the loop. [For examples of loop processing, see Appendix A, Examples 1, 7, 9, and 10.]

2b. TEST Nodes

If the node we are examining is a TEST node, then we try to elide the test. We check to see if the test is always true or always false along some incoming path by making assertions out of the test and its negation and trying to prove these assertions. Our normal refinement and path-separating mechanism described above will then separate out any incoming path for which the test can be elided. If so, we re-route that path to the appropriate true or false exit node. This re-routing may change the structure of the loops in the program, either creating new loops [example below and Appendix A, Example 8] or destroying an existing loop [Appendix A, Example 10], so we must re-analyze the structure of the program, as described in Chapter 1. Actually, we only need to re-analyze starting with the outermost loop containing the re-routed arc. After the re-structuring, we start over at that outermost loop, gathering information and proving assertions. This elision of redundant tests is an important tool for separating loop-termination issues. For example, in the program:

```
while p ≠ Λ do
    if info(q) > info (p) then q := link(q)
                          else p := link(p);
```

sometimes we make progress toward the end of the list p , and sometimes we don't. We can see in the flow diagram, Figure 4.3, that after setting q := link(q) , the test p ≠ Λ is always true since p is unchanged, so we can elide it, giving the program:

```
while p ≠ Λ do
    begin
    while info(q) > info(p) do
            q := link(q);
    p := link(p)
    end;
```

In this modified program, the two loop termination issues are separated: it is now fairly easy to prove that the outer loop terminates (if the inner loop does and assuming that we have an appropriate model of single-linked lists), and the inner loop may or may not terminate, depending on what else we know about q , info(q) , and info(p) . In some sense, the effect of our creating two nested loops is to synthesize an appropriate lexicographic ordering on (p,q) pairs.

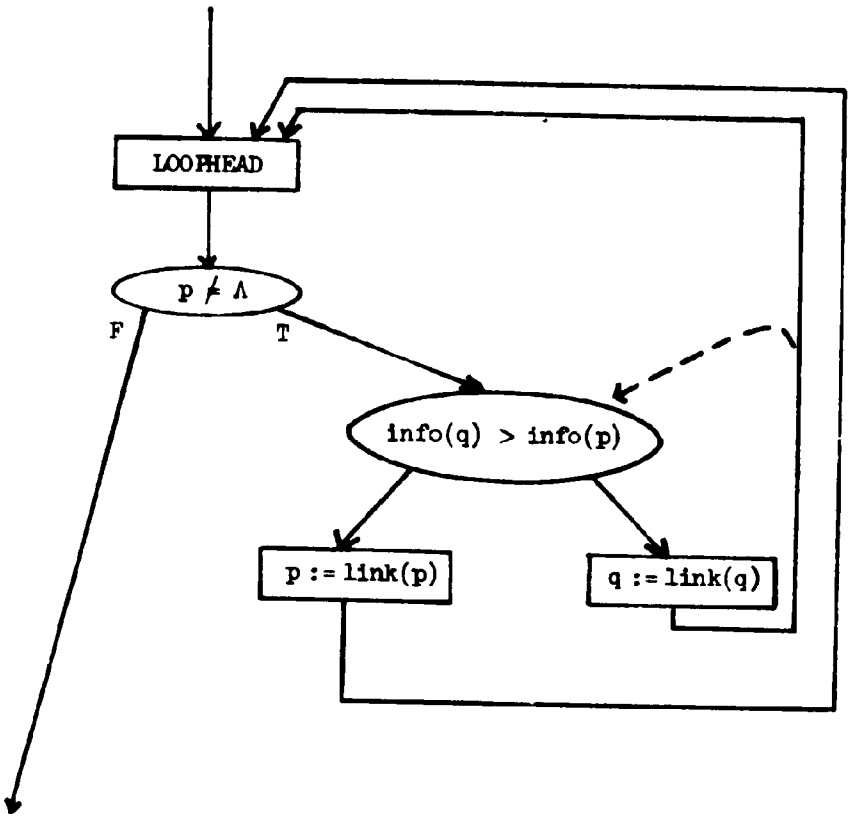

Figure 4.3. An example of eliding a test and thus changing one loop to two nested loops, separating the termination issues, and synthesizing a lexicographic ordering on p and q .

3. Develop Given Information for Exit Arcs

Before leaving a node and going on to process the next one, we must attach the appropriate given information to all of the node's exit arcs. We synthesize this by merging the given information from the node's entry arcs, adding the assertions on those arcs (the assertions must be true or the program will terminate uncleanly at that node and never traverse the exit arcs), and modifying everything to reflect any assignments within the node. Also, if the node is a TEST node, we add the tested condition and its negation to the true and false exit arcs respectively.

Simple as the preceding paragraph may sound, there are some very complicated issues involved in this step. The first complication arises when a node has multiple entry arcs with different given information, as in Figure 4.4. We could simply use the disjunction of the two cases for the exit arc:

$$(i \geq 10 \wedge m = 1) \vee (i \geq 11 \wedge m = 0)$$

but this has the drawback that all proofs based on this information will have to consider two separate cases. Since we would be creating multiple cases whenever two or more arcs merge in the program, we can be faced with 2^n cases after n merges, as in Figure 4.5, where the given information on the exit arc for C includes

$$[(A \wedge i = 2 \wedge m = 4) \vee (\sim A \wedge i = 1 \wedge m = 1)] \wedge$$
$$[(B \wedge i' = 3 \wedge m' = 9 \wedge j = 5) \vee (\sim B \wedge j = 7 \wedge i' = i \wedge m' = m)] .$$

This is an unwieldy premise for proving a later assertion like

$$i' \geq 0 ,$$

where the only relevant information is that

$$i' = 1 , 2 , \text{or } 3 .$$

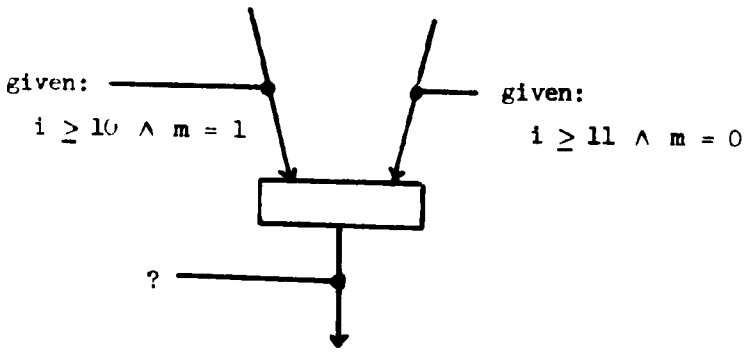

Figure 4.4. Merging of different given information.

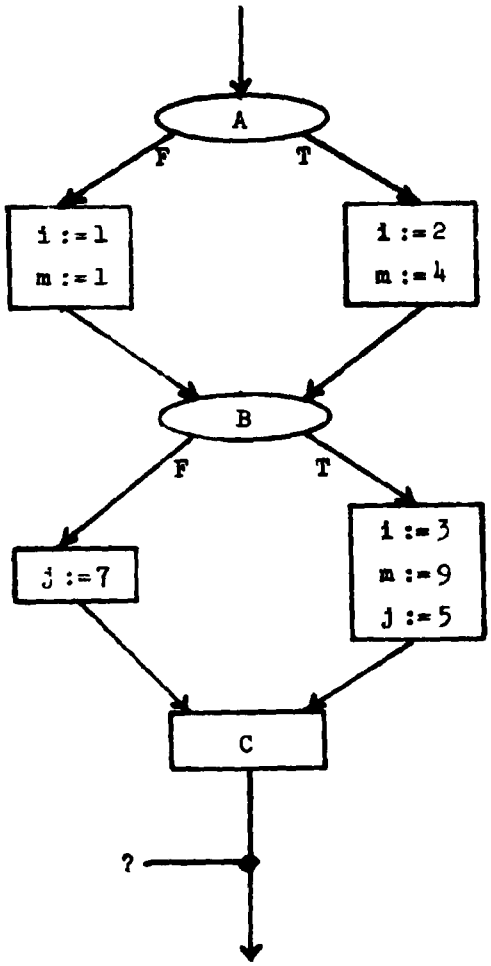

Figure 4.5. Cascaded merges can result in as many as four different cases to consider at node C .

Chapter 4.3. Proofs

Often, it is unnecessary to keep track of the interplay between i , j , m , the test in node A , and the test in node B ; it can be sufficient just to remember

$$(1 \leq i \leq 3) \wedge (m = 1, 4, \text{or } 9) \wedge (j = 5 \text{ or } 7)$$

on exit from node C .

We would like to avoid disjuncts as much as possible, mimicking the human trait of finding useful lemmas which cover all cases simultaneously. Yet we also don't want to lose any inter-relationships (such as $m = i^2$) if they are in fact required in a later proof. One strategy is to record a set of weak relations which are true on all arcs being merged, and to mark that a refinement of these relations exist, i.e., that by going back to the point of merging, the exact disjunction can be formed if necessary. We try to use the weak, disjunct-free relations to prove subsequent assertions, and only if the stronger, more exact information is needed do we construct it. In the example of Figure 4.4, we would synthesize

$$[(i \geq 10) \wedge (0 \leq m \leq 1)] \quad \text{(Refinement)}$$

as the <u>given</u> information for the exit arc. In the example of Figure 4.5 we would attach

$$[(1 \leq i \leq 3) \wedge (m = 1, 4, \text{ or } 9) \wedge (j = 5 \text{ or } 7)] \quad \text{(Refinement)}$$

to the exit arc at C .

We can develop weak relations from the following rules:

(1) Assume that information from n arcs numbered 1 to n is being merged, that the information on each arc is a set of conjuncts, $C_i = \{R_{1i} \wedge R_{2i} \wedge R_{3i} \ldots \wedge R_{ki}\}$ and that we want to produce a set of conjuncts.

(2) For each conjunct A on each of the n arcs, add A to the result set if that clause is implied by the information on each arc, i.e., if

$$\forall\, 1 \leq i \leq n \quad C_i \supset A \ .$$

(3) Form disjuncts of clauses which occur on different arcs, but which contain the same set of variables (like $j \leq k \vee j > k+2$), but avoid forming cross-product disjuncts of clauses with no variables in common (like $i = 1 \vee n = 4$).

(4) As a special case of (3), change expressions like $(n = i \vee n = i+1 \vee n = i+2)$ to $(i \leq n \leq i+2)$. Change expressions like $(i = n \vee i > n)$ to $(i \geq n)$.

The above rules are by no means "optimal", but they offer a useful heuristic for what information to keep around and what information to reconstruct only if needed to prove a particular assertion. [For use of these heuristics, see Appendix A, Examples 3 and 6.]

The crucial issue here is to make the system appropriately goal-driven: to develop high-payoff relations always, but generally not to synthesize any complex relations until the goal of proving a specific assertion demands the creation of those complex relations. Thus, we do not make all possible deductions from a set of relations (like deducing that $i < k$ from $i < j \wedge j < k$), but instead we wait until some goal or driving force makes a particular deduction relevant (for instance, we may have to prove the assertion $i \neq k$).

One of the beneficial side effects of rule (2) above is that it provides a driving force for discovering loop invariants: for each

relation on an initial entry arc of a LOOPHEAD node, we will try to prove that that relation is implied by the information on the latchback arcs. If the implication holds, then we have discovered a relation which is true on all iterations of the loop, as shown in Figure 4.6.

After merging information from multiple entry arcs, we add all of the assertions on the entry arcs to the given information we are building. We cannot cleanly exit the current node if an assertion is false, so all assertions will be true if we in fact reach an exit arc during an actual execution.

We have now formed a set of given information that needs to be transformed to reflect any assignments inside the node being processed, then attached to all the exit arcs. We will call the untransformed information G, and its transformation G'. For scalar assignments,

$$i := \text{expression},$$

the recording of the new equality

$$i = \text{expression}$$

in G' is straightforward. However, for assignments to aggregates, such as to elements of an array or fields of a node in a list structure, we have to investigate the possibility of aliases: assignments to $A[i]$ can change $A[j]$ if i can equal j. So if the node we are processing contains an assignment

$$A[i] := x,$$

we must look at G and for every subscript, s, of A in a relation R, we try to prove that either $s = i$ or $s \neq i$, based on the information in G. If $s = i$, then we reflect the assignment $A[s] := x$, and add

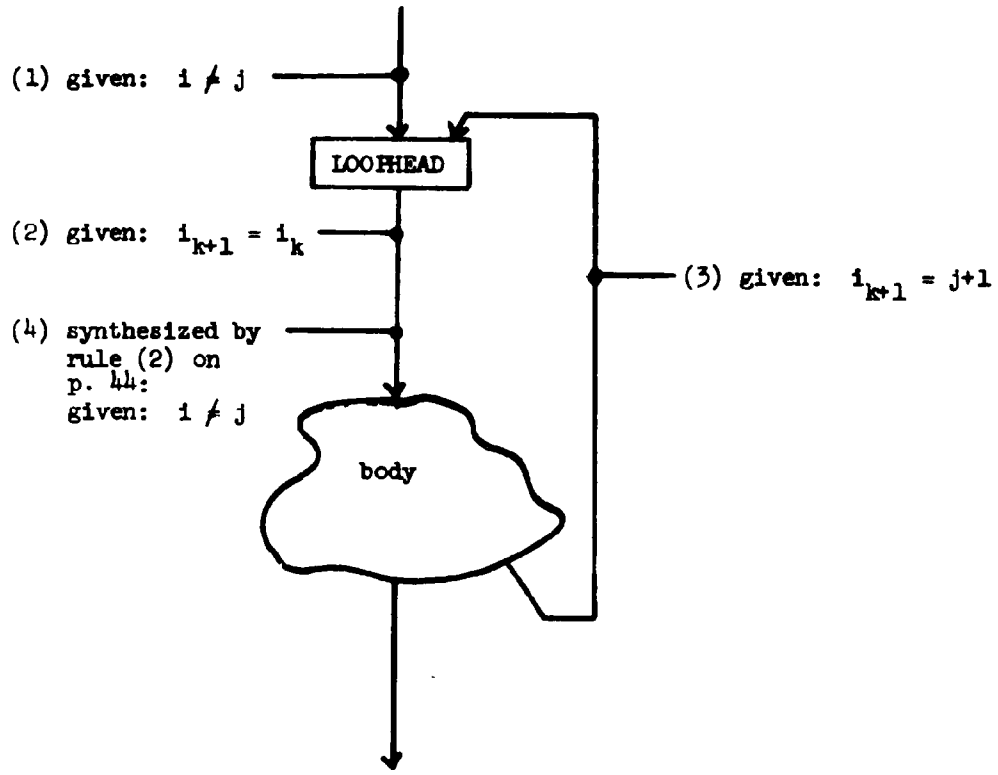

Figure 4.6. Synthesis of loop invariants. When we try to merge the information labeled (1) and (3), we find that $i_{k+1} = j+1 \supset i \neq j$, so by rule 2 on page 44, we add the relation $i \neq j$ to the set of information (4) which is true on all iterations of the loop.

the appropriate R' (which may be just the empty clause) to G' . If $s \neq i$, then the s-th element of the array is unchanged, so we copy R to G' unchanged. If we don't have enough information to prove any relationship between s and i , then we add to G' :

$$(s \neq i \wedge R) \vee (s = i \wedge R') \quad .$$

We can avoid this disjunct if $R \supset R'$ or $R' \supset R$, by adding only the weaker relation (R' or R respectively) to G' , marked of course "refinement exists". We could also use the weaker disjunct

$$R \vee R' \quad \text{(Refinement)}$$

ignoring the interaction between i , s , R and R' . Alternately, we could drop the relation involving A[s] entirely and add nothing to G' . [See Appendix A, Example 6.]

After transforming G to G' to reflect the assignments in the node, we attach G' to each of the node's exit arcs. If the node is a TEST node, we also add the test expression or its negation to the given information on the appropriate exit arc. This completes the processing at a node, so we can now move on to the next node.

Chapter 4.4. Proofs

4. Details of Loop Induction.

On the first, information-gathering pass around a loop, we try to generate a set of recurrence relations, expressing the value of each variable at the beginning of the k+1-st iteration of the loop in terms of the values of all variables at the beginning of the k-th iteration. We do this by inserting a set of dummy given information on the exit arc for the LOOPHEAD node, a set of equalities of the form:

$$v_{k+1} = v_k \quad ,$$

for each variable v in the program. We then process the nodes in the loop in exactly the way described above, proving assertions, merging information, and, most importantly, changing the v_{k+1} expressions to reflect assignments. The only difference between the first and second passes through a loop is the given information attached to the LOOPHEAD exit arc, and a flag which says "don't worry if some assertions cannot be proved on the first pass".

At the end of the first pass, we call the loop induction routine with two sets of information: (1) the initial condition of the program's variables upon entry to a loop, and (2) the recurrence relations between the values of all variables at the beginning of the k-th iteration of a loop and their values at the beginning of the k+1-st iteration. The loop induction routine has the responsibility for synthesizing a description of the range of values taken on by each variable within the loop.

In general, it is an unsolvable problem to state the exact set of values that a variable takes on within a loop, since deciding whether

the set of values for t is {true} or {true,false} in

```
t := true;
while t do
    ...;
```

is equivalent to solving the halting problem. However, there are a few useful special cases which are applicable to a large number of practical programs.

The discussion below will be in terms of variables which take on integer values, although many of the ideas can be extended to character values, list pointers, and perhaps floating-point numbers. As shown in Figure 4.7, we will focus on a variable v, with initial value v_0 on entry to the loop, with the recurrence relation $v_{k+1} = F(\bar{v}_k)$ (where $\bar{v}$ represents all the program variables and F is an arbitrary function), and with perhaps a set of relations between v and constants or other program variables.

Case 1. Invariant.

If the recurrence relation is $v_{k+1} = v_k$, then v is invariant inside the loop, so its value there is v_0.

Case 2. Monotonic relationships.

If either $v_{k+1} \geq v_k$ or $v_{k+1} \leq v_k$ is implied by the recurrence relations, then v is either monotonically nondecreasing or nonincreasing inside the loop. For simplicity, we assume the first case. The possible values of v therefore are a subset of the infinite set $\{v_0, v_0+1, v_0+2, \ldots\}$. If the recurrence relations also include an inequality like $v_{k+1} \leq x$, where x is invariant in the loop, then we can bound the infinite set:

$$v \subset \{v_0, v_0+1, v_0+2, \ldots, x\} \quad .$$

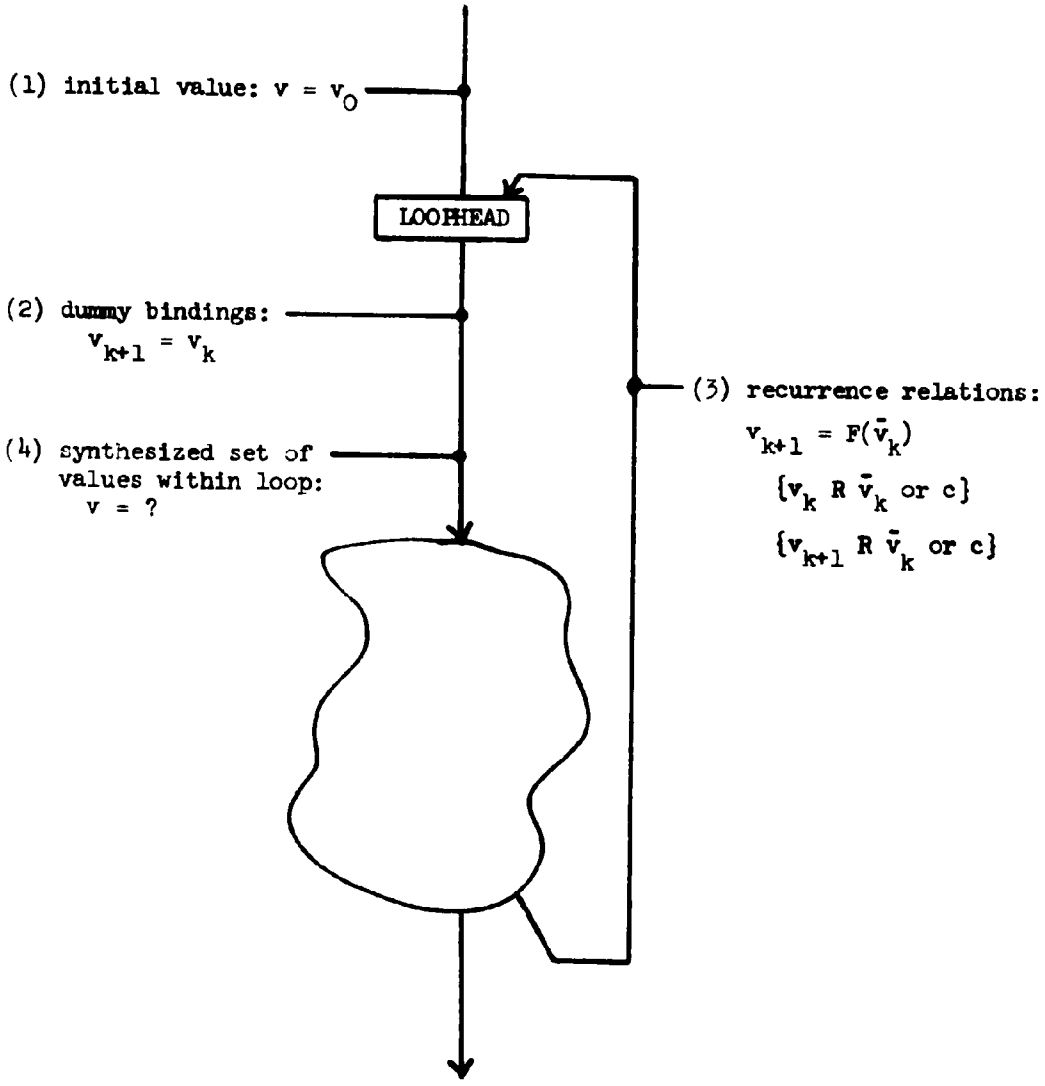

Figure 4.7. Model for loop induction on variable v .

If $v_{k+1} = v_k + c$, where c is a positive constant, then $v \subset \{v_0, v_0+c, v_0+2c, \ldots\}$ and v is strictly increasing. If v is increasing uniformly and bounded by x (invariant), then v takes on all of the values in a set:

$$v = \{v_0, v_0+c, \ldots y\}$$

where y is the largest element of the set that satisfies the bound x.

If the bound on v is $v \neq x$ instead of $v \leq x$, then the analysis is more complicated. Given

$$v \subset \{v_0, v_0+c, v_0+2c \ldots\}$$

and

$$v \neq x ,$$

then the values v takes on are bounded by x if and only if x is an element of the set:

i.e., $v = \{v_0, v_0+c, v_0+2c, \ldots x\}$

iff $x \in \{v_0, v_0+c, v_0+2c \ldots\}$.

Otherwise the restriction $v \neq x$ is meaningless and can be thrown away. Note that x can fail to be in the set either because $x-v_0$ is not a multiple of c, or because $x < v_0$.

Case 3. Searching.

The last case above is perhaps better viewed as a search, not as a bound: v takes on various values in a set, while searching for $v = x$. We encounter a more general kind of search when v is a subscripted reference:

$$v \subset \{A[i_0], A[i_0+c], A[i_0+2c] \ldots\}$$

and $v \neq x$.

In this case, the set is finite if and only if the search is satisfied, i.e.,

$$x \in \{A[i_0],\ A[i_0+c],\ A[i_0+2c] \ldots\}$$

or equivalently,

$$\exists n \in \{i_0,\ i_0+c,\ i_0+2c \ldots\} \quad \text{s.t.} \quad A[n] = x \quad .$$

Some search loops terminate on a forced match, as in:

```
A[max] := x; i := 1;
   while A[i] ≠ x do i := i+1;    .
```

In our analysis of such a loop, the initial conditions include:

$$i = 1$$

$$A[max] = x \quad ,$$

and the recurrence relations include:

$$i_{k+1} = i_k + 1$$

$$x_{k+1} = x_k$$

$$A_{k+1} = A_k$$

$$A_k[i_k] \neq x_k \quad .$$

Combining these, we attach the following <u>given</u> information to the exit arc for the LOOPHEAD node:

x is invariant in the loop

A (the whole array) is invariant

$A[max] = x$

$i \rightarrow_\varepsilon \{1, 2, 3, \ldots\}$.

(The notation $\rightarrow_\varepsilon$ means "takes on each of the values in a subset consisting of the first n elements of the ordered set, for some $n \geq 1$.") With no other driving goal, this is the end of our analysis. However, when we try to prove the loop termination assertion,

$$\exists\ k \geq 1 \quad \text{s.t.} \quad A_k[i_k] = x_k \quad ,$$

we first remove the subscripts k on the invariant variables:

$$\exists k \geq \quad \text{s.t.} \quad A[i_k] = x \quad ,$$

then we look at the initial entry given information and find that

$$A[max] = x \quad ,$$

Following the reasoning above,

$$A[max] \in \{A[1], A[2], A[3], \ldots\}$$

$$\text{if} \qquad max \in \{1, 2, 3, \ldots\} \quad .$$

Since max is also invariant in the loop, we may push this relation outside the loop as an initial entry condition:

the search loop terminates if $max \geq 1$.

[For other search loops, see Appendix A, Example 1, and the program SELECT in [Sites 1974].]

In summary, we have discussed a disciplined way of gathering information for proving assertions attached to a flow graph, including ideas for eliding tests, merging information into useful lemmas, and proving the termination of search loops.

Chapter 5. Related Literature

Highlights of related literature include Floyd's original paper on inductive assertions [Floyd 1967]; theses by James King [King 1969], Susan Gerhart [Gerhart 1972], and L. Peter Deutsch [Deutsch 1973]; surveys by Bernard Elspas [Elspas *et al.* 1972b], and Ralph London [London 1972]; the comprehensive, but now somewhat out of date bibliography of Ralph London [London 1970a]; and the complete conference proceedings from the Symposium on Semantics of Algorithmic Languages (see [Hoare 1971b]), and from the Las Cruces conference on Proving Assertions about Programs (see [Manna *et al.* 1972]).

⊗

Since the publication of Floyd's original paper on the inductive assertion method [Floyd 1967], many people have worked on mechanizing the creation and proof of verification conditions, given as input an annotated flow chart of the program.

James King built a program verifier [King 1969] which would accept simple Algol-like programs as input and produce proofs of their correctness with respect to a set of inductive assertions supplied by the user. The assertions are specified by ASSERT statements at appropriate points in the program. There must be enough assertions supplied to break all paths around loops, and more than the minimum number of such assertions may be useful for helping the verifier to distinguish the different cases involved in multiple paths around a loop. King's system can synthesize an input assertion if none is supplied, essentially stating "these input conditions are necessary for the other assertions to be true." King's work was the original inspiration for the present thesis.

Chapter 5. Related Literature

Susan Gerhart described a system for verifying APL programs [Gerhart 1972], in which the user supplied inductive assertions as comments. Since APL can express vector operations without explicit loops, the proofs were much shorter than for equivalent Algol-like programs. The system is capable of synthesizing and proving some assertions from the known semantics of APL operations such as asserting that the left operand of some operator must be a scalar, then proving that that will always be true because the left operand is in turn the result of some other operator which always returns a scalar value. This parallels our interest here in proving that a program contains no semantic errors. Gerhart also suggests a broader view of verification, including various forms of semantic checking of programs in lieu of debugging. She introduces the term formal debugging, which means obtaining information about a program from the structure or semantics of a program without executing it. Again, this parallels our interest here in giving a user feedback on the inherent internal self-consistency or inconsistency of a program.

Peter Deutsch built an interactive program verifier [Deutsch 1973], which in some sense represents a second generation of verifiers. It accepts input in a form quite similar to King's, but uses more sophisticated proof techniques to do all of King's examples, plus some harder ones. Richard Waldinger and Karl Levitt discuss a similarly-ambitious proof system running at SRI [Waldinger and Levitt 1973]. The emphasis of both systems is to extend the coordination between theorem provers and the kinds of theorems which occur when verifying programs.

Chapter 5. Related Literature

All four of the above systems bypass the problem of proving that programs terminate. Donald Good also did a related thesis [Good 1970].

Various issues of theorem prover heuristics and refinements are discussed in [Wegbreit 1974], [Elspas et al. 1972a, b], [London 1972], and [Smith 1972]. Smith refers to the problem that "... we cannot prove correctness of programs in the mathematical sense as suggested by McCarthy, more due to our inability to state what we are trying to prove than to our inability to find actual proof methods."

Hoare describes the development of an axiomatic approach to proving correctness of programs, with successive extensions to include programs with function calls and programs with jumps [Hoare 1969, 1971b] [Clint and Hoare 1972]. It is instructive to compare the various proofs of the correctness of the program FIND [Hoare 1961] found in [Hoare 1971a], [Waldinger and Levitt 1973], [Deutsch 1973], and [Igarashi et al. 1973].

Manna has explored the formal basis for induction on recursive program schemata, using first and second order predicate calculus [Manna 1969]. An excellent survey of the various kinds of induction can be found in [Manna et al. 1972]. In [Manna and Pnueli 1973], total correctness of a program (i.e., proof of termination and correctness with respect to the assertions) is discussed, but only in terms of well-founded monotonic sequences supplied by a human user. Manna's 1969 article also discusses total correctness. Total correctness is also mentioned in [Burstall 1970], but his proof of termination for a program to calculate 2^n fails to state the necessary restriction that $n \geq 0$.

Chapter 6. Extensions and Related Topics

This chapter discusses extensions of the techniques presented to cover a larger class of programs and to increase the proportion of successful proofs. Also, these techniques are related to issues in language design, to optimizing compilers, and to the current controversy about GO TO statements. ⊗

Extensions

The major extension of the work described here would of course be an implementation. Machines are much better than reading committees and referees at keeping one honest. Many of the parsing, flow graph manipulation, and theorem proving pieces of such a system exist, but it remains to pull them all together and build a coherent whole. It is important in such a system to build in heuristics, tuning, and biases to make decisions similar to those of a human about what information and lemmas in a program are important. For instance, in Example 7 of Appendix A (bubble sort), it takes a complicated deduction about the ordering of elements in the array to prove that eventually no interchanges are required and that the program therefore terminates. However, in a somewhat similar program, keeping track of the possible relationships between elements of an array might be wasted effort which does not contribute at all to the proof of some assertion.

A second major extension would be the inclusion of pointers, lists, and trees as data objects.

The system could also be extended to allow the user to state some extra assertions that he would like proved along with all the synthesized ones. These extra assertions could be merely to provide useful lemmas to the theorem prover, but they also could be used to describe the data structure that the program operates on, and to ask

the system to prove that that data structure is preserved by the program under all possible circumstances. Extra assertions could also be used to describe simple consistency checks on the intermediate data or results of a program. While preserving consistency checks is only a small step toward a program's being certified "totally correct", such checks are often easy to state and have a high payoff in detecting simple errors.

One particularly useful check is to prove that for some set (array, linked list, tree, etc.), no elements of that set are "lost" during the execution of the program. For instance, in a sorting program, it is useful to prove that the output is a permutation of the input. An alternate way to state this, which may be easier to prove, is that all of the elements of the input set are elements of the output set. If the set is an array being sorted in place, then it is only necessary to prove the local condition that whenever an element of the array is destroyed, by assignment to it, a copy of that element must exist either somewhere else in the array or in another program variable. If the steady state is that all elements are somewhere in the array, then the proof system would only have to keep track of those (typically) one or two variables which contain copies of elements being manipulated, and to detect the point in the program when the steady state is reached again. In keeping track of copies of array elements, it would be necessary to re-assign the "unknown" value ω to all local variables upon leaving a block, to reflect the possibility of "losing" an element exactly at block exit. A similar technique could be used in programs which manipulate list structures, asking the system to prove that no node in an input structure is lost by

ending up with nothing pointing to it. With a model of the steady state of the structure, the proof system must keep track of the few link fields and pointer variables which do not match the model midway in the execution of a valid change to the data structure. For such programs, it is necessary for the user to use a declaration-like language to describe the intended steady state of his data structure. Some of the concepts in [Dahl and Hoare 1972] might be useful in defining such a language.

The system can also be extended to include programs which operate on floating-point numbers. Although this is a difficult area, sufficient tools are becoming available to treat floating-point computations very precisely. For example, [Malcolm and Palmer 1974] treat an algorithm for solving tridiagonal equations in terms of computer arithmetic instead of real number arithmetic. [Good and London 1970] give Algol procedures for interval arithmetic designed to use computer arithmetic and prove that they work. [Hull et al. 1972] combines Floyd assertions with backward error analysis. [Yohe 1970] discusses floating-point arithmetic, using case analysis and assertions.

In the system presented, there are weak areas which need more refinement and sophistication. These areas include the rules to be applied during loop induction, the rules to be applied when merging given information, and the rules for deciding to move a false assertion to an earlier place in the program. Extending the current system to recursive programs may not be easy.

Chapter 6. Extensions

Language Design

As Gerhart has pointed out in her thesis [Gerhart 1972], one of the difficulties in mechanically analyzing a program is to model the effect of a loop on a set of data. For example, it is hard to mechanically deduce from

```
        i := 1;
loop:   A[i] := 0;
        i := i+1;
        if i ≤ n then goto loop;
```

that

$$A[1] = 0 \quad \text{and} \quad \forall\, 2 \leq i \leq n\,,\ A[i] = 0\ .$$

(The programmer may well have either "known" or assumed that $n \geq 1$, but the proof system has to entertain other possibilities.) It is much easier to mechanically deduce what happens in the APL statement

$$A \leftarrow n \rho 0\ ,$$

where, among other things, it is not possible for some of the elements of A to be left undefined. From considerations such as the above, it is easy to conclude that, to the extent that a language can express operations without explicit loops, it will be easier to prove properties about programs in that language.

As part of our proposed style of explicitly stating all of the restrictions on the inputs of a program, it would be useful for a language to have an assert statement. The form of the statement could be:

```
assert boolean expression
```

and its meaning would be:

```
if boolean expression then comment OK;
                      else terminate uncleanly
```

Such a construct was added to the Stanford Algol W compiler by Ed Satterthwaite in 1970 [Sites 1972, p. 49]. The statement could either compile into executable code or not, depending on the user's choice of faster execution versus complete checking, but in either case, the proof system can assume that the expression is true, and prove that the program terminates cleanly with respect to the stated assumptions. Perhaps most importantly, the statement serves as documentation embedded in the source code of the program, so that anyone reading the program immediately knows the assumptions which need to be satisfied for correct operation.

Optimizing Compilers

Most of the techniques presented here attempt to gather the same kind of information about a program that an optimizing compiler does. While optimizing compilers often generate code which executes quickly and which completely ignores run-time error checking, it is possible to have both fast execution and careful error checking. For instance, the compiler can logically generate subscript bounds checking code for every array access, then use techniques like those presented here to prove that many of the tests are unnecessary, either because the subscript expression is inherently in range (perhaps generated by a containing for loop), or because the same expression is used in an earlier reference and need be checked only once.

In a language like APL, the same idea can be used to prove that the interpreter need not bother with some conformability checks, and can perhaps be used to prove that the shape, size, and type of variables used in some statements are static enough that the statements could be compiled instead of interpreted.

Chapter 6. Extensions

Another issue which occurs in optimizing compilers is safety, the question of whether a calculation can be moved to a less-frequently executed place without introducing semantic errors which would not have occurred in the original location. The classic example is:

```
if x ≠ 0 then
    y := 1/x;
```

in which some external consideration suggests moving the calculation of 1/x to a place in front of the test. This move is unsafe if x could equal zero at the new place. The information gathered by the system described here to prove that 1/x does not produce a semantic error is exactly the information needed by an optimizing compiler to decide if the movement of 1/x is safe.

The GOTO Controversy

One of the classic examples to support the argument that eliminating go to's can introduce redundant computation is this search loop from [Knuth and Floyd 1971]:

```
for i := 1 step 1 until n do
    if A[i] = x then go to found;
notfound: n := i; A[i] := x; B[i] := 0;
found: B[i] := B[i]+1;
```

which can be modified to:

```
i := 1;
while i ≤ n and A[i] ≠ x do
    i := i+1;
if i > n then
    begin n := i; A[i] := x; B[i] := 0 end;
B[i] := B[i]+1;
```

which has a redundant test $i > n$ just below the loop. The test elision technique described in Chapter 4 will eliminate the redundant test, as shown in Figure 6.1, thus making the go to -less form just as efficient as the go to form (admittedly, requiring a smarter compiler).

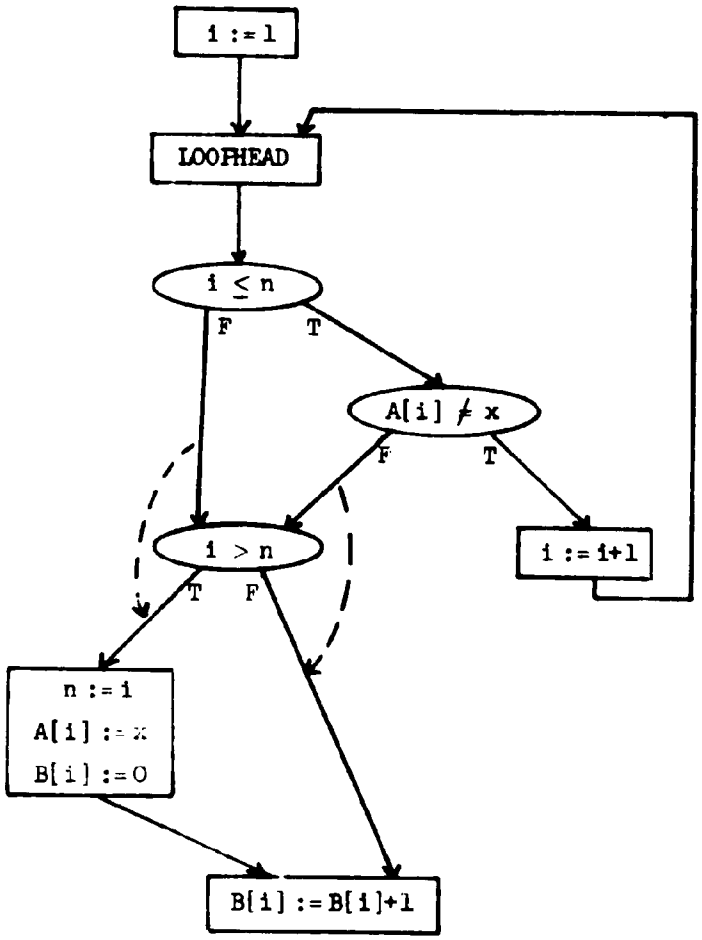

Figure 6.1. Test elision makes this goto-less search loop potentially as efficient as its goto form.

Chapter 7. Conclusion

It is my hope that eventually a system will be built which will be better than a human programmer at taking an unknown program and discovering what it does and how it can fail. Perhaps the ideas in this thesis will form a small piece of the foundation of that system.

"I would rather write programs
to help me write programs than
write programs."

Appendix A. Examples

This appendix contains ten examples worked out in various levels of detail: King's nine examples, and McCarthy's 91 function. Two other examples appear in [Sites 1974]: Floyd's TREESORT3, and Rivest's SELECT (a linear-time median-finder). To my knowledge, SELECT has not been examined formally before. Proofs of partial or total correctness of the other programs can be found in the following references: King's examples [King 1969], [Deutsch 1973]; the 91 function [Manna et al. 1972].

As a reference aid, the outline below summarizes the issues discussed in each example.

Example 1. Multiplication.

- First and second passes through a loop.
- Sequence of four stages of loop given information.
- Termination of search for $y = 0$.
- Infinite loops vs. overflow detection.
- Pushing invariant assertion out of loop, false assertion toward front of program.

Example 2. Division

- Proof of partial correctness vs. proof of clean termination.
- Distinction between first iteration given information and subsequent iterations.
- Pushing invariant assertion out of loop.
- Correlation between usage of two variables in proving that overflow can't occur.

Example 3. Exponentiation.

- Integer division model.
- Loop termination for absolute value approaching zero.
- Merging given information and forming refinement.

Appendix A. Examples

Example 4. Primality.

Multiple exit tests.
Overflow assertion actually proved.
Termination proof unrelated to what program does.

Example 5. Zeroing.

Arrays, subscript bounds.
Union of sets and " $\rightarrow\varepsilon$ " notation used in careful description of values of i .

Example 6. Maximum.

Alias problem for arrays.
Forward vs. backward analysis.
Lemma discovery during merging of given information.
Proof of no overflow only on second pass through loop.

Example 7. Bubble Sort.

Failure to prove termination.
Permutation for leading tests.
Two nested loops.

Example 8. Multiplication via increment/decrement.

Use of refinement of merged information.
Test elision forces node splitting.
Re-analysis of loop structure makes two parallel loops.
Mechanical removal of invariant test from loop.

Example 9. Selection Sort.

Termination proof unrelated to what program does.
Two nested loops, with successively more useful given information on four passes through inner loop.

Appendix A. Examples

Example 10. The 91 Function.

Transformation of single interval into two nested loops.

Permutation for leading tests.

Two nested loops, with successively more useful given information on four passes through inner loop.

Use of refinement of merged information.

Test elision removes an inner loop.

Partial correctness plus proof of clean termination gives total correctness.

Example 1: King's Example 1. Multiplication.

This example is worked out in complete detail, to give a cohesive, concrete example of the whole process discussed in Chapters 1 - 4. Subsequent examples will only pick out highlights. All of the first nine examples are modifications of the nine examples in King's thesis. The modifications consist of assigning the undefined value ω to all variables at the beginning of a program, and inserting READ statements for those variables which are essentially input parameters. Also, the inductive assertions which King supplied are stripped out.

Figures A1.1 through A1.7 are self-explanatory. The commentary picks up again after Figure A1.7.

Example 1. Multiplication

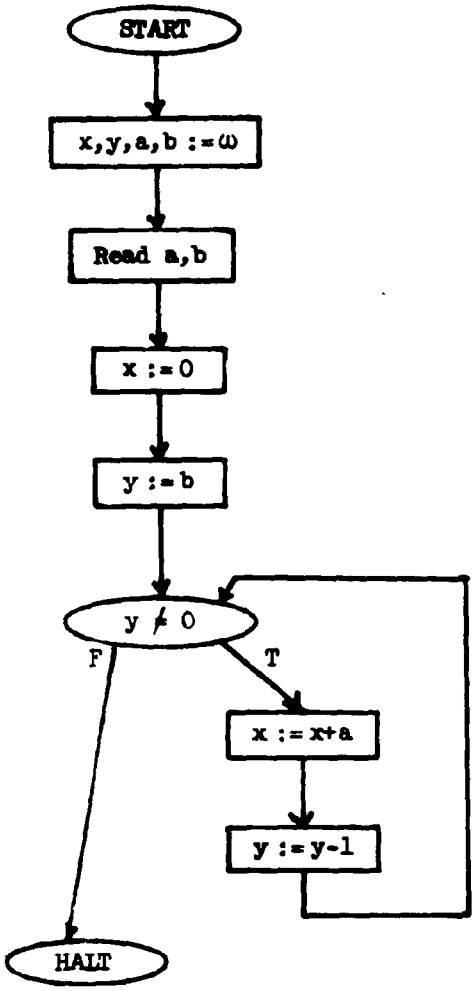

Figure A1.1. Input flow graph for program to calculate $x = a * b$ by successive additions. Nothing else is supplied by the human user.

Example 1. Multiplication

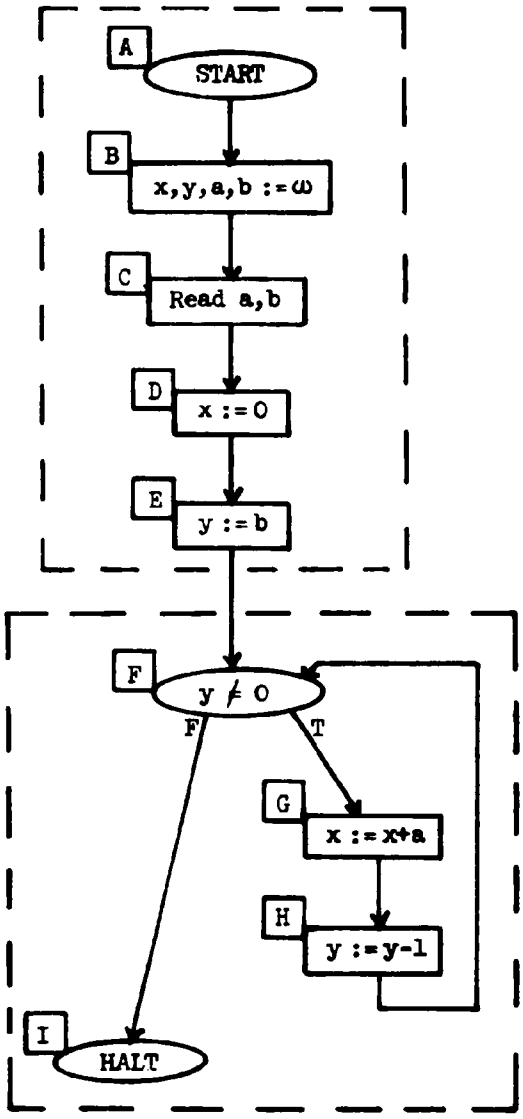

Figure A1.2. Interval analysis of flowgraph A1.1. The nodes are labeled for reference.

Example 1. Multiplication.

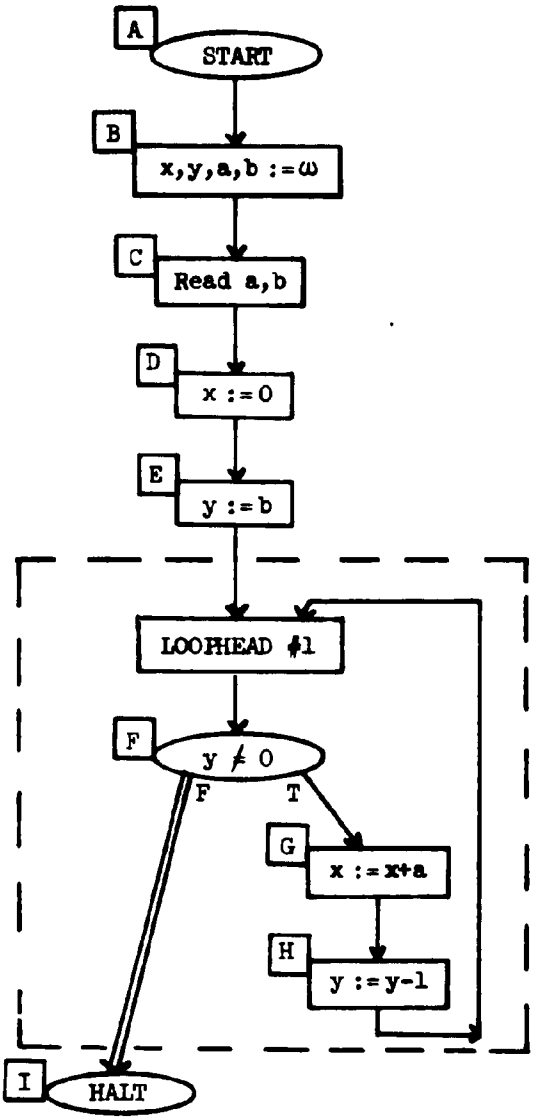

Figure A1.3. The graph of Figure A1.2 after insertion of the LOOPHEAD node and identification of the loop exit arc from node F to node I. All loop exit arcs will be shown with double lines. Every path around the loop goes through an exit test (node F) and all exit tests are just after the LOOPHEAD node. Topological sorting of the nodes yields the following order for subsequent processing: A-E, LOOPHEAD #1, F-I. This graph represents all the modifications described in Chapter 1.

Example 1. Multiplication

Node	Operator	Assertion
A. START	--	--
B. x,y,a,b := ω	:=	--
C. Read a,b	Read	--
D. x := 0	:=	--
E. y := b	:=	$b \neq \omega$
LOOPHEAD #1	--	--
F. $y \neq 0$	$\neq$	$y \neq \omega$
G. x := x+a	+	$x \neq \omega \wedge a \neq \omega \wedge I_{min} \leq x+a \leq I_{max}$
	:=	--
H. y := y-1	-	$y \neq \omega \wedge I_{min} \leq y-1 \leq I_{max}$
	:=	--

Figure A1.4. Generation of semantic error assertions. Assertions for small constants ($0 \neq \omega$, $1 \leq I_{max}$) are ignored.

Loop	Exit test	Assertion
#1	$y \neq 0$	$\exists\, k \geq 1$ s.t. $y_k = 0$

Figure A1.5. Generation of loop termination assertion.

Example 1. Multiplication

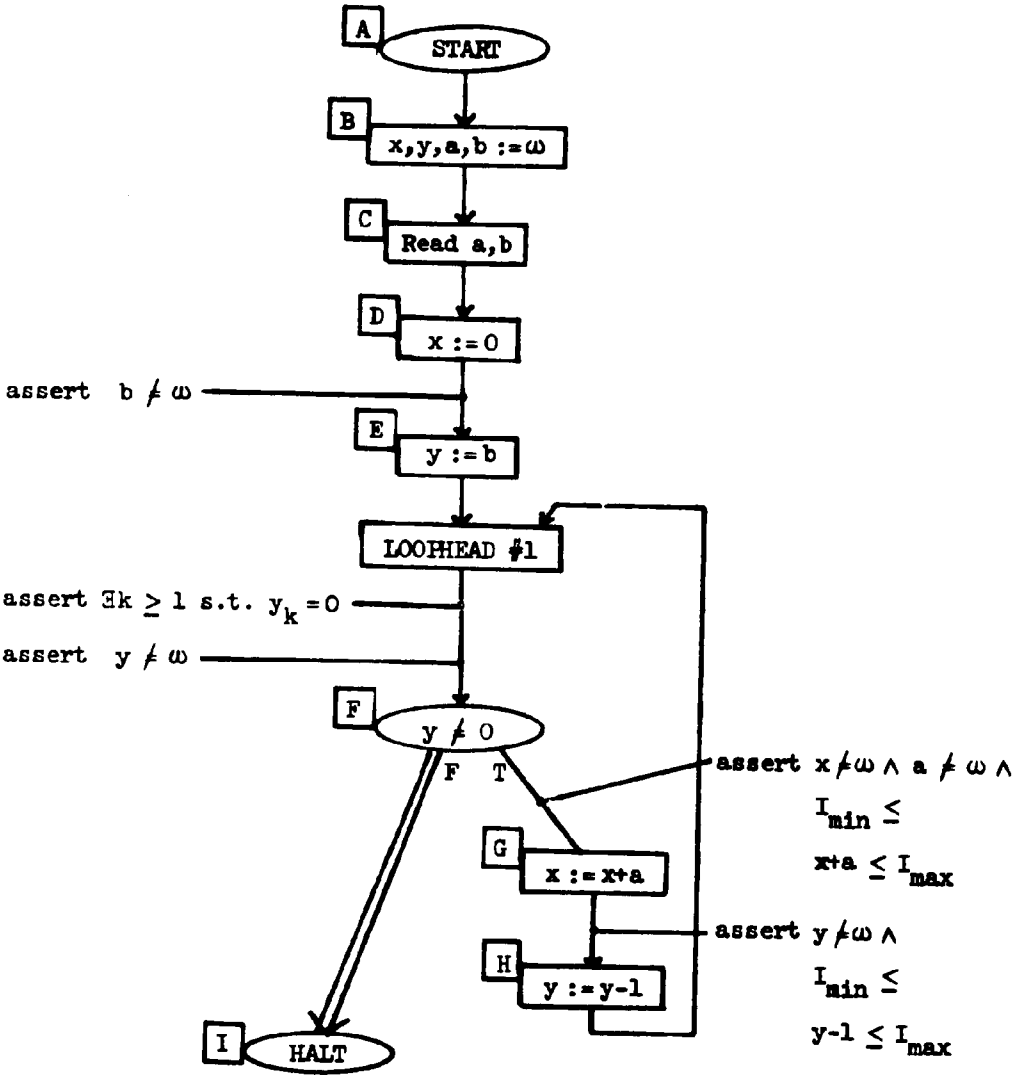

Figure A1.6. Flow graph A1.3 with semantic error and loop termination assertions attached. If all of these assertions are proved true, then the program always terminates cleanly. We will in fact find that there is nothing to prevent the overflow in node G, and that the loop won't terminate if b is negative. We will thus synthesize King's input assertion that $b \geq 0$. This graph represents all the processing described in Chapters 1-3.

Example 1. Multiplication

Node	Input "given" info	Assertions to Prove	Output "given" info
B: x,y,a,b = ω	--	--	$x = \omega \wedge y = \omega \wedge a = \omega \wedge b = \omega$
C: Read a,b	$x = \omega \wedge y = \omega \wedge a = \omega \wedge b = \omega$	--	$x = \omega \wedge y = \omega \wedge a \neq \omega \wedge b \neq \omega$ (Semantics of Read say that a and b are defined, therefore between I_{min} and I_{max}, but nothing else.)
D: x := 0	$x = \omega \wedge y = \omega \wedge a \neq \omega \wedge b \neq \omega$	--	$x = 0 \wedge y = \omega \wedge a \neq \omega \wedge b \neq \omega$
E: y := b	$x = 0 \wedge y = \omega \wedge a \neq \omega \wedge b \neq \omega$	$b \neq \omega$ True.	$x = 0 \wedge y = b \wedge a \neq \omega \wedge b \neq \omega$
LOOPHEAD #1	$x = 0 \wedge y = b \wedge a \neq \omega \wedge b \neq \omega$	--	First pass thru loop: Attach the following symbolic info to exit arc of LOOPHEAD node: $x_{k+1} = x_k \wedge y_{k+1} = y_k \wedge a_{k+1} = a_k \wedge b_{k+1} = b_k$

Figure A1.7. Proof processing of nodes A - E , and first pass thru loop # 1, nodes F - H .

Example 1. Multiplication

Node	Input "given" info	Assertions to Prove	Output "given" info
F: $y \neq 0$	$x_{k+1} = x_k \wedge$ $y_{k+1} = y_k \wedge$ $a_{k+1} = a_k \wedge$ $b_{k+1} = b_k$	$\exists k \geq 1$ s.t. $y_k = 0$ $y \neq \omega$ Neither proven Test elision: Try to prove that $y = 0$ or $y \neq 0$ on some incoming path, but no luck.	True arc: $x_{k+1} = x_k \wedge$ $y_{k+1} = y_k \wedge$ $a_{k+1} = a_k \wedge$ $b_{k+1} = b_k \wedge$ $y_k \neq 0$ False arc: same, except last term is $y_k = 0$.
G: x := x+a	$x_{k+1} = x_k \wedge$ $y_{k+1} = y_k \wedge$ $a_{k+1} = a_k \wedge$ $b_{k+1} = b_k \wedge$ $y_k \neq 0$	$x \neq \omega \wedge a \neq \omega \wedge$ $I_{min} \leq x+a \wedge$ $x+a \leq I_{max}$ None proven.	$x_{k+1} = x_k + a_k \wedge$ $y_{k+1} = y_k \wedge$ $a_{k+1} = a_k \wedge$ $b_{k+1} = b_k \wedge$ $y_k \neq 0$
H: y := y-1	$x_{k+1} = x_k + a_k \wedge$ $y_{k+1} = y_k \wedge$ $a_{k+1} = a_k \wedge$ $b_{k+1} = b_k \wedge$ $y_k \neq 0$	$y \neq \omega \wedge I_{min} \leq$ $y-1 \leq I_{max}$ Last part, $y-1 \leq I_{max}$, is true.	$x_{k+1} = x_k + a_k \wedge$ $y_{k+1} = y_k - 1 \wedge$ $a_{k+1} = a_k \wedge$ $b_{k+1} = b_k \wedge$ $y_k \neq 0$

Figure A1.7 (continued). Proof processing of nodes A - E , and first pass through loop # 1 , nodes F - H .

Example 1. Multiplication

After the first pass through the loop with the symbolic variables x_{k+1} , x_k , etc., we have developed a set of recurrence relations for the values of all variables at the beginning of the k+1-st iteration of the loop in terms of their values at the beginning of the k-th iteration. From these recurrence relations on the latchback arc, and the initial entry conditions on the arc from node E to the LOOPHEAD node, we now must synthesize expressions for the values of all variables during all iterations of the loop, as shown in Figure A1.8.

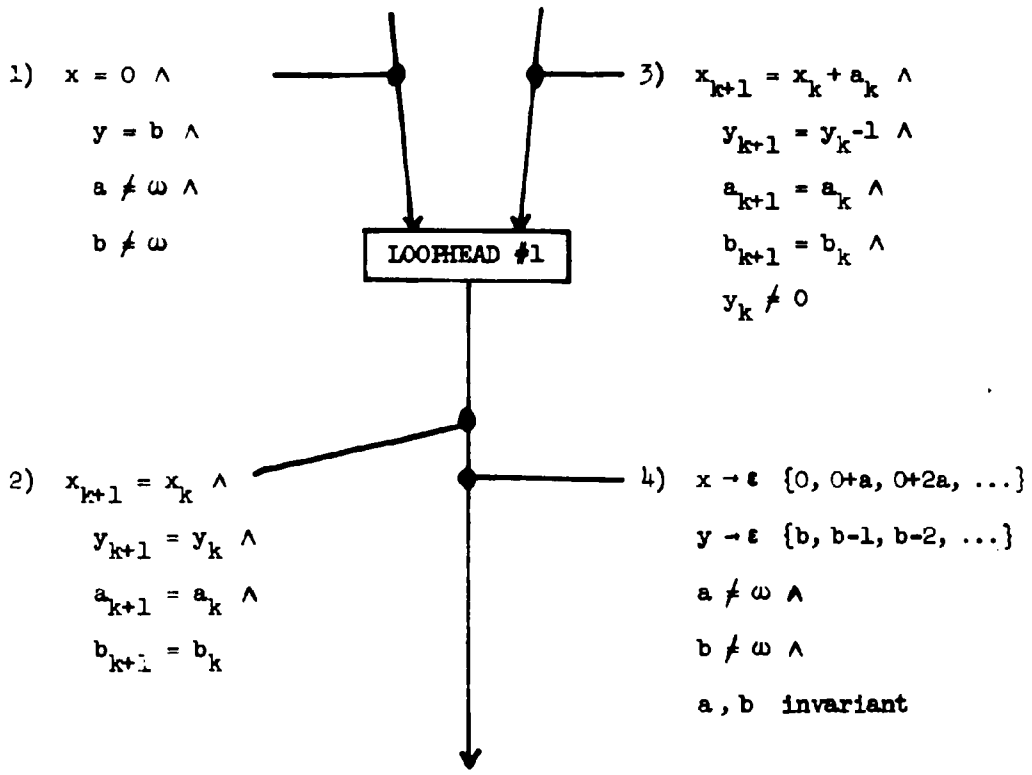

Figure A1.8. Loop induction, using all facts except $y_k \neq 0$ to detect variables invariant within the loop and to synthesize infinite sets which encompass the sets of values for x and y . The notation → ε means "for some $n \geq 1$, takes on each of the values in the subset consisting of the first n elements of the ordered set".

Example 1. Multiplication

If the exit test for the loop were

$$y > 0$$

instead of

$$y \neq 0 \ ,$$

then we could easily conclude at this point that

$$0 \leq y \leq b$$

inside the loop. But, given the comparison for exactly zero, we must work a little harder to synthesize a range of values for y inside the loop.

If $0 \in \{b, b-1, b-2, \ldots\}$ (i.e., $b \geq 0$), then the set of values y takes on is finite and bounded by zero:

$$y = \{b, b-1, b-2, \ldots, 0\} \ ,$$

where the equality sign means that y must necessarily take on the value of each and every member of the set exactly once.

If $0 \notin \{b, b-1, b-2, \ldots\}$ (i.e., $b < 0$), then the set of values y takes on is essentially infinite. In reality, the set of values for y is bounded by I_{min}, but we discover this fact by assigning y the infinite set, then failing to prove that underflow never occurs in the statement

$$y := y-1 \ .$$

In our induction processing we try to decide if the values of y are a finite or infinite set by examining the initial value of b and asking if $b \geq 0$ (i.e., if $0 \in \{b, b-1, b-2, \ldots\}$). We find no answer to this question, only the information that $b \neq \omega$. So we give up; knowing nothing else about b, the best we can say about y is that

$$y \rightarrow \varepsilon \ \{b, b-1, b-2, \ldots\} \ .$$

With the ranges of values synthesized in Figure A1.8, we proceed to take a second pass through all the nodes of the loop, using the ranges to prove assertions and to develop new given information on subsequent arcs. The first assertion we try to prove is the loop termination assertion:

$$\exists\, k \geq 1 \quad \text{s.t.} \quad y_k = 0 \quad .$$

Now the groundwork of the above discussion about synthesizing the range of values of y becomes useful: we discovered above that y will take on the value zero iff $b \geq 0$, so the loop termination assertion is equivalent to asserting that, on the exit arc of the LOOPHEAD node,

$$b \geq 0 \quad .$$

Since b is invariant in the loop, we can push this assertion back to the initial entry arc of the loop, and then as far back as the Read node, as in Figure A1.9. Figure A1.10 details the remaining proofs, during the second pass through the loop, plus any subsequent nodes.

The complete process leaves us with two unproved assertions:

$b \geq 0$ before node D

and $I_{min} \leq x+a \leq I_{max}$ before node G .

If the user can guarantee that these two assertions are always true, then the program terminates cleanly. If the user cannot guarantee that these assertions are always true, then they describe the only two ways in which the program can "blow up" during execution:

if $b < 0$, then loop #1 never terminates

and if $I_{min} > x+a$ or $x+a > I_{max}$, then an overflow occurs at node G .

The user is assured that there is no other way (such as an overflow in node H) for the program to blow up.

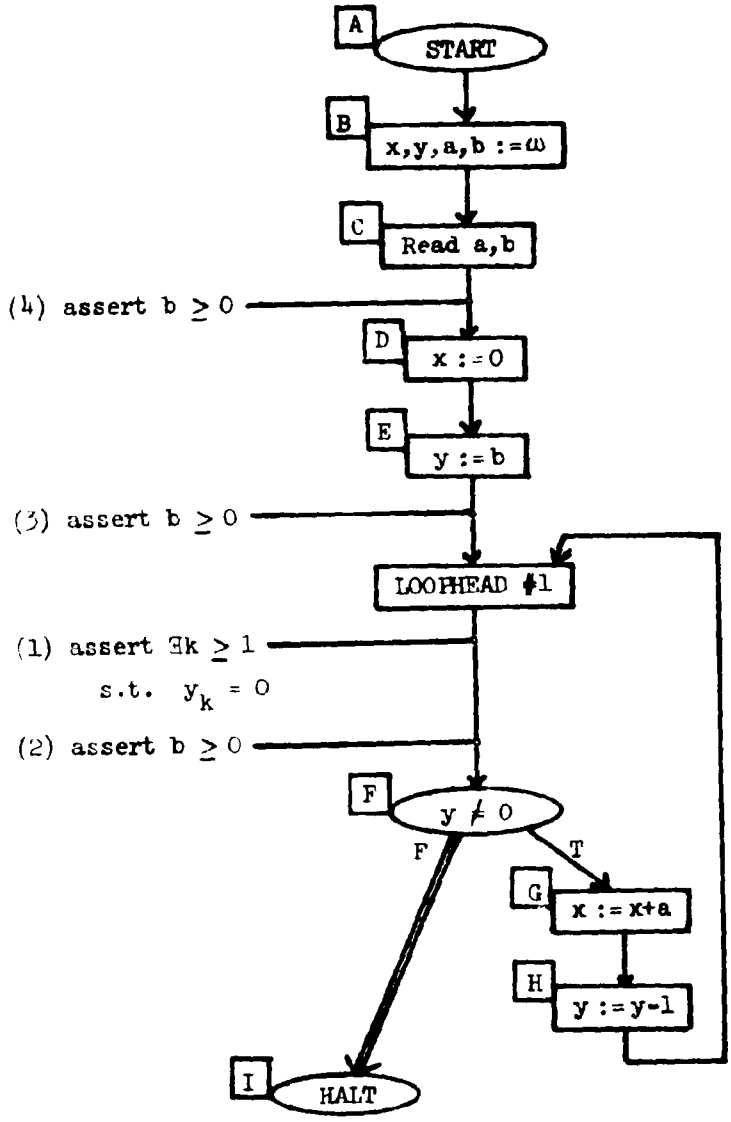

Figure A1.9. Steps in synthesizing the input assertion $b \geq 0$.

(1) Original loop termination assertion, generated from exit test.

(2) Equivalent loop termination assertion, generated from analysis of the set of values y takes on inside the loop.

(3) Assertion moved back, outside of the loop, because it is invariant in the loop and must therefore be true on entry.

(4) Assertion moved back as far as possible, in this case, to the exit arc of the Read statement. Note that this is the best place for the user to insert an executable test that the value read for b is in fact non-negative.

Example 1. Multiplication

Node	Input "given" info	Assertions to Prove	Output "given" info
LOOPHEAD #1	--	--	Second pass thru loop: Attach synthesized "given" info: $x \rightarrow \varepsilon \{0,0+a,0+2a,\ldots\} \wedge$ $y = \{b,b-1,b-2,\ldots,0\} \wedge$ $a \neq \omega \wedge b \geq 0$
F: $y \neq 0$	$x \rightarrow \varepsilon \{0,0+a,0+2a,\ldots\} \wedge$ $y = \{b,b-1,b-2,\ldots,0\} \wedge$ $a \neq \omega \wedge b \geq 0$	$y \neq \omega$ true Test elision: $y = 0$ maybe $y \neq 0$ maybe	True exit: $x \rightarrow \varepsilon \{0,a,2a,\ldots\} \wedge$ $y = \{b,b-1,b-2,\ldots,1\} \wedge$ $a \neq \omega \wedge b \geq 0$ False exit: $x \rightarrow \varepsilon \{0,a,2a,\ldots\} \wedge$ $y = 0 \wedge$ $a \neq \omega \wedge b \geq 0$
G: x := x+a	$x \rightarrow \varepsilon \{0,a,2a,\ldots\} \wedge$ $y = \{b,b-1,b-2,\ldots,1\} \wedge$ $a \neq \omega \wedge b \geq 0$	$x \neq \omega \wedge a \neq \omega \wedge$ $I_{min} \leq$ $x+a \leq I_{max}$ First two are true, and last one is maybe	$x \rightarrow \varepsilon \{a,2a,3a,\ldots\} \wedge$ $y = \{b,b-1,b-2,\ldots,1\} \wedge$ $a \neq \omega \wedge b \geq 0$
H: y := y-1	$x \rightarrow \varepsilon \{a,2a,3a,\ldots\} \wedge$ $y = \{b,b-1,b-2,\ldots,1\} \wedge$ $a \neq \omega \wedge b \geq 0$	$y \neq \omega \wedge$ $I_{min} \leq y-1$ Both are true	$x \rightarrow \varepsilon \{a,2a,3a,\ldots\} \wedge$ $y = \{b-1,b-2,b-3,\ldots,0\}$ $a \neq \omega \wedge b \geq 0$

Figure A.1.10. Proof processing of second pass through loop, nodes F - H .

Example 2. King's Example 2. Division.

This example exposes some of the complications in actually proving a loop termination assertion when some input assumptions which were in the programmer's mind are not stated, and hence need to be synthesized by the proof mechanism. King assumed the restrictions that $a \geq 0 \wedge b \geq 0$ and then proved that the program is partially correct in generating the proper quotient and remainder, but he failed to note that the program never terminates if $b = 0$. To confront this termination issue directly, we state no assumptions about a and b, and see what restrictions can be automatically synthesized. (If, however, we used King's restrictions, our system would still complain about the $b = 0$ case.) The starting point for this and most subsequent examples is the modified flow graph of the program annotated with assertions (Figure A2.1). In this and all subsequent examples, we will ignore the "undefined variable" assertions, since their proofs are all essentially trivial.

After processing nodes A - D and taking a first pass through the loop, we have gathered the following information for the loop induction.

Initial entry: $q = 0 \wedge r = a$

Recurrence relations: $a_{k+1} = a_k \wedge b_{k+1} = b_k \wedge$
$q_{k+1} = q_k \wedge r_{k+1} = r_k - b_k \wedge$
$r_k \geq b_k$

From this information, it is straightforward to synthesize the ranges:

a invariant
b invariant
$q \rightarrow \epsilon \{0, 1, 2, \ldots\}$
$r \rightarrow \epsilon \{a, a-b, a-2b, \ldots\}$
$r_k \geq b$ or, equivalently, $r_{k+1} \geq 0$.

Example 2. Division

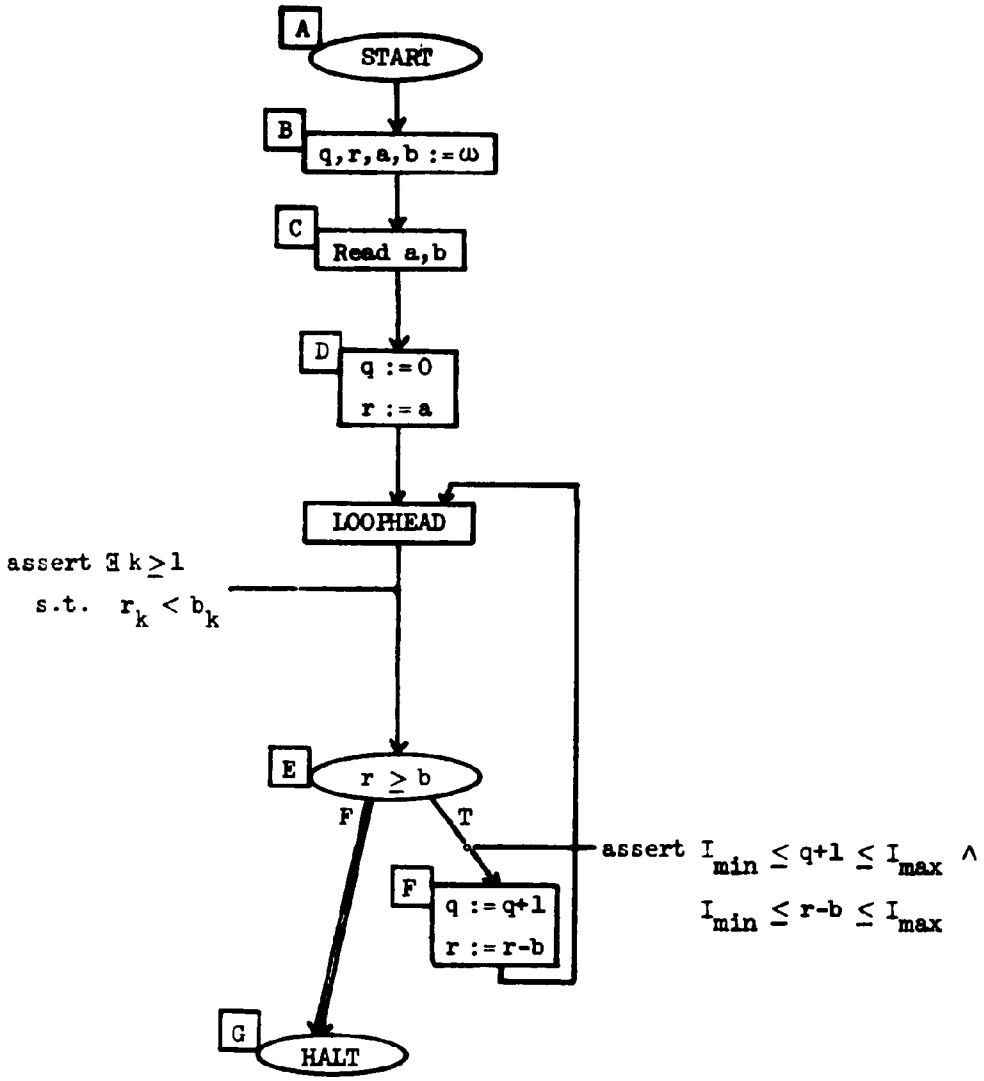

Figure A2.1. Mechanically annotated, modified flow graph for King's Example 2. Since we are working with little or no human assistance, King's assumptions that $a \geq 0$ and $b \geq 0$ are not supplied. The crux of this example is to synthesize appropriate restrictions on a and b .

We now want to prove the loop termination assertion:

$$\exists k \geq 1 \quad \text{s.t.} \quad r_k < b_k \quad ,$$

or, since b is invariant in the loop,

$$\exists k \geq 1 \quad \text{s.t.} \quad r_k < b \quad .$$

The assertion is not always true (e.g. if $a = 2$ and $b = -1$), and none of the techniques discussed in Chapter 4 will help synthesize appropriate restrictions on a and b which would make the assertion true. So the proof system would simply give up and direct the human user to supply appropriate restrictions on a and b.

There is, however, a useful heuristic for an automatic proof system to use: separate the case of zero iterations of the loop from the case of one or more iterations. To prove that $r_k < b$, we can try to prove that either $r_1 < b$ or that r is strictly monotonically decreasing. Only by explicitly considering r_1 as a special case can we pick up all the degenerate situations which result in zero executions in the loop.

Since $r_1 = a$, the condition

$$a < b$$

guarantees that the loop terminates (by never executing at all).

In the general case, r is monotonically decreasing if

$$r_{k+1} < r_k \quad .$$

From the recurrence relation $r_{k+1} = r_k - b$, we have

$$r_k - b < r_k$$

or

$$-b < 0$$

or

$$0 < b \quad .$$

Example 2. Division

Thus, we find that the loop terminates iff

$$(a < b) \vee (0 < b) \quad .$$

Since this relation is invariant inside the loop and must therefore be true on entry, we can push it outside the loop and then back to the READ node.

Examining the overflow assertions for node F, we find that the first of these, $I_{min} \leq q+1$, is clearly true, because adding a positive constant can never create a sum which is too negative. The second, $q+1 \leq I_{max}$, cannot be proved, and must be tossed back to the user with a "maybe". As described, our system cannot make any correlation between q and r, such as: q will be incremented as many times as r is decremented, so q cannot in fact overflow, if the loop terminates at all. For a slightly different loop with r := r+b instead of r := r-b in node F, the values $a = I_{max}$, b = -1 would result in q overflowing. So any attempt at correlating the overflow possibilities of one expression with the number of times another expression is executed must consider such factors as size of increment and total range covered by each expression.

The assignment r := r-b in node F cannot overflow if b > 0 because

$$b > 0 \supset r-b < I_{max}$$

and

$$b > 0 \wedge r \geq b \supset r-b \geq 0 > I_{min} \quad .$$

The same assignment cannot overflow if a < b because it is never executed.

The flow graph in Figure A2.2 represents the final result of our analysis.

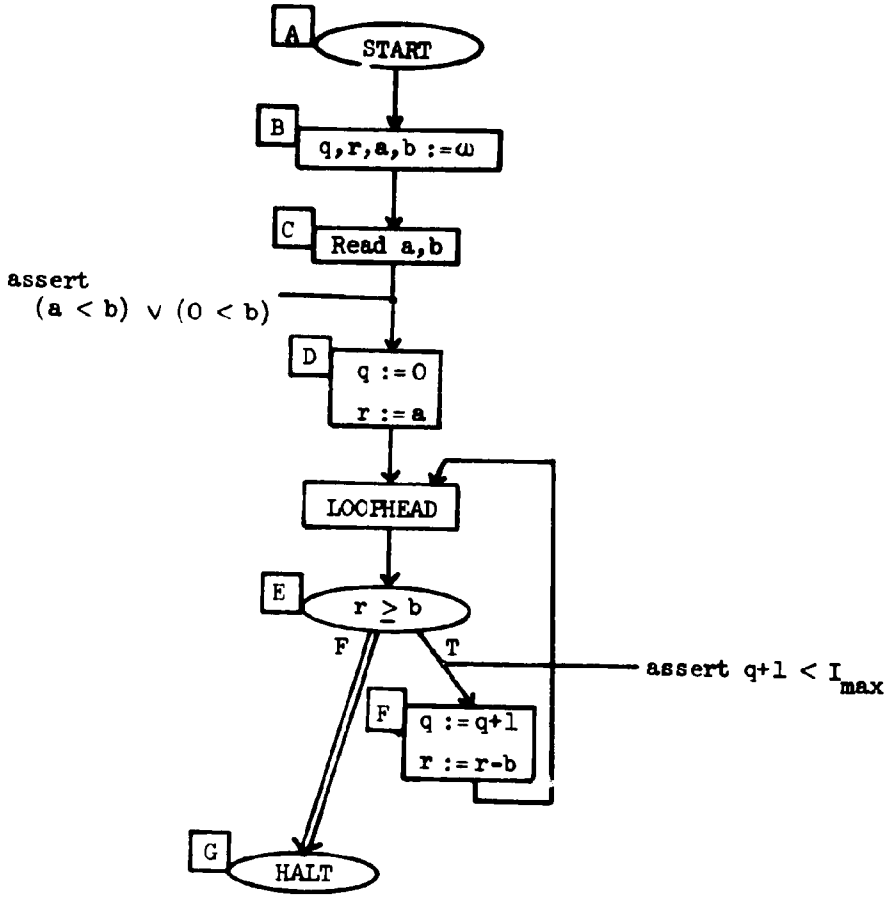

Figure A2.2. Final result of analysis of flow graph in Figure A2.1: the loop terminates iff the restriction on a and b is true after the Read ; the system is not powerful enough to prove that q will never overflow. Note that the synthesized restriction $(a < b) \vee (0 < b)$ allows some cases (such as a negative and b positive) that King's assumption $(a \geq 0) \wedge (b \geq 0)$ does not allow, and that our restriction excludes the infinite loop case $(a \geq 0) \wedge (b = 0)$.

Example 3. King's Example 3. Exponentiation.

In this example, the two interesting issues are the treatment of division in $y := y \div 2$, and the merging of information from the conditional assignment $z := z * x$.

In the analysis of integer division, we will use these axioms:

$$|p \div q| < |p| \quad \text{for} \quad |q| > 1 \text{ and } p \neq 0$$

$$|p \div q| = |p| \quad \text{for} \quad |q| = 1 \text{ or } (|q| > 1 \text{ and } p = 0)$$

$$|p \div q| = \text{undefined for } q = 0 .$$

Figure A3.1 shows the flow graph for this example with all the non-trivial assertions attached. As usual, the overflow assertions, $I_{min} \leq \text{expression} \leq I_{max}$, cannot be proved, and hence they represent definite problems for the user to consider.

The induction for the value of y at the LOOPHEAD node uses the initial value information

$$y := b$$

and the recurrence relation

$$y_{k+1} = y_k \div 2 \wedge y_k \neq 0 .$$

From $y_k \neq 0$ and $|2| > 1$, we can use the first axiom to conclude that

$$|y_{k+1}| < |y_k|$$

and hence that y is a subset of the range $-b$ to b:

$$y \subset \{-|b|, -|b|+1, \ldots |b|\} .$$

To prove the loop termination assertion,

$$\exists k \geq 1 \text{ s.t. } y_k = 0 ,$$

we can use the fact that the absolute value of y is strictly monotonically decreasing, and hence will eventually equal zero. Thus,

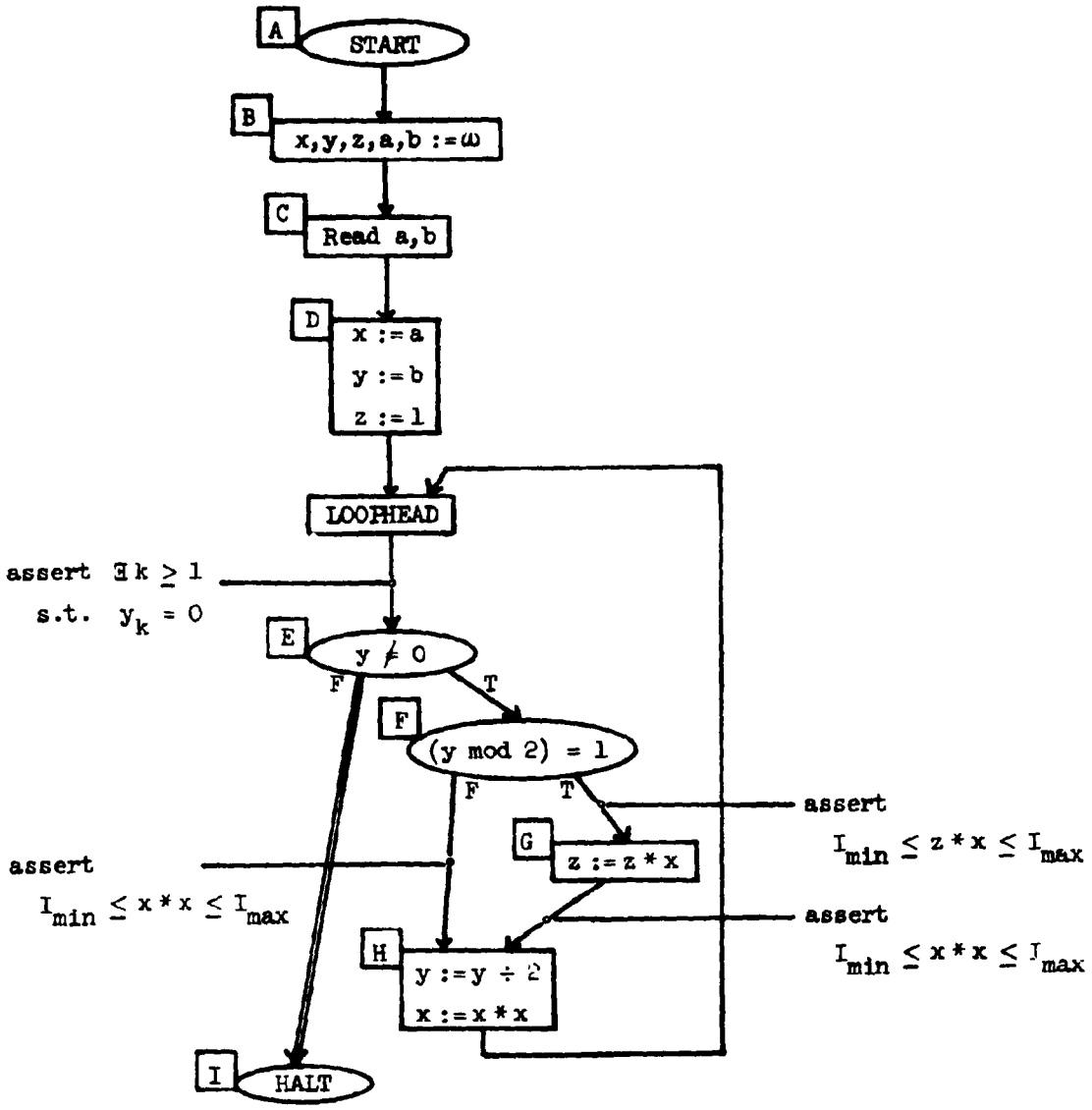

Figure A3.1. Flow graph for King's third example, with all mechanically generated assertions except those of the form $v \neq \omega$, for any variable v. We can prove that the loop terminates, but cannot prove the absence of overflows in nodes G and H.

we can prove that for all values of a and b , positive, zero, or negative, the loop terminates. Of course, if b is negative, the program doesn't compute a^b , but our proof of clean termination (if no overflows occur) can be combined with King's proof of correctness under the restriction that $b \geq 0$ to prove the total correctness of this program.

A second issue in the analysis of this program is the merging of information required at node H . On the first pass through the loop, the given information on the two entry arcs for node H includes

$$\text{arc F - H: } z_{k+1} = z_k \qquad \text{arc G - H: } z_{k+1} = z_k * x_k \ .$$

As discussed in Chapter 4, this information is merged to form the disjunct

$$z_{k+1} = z_k \ \vee \ z_{k+1} = z_k * x_k \qquad \text{(Refinement)},$$

ignoring the interaction between $y \bmod 2$ and z , but marking the disjunct "refinement exists", so that if necessary in a subsequent proof, the complete interaction can be reconstructed:

$$((y_k \bmod 2) = 1 \ \wedge \ z_{k+1} = z_k * v_k) \ \vee \ ((y_k \bmod 2) \neq 1 \ \wedge \ z_{k+1} = z_k) \ .$$

In this particular example, information about z is not needed to prove clean termination. If we were just interested in loop termination, all variables which have no effect on branching could be stripped out of the program early in its analysis.

Example 4. King's Example 4. Primality.

In this example, we encounter multiple exit tests and a proof of no overflow based on the fact that a defined variable has a representable value. Also, the proof of termination has absolutely nothing to do with what the program does.

The loop induction information for i includes the initial value

$$i = 2$$

and the recurrence relation

$$i_{k+1} = i_k + 1 \quad ,$$

so the values of i are an initial subset of $\{2,3,4,\ldots\}$, and are strictly monotonically increasing.

To prove the loop termination assertion,

$$\exists k \geq 1 \text{ s.t. } (i_k \geq a_k) \vee ((i_k < a_k) \wedge (a_k \bmod i_k = 0)) \quad ,$$

we try the simpler clause first. We find that a is invariant in the loop and that i is strictly increasing, so

$$i_k \geq a$$

will eventually be true and we have proved the loop termination without examining the second clause. Note that the loop terminates even if $a < 2$.

For the overflow assertion, we need to prove that

$$(i < a \wedge a \neq \omega \wedge a \bmod i \neq 0) \supset (i+1 \leq I_{max}) \quad .$$

Since $a \neq \omega$ means that a has some representable value, $I_{min} \leq a \leq I_{max}$, it follows that

$$(i < a \wedge a \leq I_{max}) \supset (i+1 \leq a \leq I_{max}) \supset (i+1 \leq I_{max}) \quad .$$

Example 4. Primality

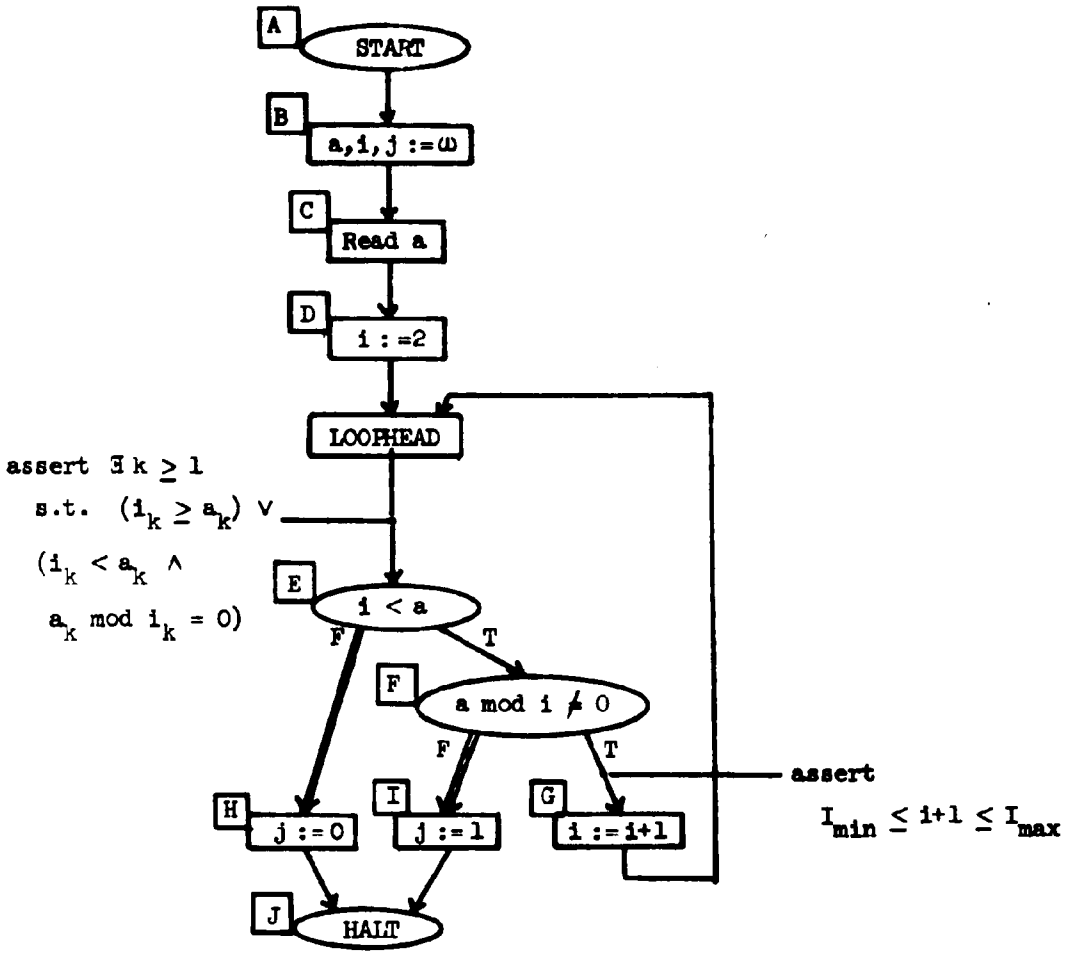

Figure A4.1. Flow graph for King's fourth example, with only the non-trivial assertions attached. There are two loop exit arcs, so we have a complex loop termination assertion (whose first clause is true). The overflow assertion $I_{min} \leq i+1 \leq I_{max}$ is true because $(i < a) \supset (i+1 \leq a) \supset (i+1 \leq I_{max})$, since whatever value is stored in a is representable, and hence $a \leq I_{max}$.

Example 5. King's Example 5. Zeroing.

Arrays are introduced in this example, presenting some new complications in describing the intended range of value subscripts, and in synthesizing an appropriate description of the values stored in the array.

As indicated by the annotations on the flow graph, Figure A5.1, the bounds for the array A must be supplied. In some programming languages (like Fortran), declarations of bounds are required for all arrays. For such languages, it is easy to insert the needed annotation mechanically. In other programming languages (like Algol 60), declarations of bounds are not required for arrays which are parameters of subprograms. For such languages, the human user must supply the needed annotation. In either case, the semantic assertion routine then uses these bounds to create subscript range assertions, like $1 \leq i \leq n_0$.

The loop induction step uses the following information:

initial values: $A = \omega$ (the whole array)

$i = 1$

recurrence relations: $i_{k+1} = i_k + 1$

$n_{k+1} = n_k$

$A_{k+1}[i_k] = 0$

$\forall \ell \neq i_k \ , \ A_{k+1}[\ell] = A_k[\ell]$

$i_k \leq n_k$.

From this information, we can deduce that:

Example 5. Zeroing

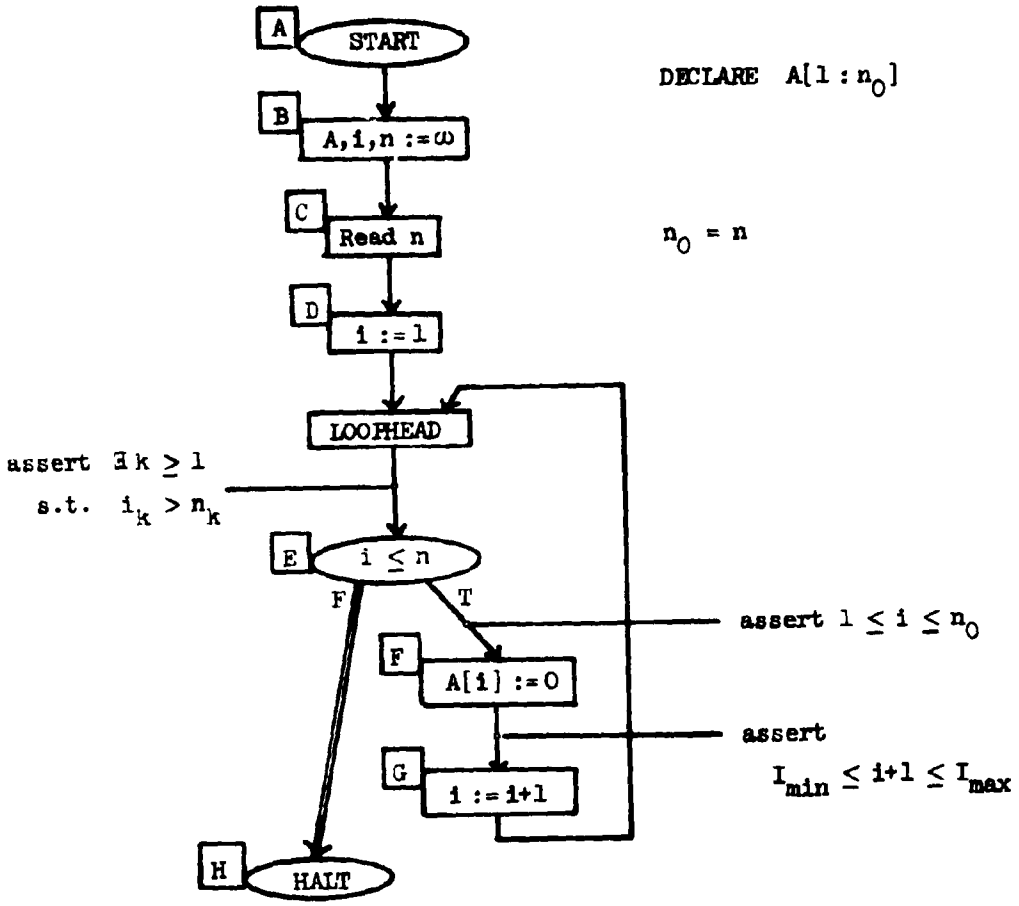

Figure A5.1. Flow graph for King's Example 5, a program to zero out an array. The declaration of A is an annotation added by the user, signifying that the valid bounds on A are 1 to n_0, where n_0 is the value read in for n. This binding of n_0 is to allow for the possibility that the value of n changes during execution.

Example 5. Zeroing

n is invariant

$i \rightarrow_c \{1,2,3 \ldots\} \wedge i \leq n+1$

so $i = \{1\} \cup \{2,3,4,\ldots,n+1\}$

where the second set is empty if $n \leq 0$

$\forall \ell \in \{1,2,3,\ldots,n\}$, $A[\ell] = 0$.

The deduction about n is straightforward. The deductions about i need some careful attention to detail: the $i \rightarrow_c \ldots$ notation implies that the actual set of values for i contains at least one element. Now, if $n < 0$, the set $\{1,2,3,\ldots,n+1\}$ can be strictly construed as the empty set, so to properly reflect the fact that i always has its initial value at the LOOPHEAD node, we adopt the union of sets notation. The set $\{2,3,4,\ldots,n+1\}$ reflects all the subsequent values of i , it is properly empty if $n \leq 0$ (and hence the loop is never traversed), and $n+1$ is in fact an element of the set $\{2,3,4,\ldots\}$ if $n \geq 1$. (If the step size for i were not one, but c , we would have to entertain the third possibility that $n+1$ is not in the set $\{1, 1+c, 1+2c, \ldots\}$ at all.)

In subsequent processing, we can easily prove the loop termination assertion,

$$\exists k \geq 1 \quad \text{s.t.} \quad i_k > n$$

since i is strictly increasing. The subscript range assertion,

$$1 \leq i \leq n_0$$

is true because n is invariant, hence equal to n_0 , $1 \leq i \leq n+1$ at the LOOPHEAD, and $1 \leq i \leq n$ on the true branch from node E . The overflow assertion,

$$I_{min} \leq i+1 \leq I_{max}$$

cannot be proved, and the program in fact generates an overflow if $n = I_{max}$. The user is asked if this value of n is possible.

Example 6. King's Example 6. Maximum.

This program to find the largest element of an array by successive interchanges shows how the recurrence relations express an interchange as a simultaneous assignment to two elements of the array, how aliases are handled, and how lemmas can be discovered. The flow graph for the program is in Figure A6.1.

In Figure A6.2, we show the given information gathered on the first pass through the loop. Most of this processing is straightforward, but there are some complications after node I . In reflecting the assignment A[i-1] := x , we must check for aliases (as explained in Chapter 4) to see if that assignment changes an element of A that we also know under some other name in our set of given information. In this example, $A_{k+1}[i_k]$ is referred to in the given information on the entry arc for node I , so we try to prove the two theorems

$$i_k = i_k\text{-}1$$

and

$$i_k \neq i_k\text{-}1 \quad .$$

If the first is true, then $A_{k+1}[i_k]$ is an alias for $A_{k+1}[i_k\text{-}1]$ and both would be equal to x on exit from node I . If the second is true, then the assignment to $A_{k+1}[i_k\text{-}1]$ cannot affect the value of $A_{k+1}[i_k]$, so there is no alias problem. In the current example, of course, $i_k \neq i_k\text{-}1$, so there is no alias problem. In the more general case of successive assignments to A[i] and A[j] , we try to prove that either $i_k = j_k$ or $i_k \neq j_k$. If we cannot prove either theorem, then we must allow for both possibilities:

$$(i_k = j_k \wedge A_{k+1}[i_k] = A_{k+1}[j_k]) \vee (i_k \neq j_k \wedge A_{k+1}[i_k] = \text{old value}).$$

Example 6. Maximum

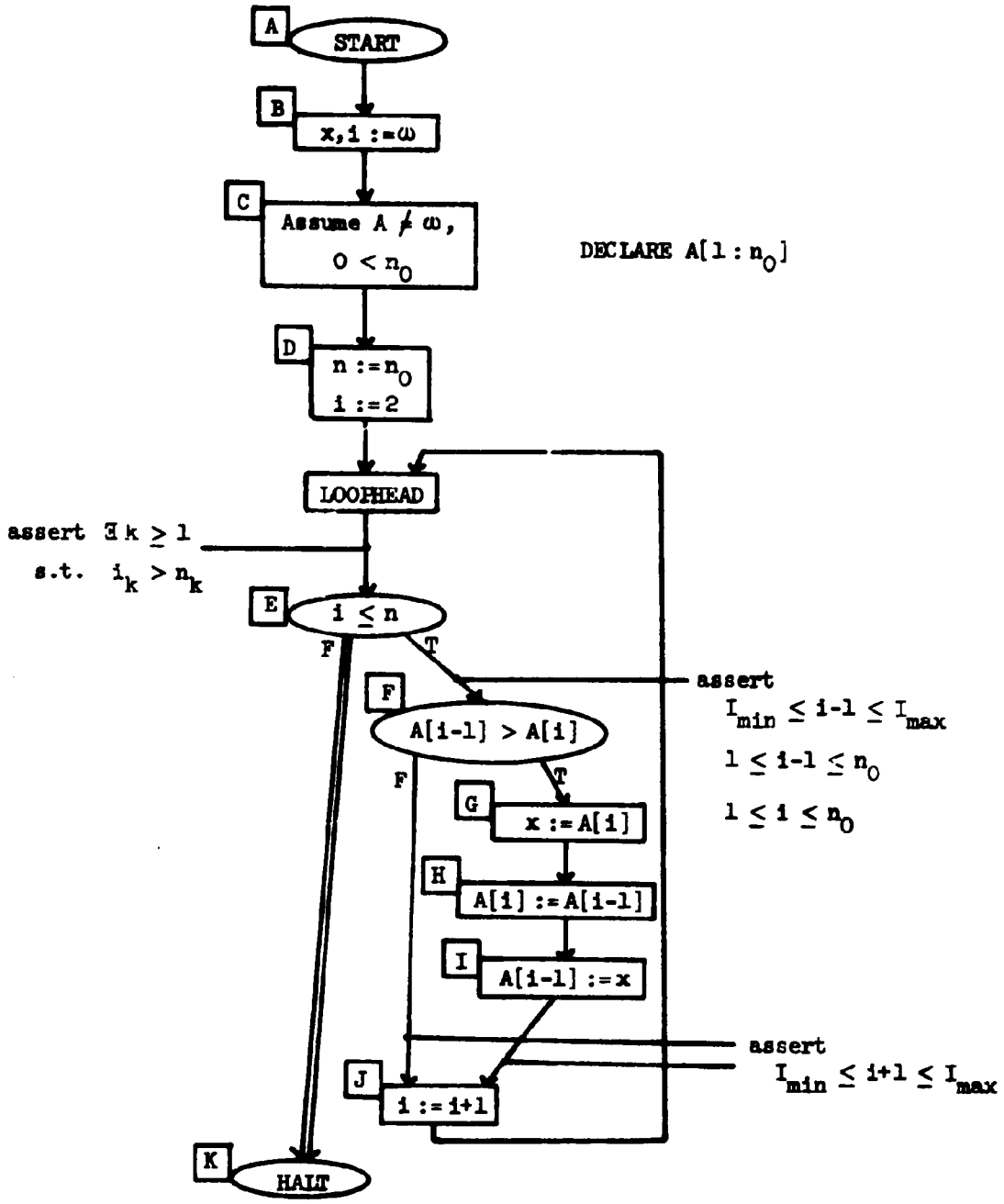

Figure A6.1. Flow graph for King's Example 6, with explicit assumptions about A and n in nodes C and D, and with an annotation describing the subscript bounds for A. Only the significant assertions are shown.

Example 6. Maximum

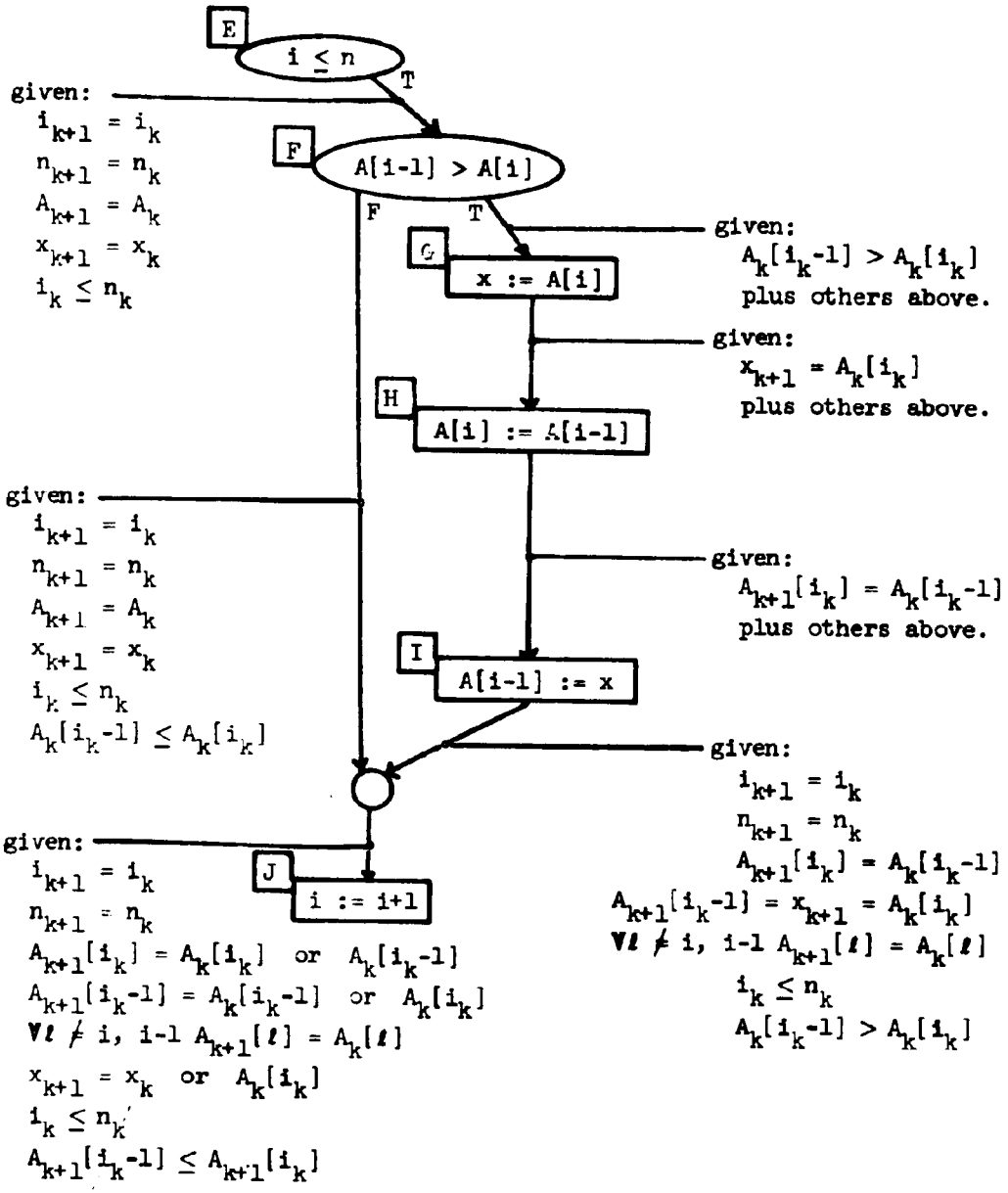

Figure A6.2. Gathering of given information on the first pass through the body of the loop. All deductions are straightforward, except for the last merged one on the arc leading to node J , $A_{k+1}[i_k-1] \le A_{k+1}[i_k]$. The text explains the derivation of this lemma. The dummy node just before J was inserted for clarity of expression.

We may then weaken this expression to avoid the cross-product terms relating i to A :

$$A_{k+1}[i_k] = x_{k+1} \quad \text{or} \quad A_{k+1}[i_k] = \text{old value} \quad \text{(Refinement)} .$$

If it later becomes necessary to use the exact relationship, it can be reconstructed.

Jim King discusses the alias issue on pages 77-82 of his thesis [King 1969], and again on pages 132-140. In the latter section, he discusses the problem of working backwards through a program, generating expressions for all possibilities of subscript aliasing. A series of four assignments can easily generate an expression containing 16 different cases.

The problem with working backward through a program is that, for a sequence like

```
i := j+2;
A[i] := 3;
A[j] := 4;
```

there is no information about the relationship between i and j when the assignment to A[i] is processed. Thus, King must generate an expression like

(i = j ∧ A[i] is changed by both assignments) ∨

(i ≠ j ∧ A[i] is changed only by the first) ,

and later try to decide which case applies. By working forward, our system has seen the assignment

```
i := j+2
```

before processing the array assignments, so enough information is available for the theorem prover to be called to answer the alias question:

Example 6. Maximum

$$(i = j+2) \supset (i \neq j) \supset (A[i] = 3 \wedge A[j] = 4) \quad .$$

There are two problems in working forward. (1) The process is not goal-directed; in contrast to working backward, there is no definite assertion or verification condition to be proved, so it is possible either to discard crucial information or to retain useless verbiage. (2) The information required to prove theorems such as $i \neq j$ inside a loop may depend on assignments near the bottom of the loop, making it impossible to prove the theorem on a single forward pass.

The techniques presented in Chapter 4 try to mitigate these two problems by (1) using a set of heuristics to merge information into "useful" lemmas, while still retaining access to the unmerged (refinement) information in case it is crucial to a later proof, and (2) processing loops in two passes, where sometimes an alias theorem can be proved on the first pass because it is true independently of subsequent assignments, and sometimes an alias theorem can be proved only on the second pass, after ranges of values for program variables and the invariant relationships between them have been determined.

After that somewhat lengthy discussion, we return to our example and the merging of <u>given</u> information at the dummy node in Figure A6.2. The left arc includes the information:

$$A_k[i_k-1] \leq A_k[i_k] \quad \text{and}$$

$$A_{k+1} = A_k \quad .$$

The right arc includes the information

$$A_k[i_k-1] > A_k[i_k] \quad \text{and}$$

$$A_{k+1}[i_k-1] = A_k[i_k] \quad \text{and}$$

$$A_{k+1}[i_k] = A_k[i_k-1] \quad .$$

In merging this information, we try to find an expression which is implied by the information on both arcs. We start with the expressions that already attached to the incoming arcs, trying unsuccessfully to prove that:

right arc info $\supset$ left arc info

or left arc info $\supset$ right arc info .

This strategy works in merging, say, $i_{k+1} = i_k$ at the dummy node, but fails to produce any common information about A . If there is no common information in the relationships between the old values (subscript k) of variables, perhaps there is some common relationship between the new values (subscript k+1); perhaps the whole point of the separate paths which are now merging was to create some useful relationship between the new values of variables.

To discover useful lemmas about the relationship between the new values of A , we modify any old relationships on each path to reflect the assignments on that path, giving

$$A_{k+1}[i_k-1] \leq A_{k+1}[i_k]$$

on the left arc, since $A_{k+1} = A_k$ on that arc, and giving

$$A_{k+1}[i_k] > A_{k+1}[i_k-1]$$

on the right arc, since $A_{k+1}[i_k] = A_k[i_k-1]$ and $A_{k+1}[i_k-1] = A_k[i_k]$.

We again try to find an expression which is implied by the information on both arcs:

left arc info $\supset$ right arc info

i.e., $A_{k+1}[i_k-1] \leq A_{k+1}[i_k] \not\supset A_{k+1}[i_k] > A_{k+1}[i_k-1]$.

Example 6. Maximum

The strictly greater than relation is not the weaker, so we try to prove:

right arc info $\supset$ left arc info

i.e., $A_{k+1}[i_k] > A_{k+1}[i_k\text{-}1] \supset A_{k+1}[i_k\text{-}1] \leq A_{k+1}[i_k]$.

This implication is true, so we have just discovered the lemma we are seeking: the inequality

$$A_{k+1}[i_k\text{-}1] \leq A_{k+1}[i_k]$$

is true on both arcs, so we attach it as part of the merged information on the entry arc to node J .

In our current example, this mechanically synthesized lemma is not needed to prove any of the assertions in the program, but a similar process is crucial in the loop termination proofs in SELECT [Sites 1974]. In fact, all the assertion proofs on the second pass through the loop in our current example are straightforward. The loop terminates because i is monotonically increasing. The subtraction i-1 does not overflow because $i \geq 2$, a fact which we could not know on the first pass through the loop, since it depends not only on the initialization at node D , but also on the assignment at node J . The subscripts are all in range, and the assignment at node J may in fact overflow. Note that the human user could remove the overflow problem by including in node C the assumption (restriction) that $n_0 < I_{max}$.

Example 7. King's Example 7. Bubble Sort.

This is the first example in which we cannot prove that the program terminates. It is also the first example in which we have two nested loops. Following the process in Chapter 1, we do the interval analysis of the flow graph, find the loops, and then try to put them in leading test form. Figure A7.1 shows the flow graph before this last transformation; the inner loop is in leading test form, but the outer is not. Figure A7.2 shows the change in structure required to put the outer loop in leading test form also. Then it is easy to synthesize the loop termination conditions:

$$\exists \ell \geq 1 \quad \text{s.t.} \quad j_\ell = 0 \qquad \text{for loop \#1,}$$

$$\exists k \geq 1 \quad \text{s.t.} \quad i_k > n_k \qquad \text{for loops \#2 and \#2'.}$$

Loops 2 and 2' are essentially the same as Example 6, and terminate for the same reason -- i is monotonically increasing. The rest of this discussion therefore centers on the behavior of j. Since loops 2 and 2' are identical, we shall concentrate on the nested pair, 1 and 2. The reader can fill in the details of the degenerate case of an initially completely sorted array, when loop 2' exits with $j = 0$ and hence loop 1 never iterates.

Figure A7.3 shows the details of the multiple passes over the nested loops to find out the range of values for j during all possible iterations. The assignment to j in node D turns the outer loop induction into a degenerate case: $j_{\ell+1}$ does not depend at all on j_ℓ, so the second outer pass contributes no new information after node D. Eventually however, we find that the range of values for j at node J is 0 or 1, and that there are no reasons

that j must sometimes equal 1 . Therefore, we cannot prove that the outer loop terminates. The human user will have to look at this loop and convince himself that the loop does in fact terminate (because humans "know" that eventually no interchanges will take place and therefore the assignment at node H will not be executed).

It is possible in this example to split up the inner loop so that if the interchange never takes place, the inner loop exits directly to node K , but that turns out not to help us prove loop termination, because we still cannot prove that eventually no interchange will occur.

Example 7. Bubble Sort

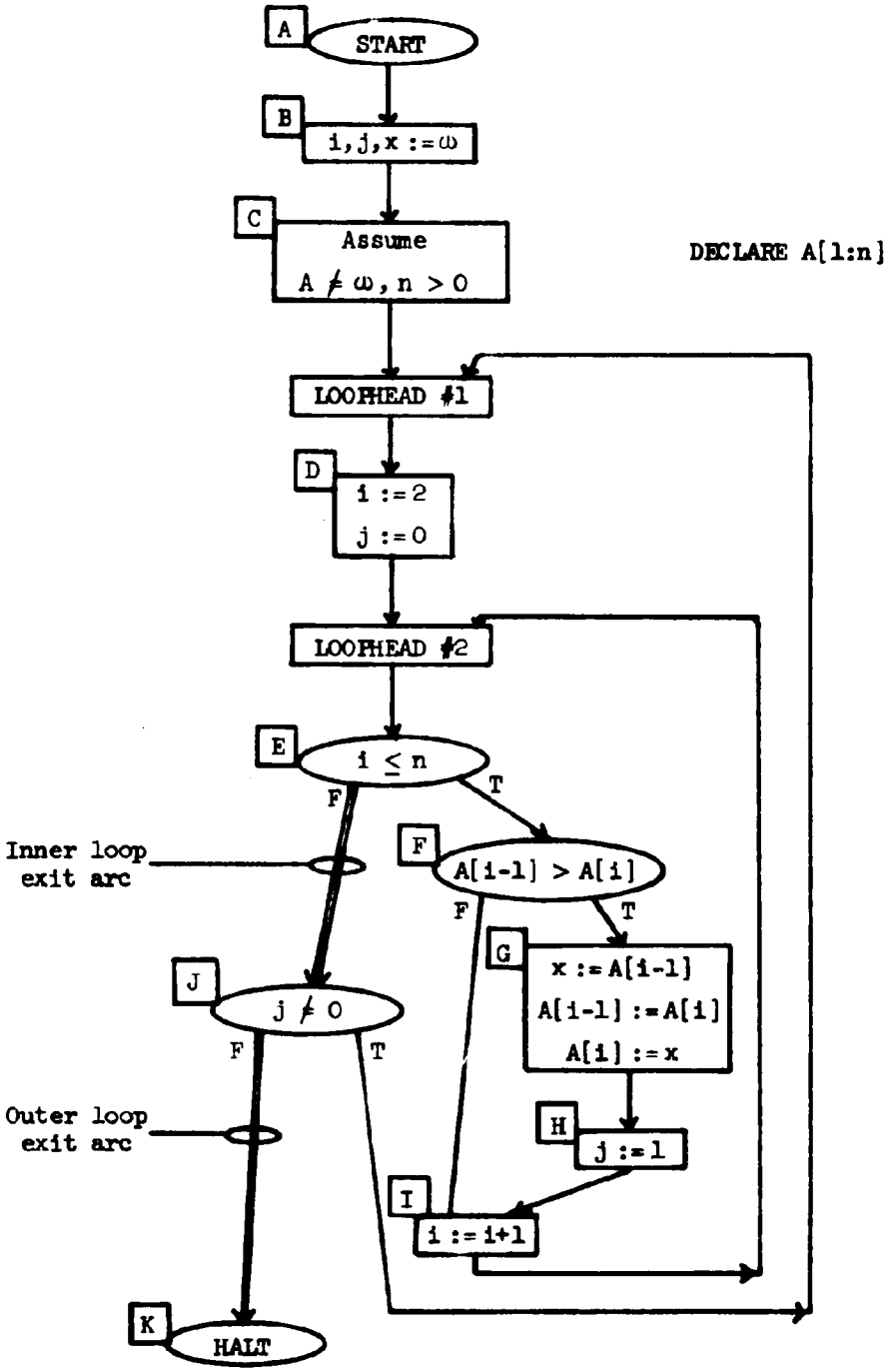

Figure A7.1. Flow graph for King's bubble sort. We will permute the nodes so that the loop exit node for the outer loop, J, is just after the LOOPHEAD #1 node. We will not be able to prove that the outer loop terminates, since its termination depends on no further interchanges taking place in the inner loop.

Example 7. Bubble Sort

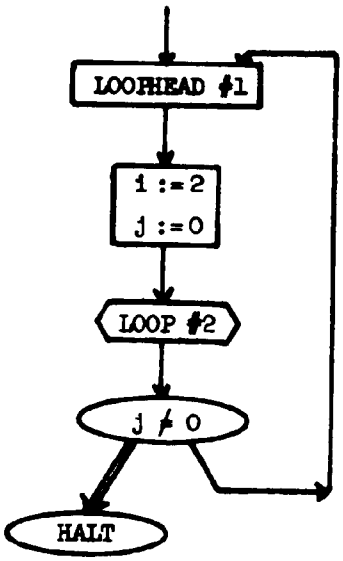

Figure A7.2a. Structure of the nested loops in Figure A7.1. The outer loop is not in leading test form, so we permute the nodes inside the loop until it is, as described in Chapter 1.

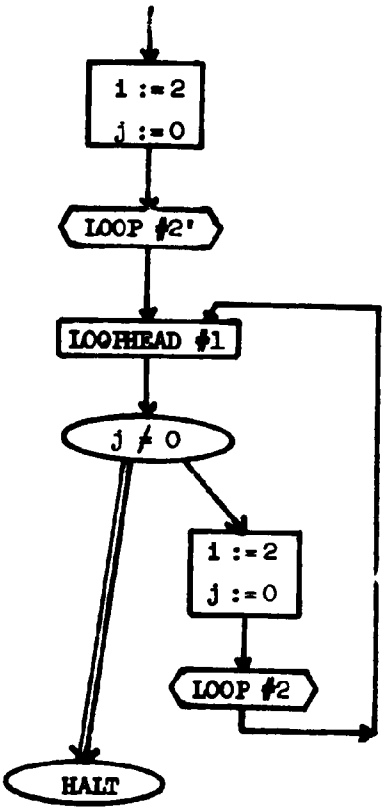

Figure A7.2b. Structure of the permuted nested loops from Figure A7.2a. We have made copies of the initial assignments to i and j, and of the entire inner loop.

Example 7. Bubble Sort

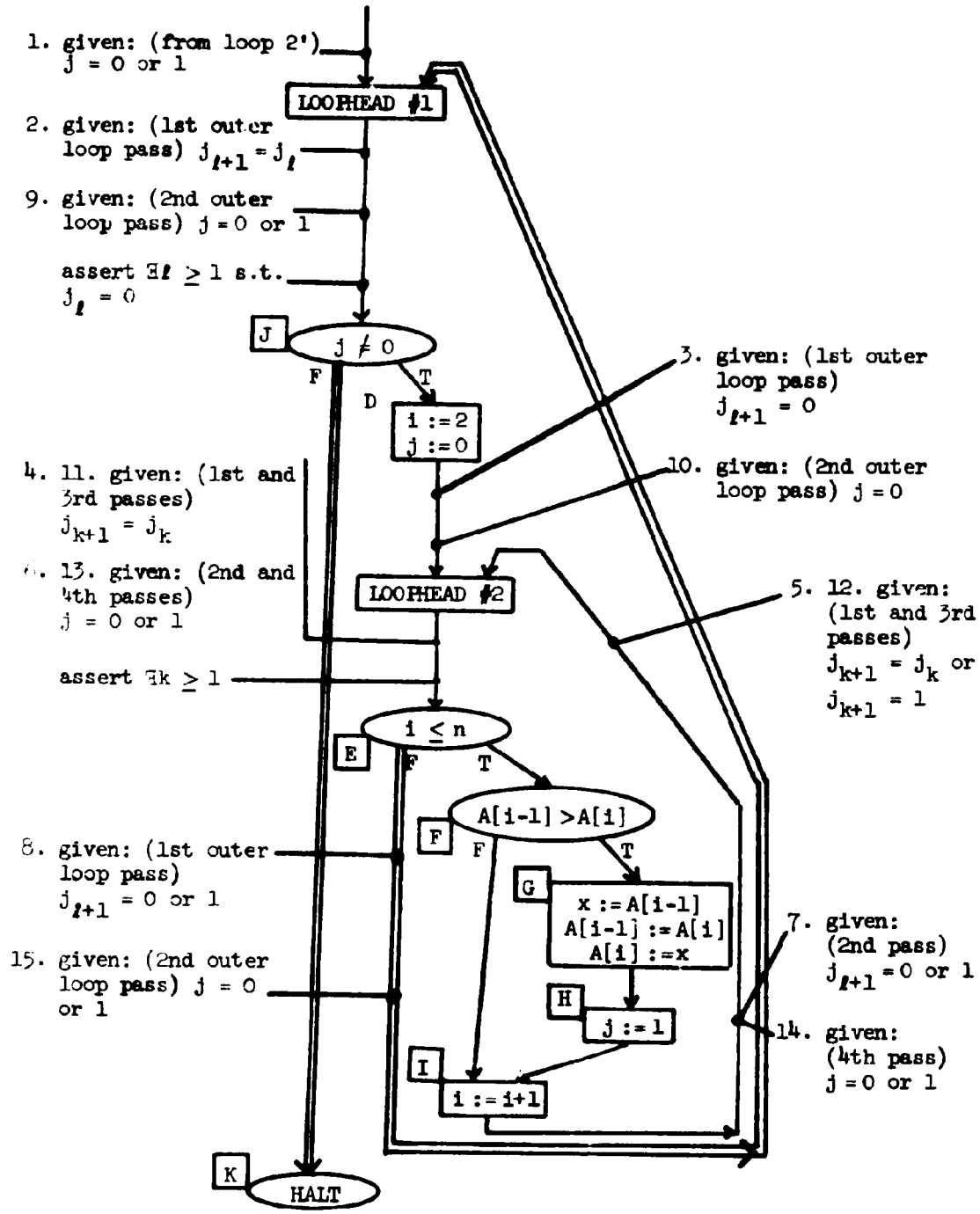

Figure A.7.3. Details of gathering information about j . We take two passes through the outer loop, first looking (continued next page)

Figure A7.3 (continued)

at the symbolic values $j_{\ell+1}$ and j_ℓ. On this first pass through the outer loop, we take two passes over the inner loop, first with j_{k+1} to find out that it can remain the same or becomes 1, and second with $j_{\ell+1} = 0$ as the initial value. After the first outer pass, we find that $j = 0$ or 1 for all iterations of the outer loop. During the second pass through the outer loop, we again traverse the inner loop twice (passes 3 and 4). Pass 3 is exactly identical to pass 1 and can clearly be implemented to take advantage of this; pass 4 uses $j = 0$ as the initial value, instead of the symbolic induction variable $j_{\ell+1}$ from the outer pass 1. In this example, passes 2 and 4 are identical, but only because of the assignment to j in node D. In general, pass 2 would have found less specific information.

Example 8. King's Example 8. Multiplication via increment/decrement.

In this example, we use a refinement of some merged information to restructure a loop into two simpler loops. This particular restructuring turns out to be a classical program optimization transformation of taking invariant tests out of loops.

The three loops in this example (Figure A8.1) all have the identical structure, so we will just consider the stripped-down version in Figure A8.2. As that loop is written, it either counts x down to zero if x is positive, or counts it up to zero if x is negative. It may be a good heuristic to say that if a loop termination test is a comparison for exactly zero, then look at the absolute value of the expression involved. Such a heuristic would allow us to prove that $|x|$ is monotonically decreasing and hence that the loop will terminate. However, by doing some node splitting to access a refinement of the merged information about x at the loophead node, we can restructure the program, as in Figures A8.3 and A8.4, into two loops, one for x positive, and one for x negative, then easily prove that each loop terminates, without using the absolute value heuristic.

Example 8. Multiplication

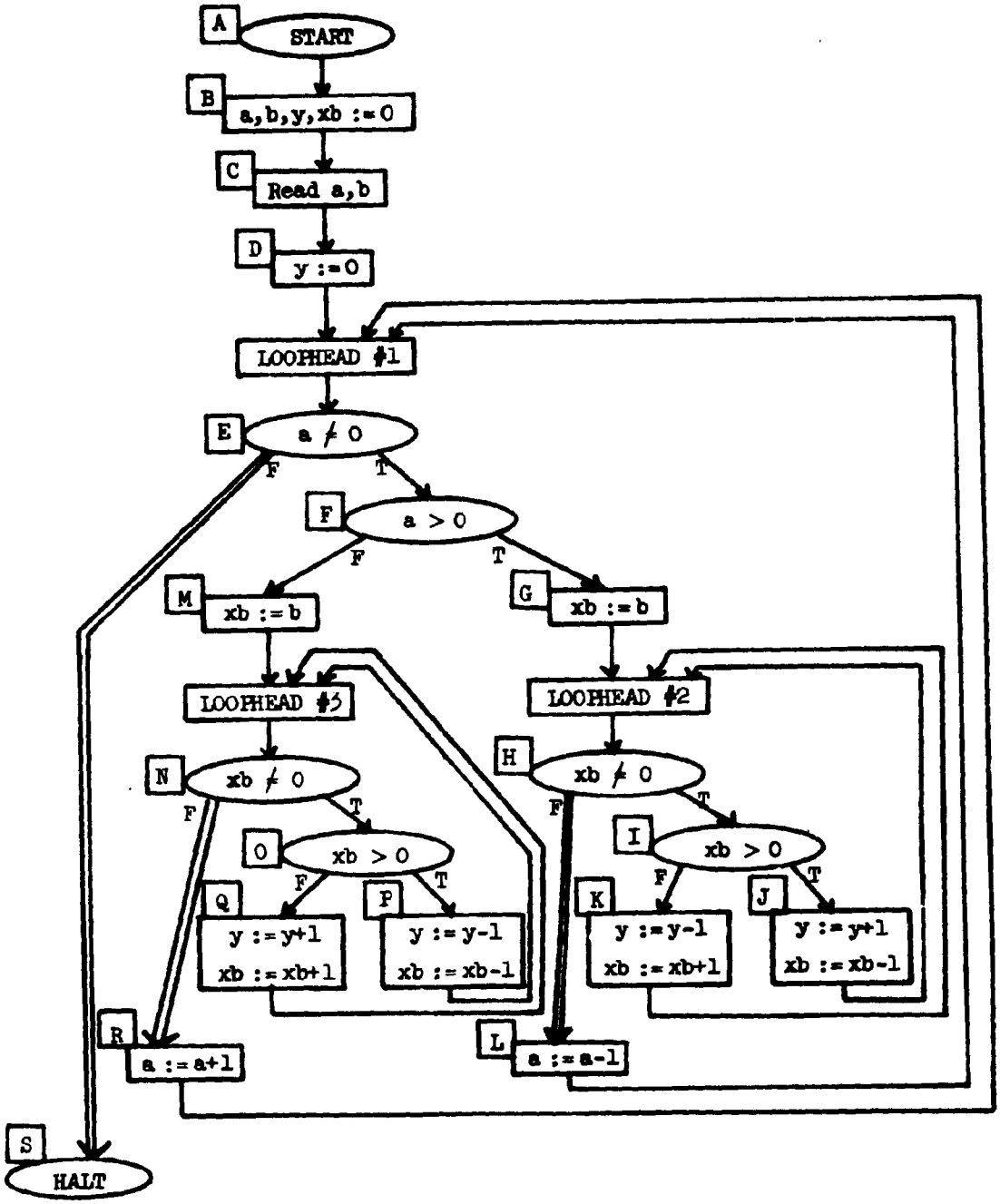

Figure A8.1. Flow graph of King's Example 8, which has three loops with identical structure and identical termination problems. We will break each loop into two; one to count a positive variable down to zero, and one to count a negative variable up to zero.

Example 8. Multiplication

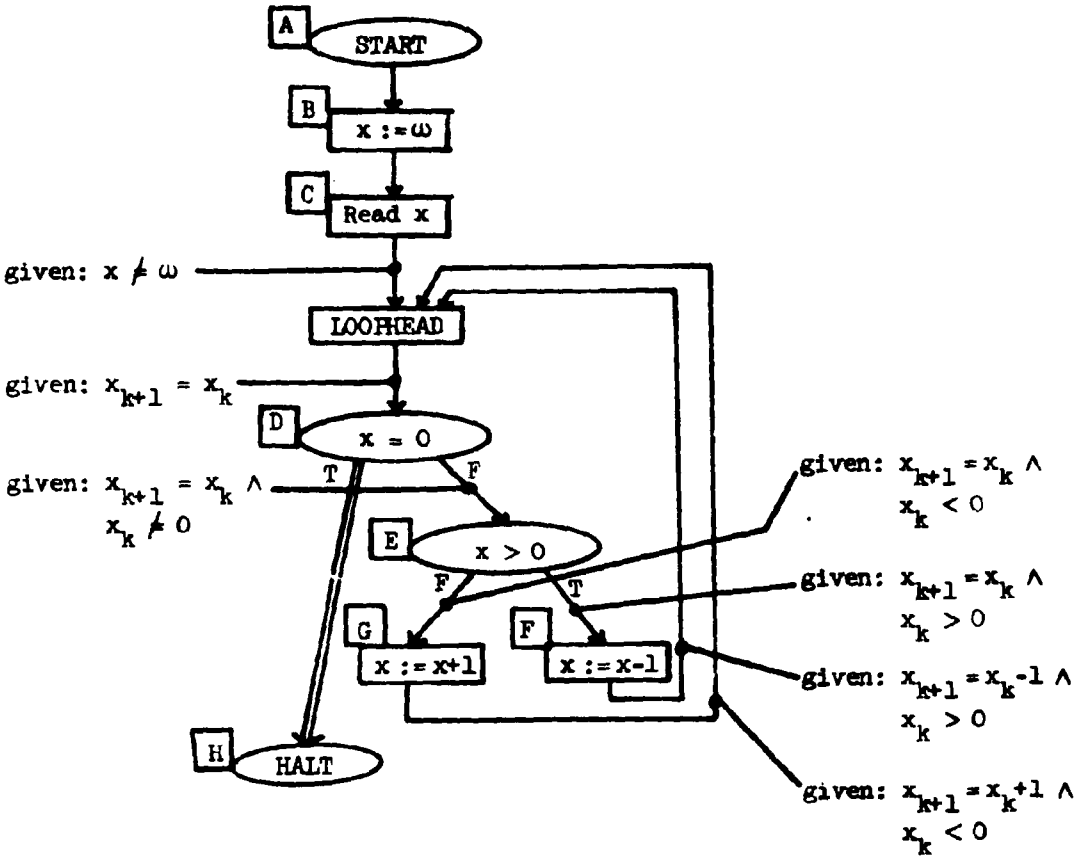

Figure A8.2. Essential structure of the loops in Figure A8.1, with given information from the first analysis pass attached. In merging the value information $x_{k+1} = x_k - 1$ and $x_{k+1} = x_k + 1$ at the LOOPHEAD node, we find only that $x_{k+1} \neq x_k$. In looking for common relationships between the new (subscript k+1) values of x, we find on one arc that $x_{k+1} \geq 0$, and on the other $x_{k+1} \leq 0$. The only common thing implied by these two expressions is that $x_{k+1} \neq \omega$. Thus, the range for x that we use on the second pass through the loop is

$$x \neq \omega \quad \text{(Refinement)} .$$

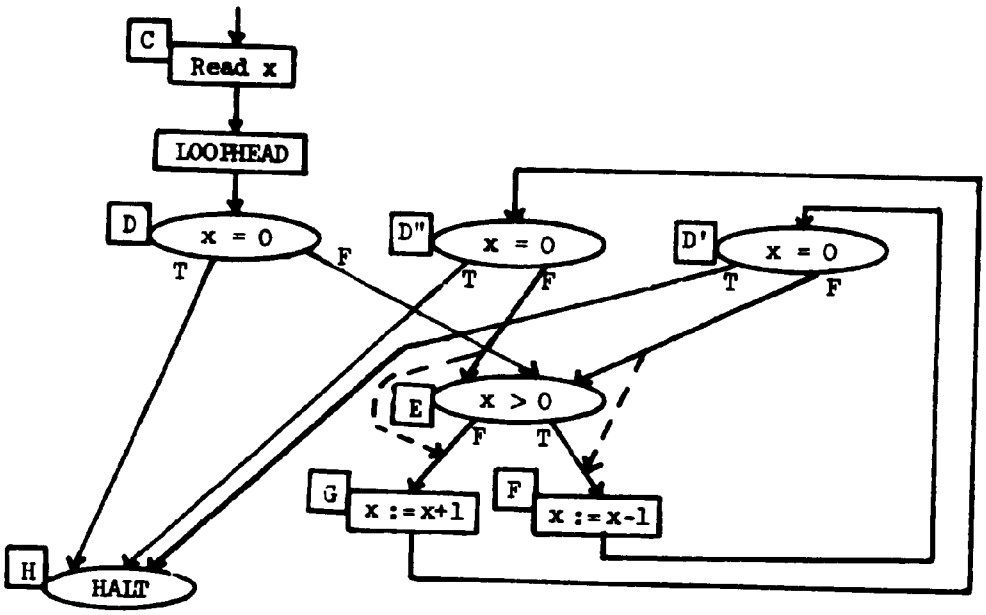

Figure A8.3. First step in restructuring loop in Figure A8.2 is to try to elide tests along some paths by proving that they are always true or always false on that path. The test at node D is inconclusive along all paths, so we cannot elide it. The test at node E, however, is always true along the path F - D - E, so we split out that path, making a copy of node D in the process, and then elide the test as shown by the dotted line. Similarly, the test at node E is always false along the path G - D - E, so we split out that path (making another copy of D) and elide the test. We must now re-analyse the loop structure of the program, starting at the LOOPHEAD node.

Example 8. Multiplication

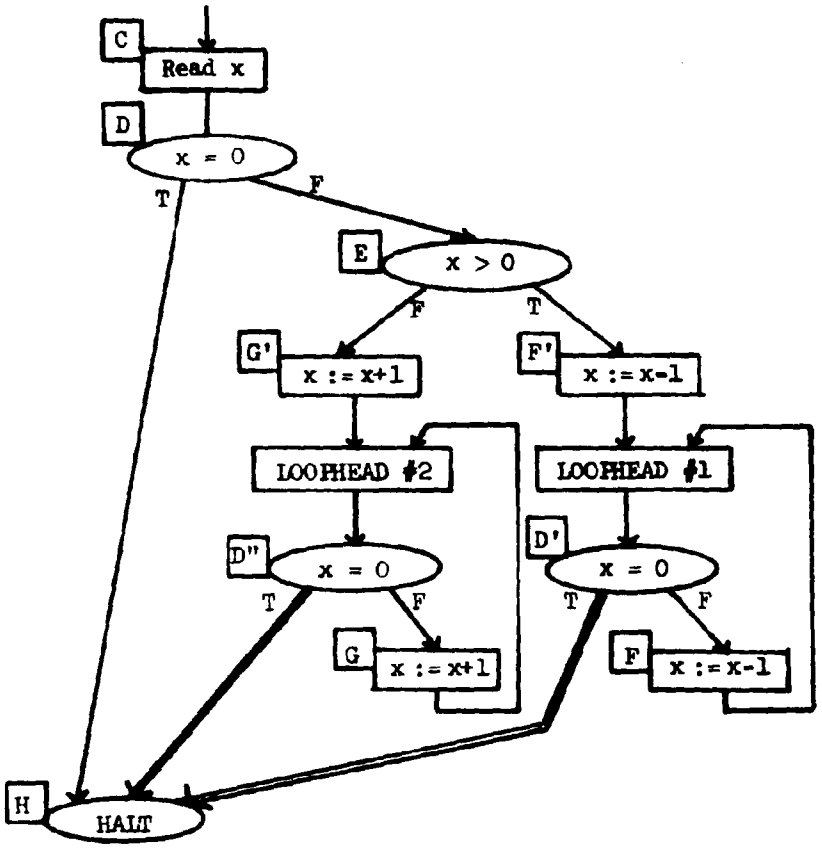

Figure A8.4. The loop in Figure A8.3 after re-analysing the loop structure and permuting the loops so that they have leading tests (thus forcing copies of nodes G and F). It is now fairly easy to prove that each loop terminates, without resorting to any arguments about absolute value. Note also that a careful programmer could have written the original program in this two-loop form, in order to avoid the redundant test of $x > 0$ inside the loop.

Example 9. King's Example 9. Selection Sort.

In this example (Figure A9.1), there is very little overlap between the information gathered to prove that the program terminates cleanly and the information gathered by King to (attempt to) prove that the program correctly sorts an array. The two nested loops have the same structure seen in earlier examples and it is easy to prove that they terminate, that all the variables are defined on use, and that no overflows occur because we anticipated the problem and assumed that the size of the array is less than I_{max}. The only difficulty is proving that the subscripts are in range in nodes H and K, so we will examine the information gathered about i, j, k, and n more closely. To avoid confusion in the notation, p and q are used as iteration subscripts in the recurrence relations for loop #1 and loop #2 respectively.

Figure A9.2 shows the first few steps in collecting information about i, j, k, and n: symbolic names (subscript p and p+1) are used to develop recurrence relations about how the values of variables change once around the loop. In the midst of this first pass outer loop processing, two passes are made through the inner loop, as shown in Figure A9.3. The results of A9.3 are passed as given information to node K in the first outer loop pass. This nested processing allows us to discover, for example, that n is invariant in the inner loop, and hence to discover a little later that n is invariant in the outer loop.

After the first pass through the outer loop, we use the recurrence relations gathered (attached after node L) and the initial values

Example 9. Selection Sort

$i = 1$, $j = \omega$, $k = \omega$, $n \neq \omega$

to synthesize a range of values for each variable at the LOOPHEAD #1 node during all iterations of the outer loop:

n is invariant and $n \neq \omega$

$i = \{1\} \cup \{2,3,\ldots,n\}$

$j = \omega$ or $j > n$

$k = \omega$ or $k \subset \{1,2,\ldots,n\}$.

We then take a second pass through the outer loop, using these ranges to prove assertions. At nodes F and K , it is now clear that i is in the proper subscript range: $1 \leq i \leq n$. When we encounter the inner loop, we use the new initial value information (as it stands on exit from node F) with the old inner recurrence relations (subscripts q and $q+1$) to synthesize a tighter set of ranges for variables inside the inner loop. In this example, the ranges attached to the exit arc of the LOOPHEAD #2 node are:

n is invariant and $n \neq \omega$

$i = \{1,2,\ldots,n-1\}$

$j = \{i+1, i+2, \ldots, n+1\}$

$k \subset \{i, i+1, \ldots, n\}$.

Following the test in node G , we can prove that j is in the proper subscript range in nodes H and I : $1 \leq j \leq n$. On this third pass through LOOP #2, we can also prove that $j = n+1$ on exit to node K . On previous passes, we did not know anything about the relationship between j and n , so we had to allow for an initial case like $j = 342$ and $n = 12$, in which we could only state that $j > n$ on exit, not that $j = n+1$. However, now we know that the

maximum initial value of j is n, hence the inner loop always iterates at least once and $j = n+1$ on exit. (Note that our analysis system would actually use the fact that $j \leq n$ initially at LOOPHEAD #2 to elide the test in node G for the first iteration of the loop, forcing a complete copy of the nodes in the loop to be used to reflect the unconditional first iteration. We will ignore this complication.)

Following the third pass through LOOP #2, we arrive at node K with the following <u>given</u> information:

$$n \neq \omega$$

$$i = \{1,2,\ldots,n-1\}$$

$$j = n+1$$

$$k \subset \{i, i+1, \ldots, n\} \quad .$$

This is sufficient to prove that the subscripts i and k are always in the proper range $(1 \leq i \leq n, \ 1 \leq k \leq n)$ at node K. We have thus proved, through a somewhat tedious process, that all subscripts are in range in this program, during all iterations of both the inner and outer loops.

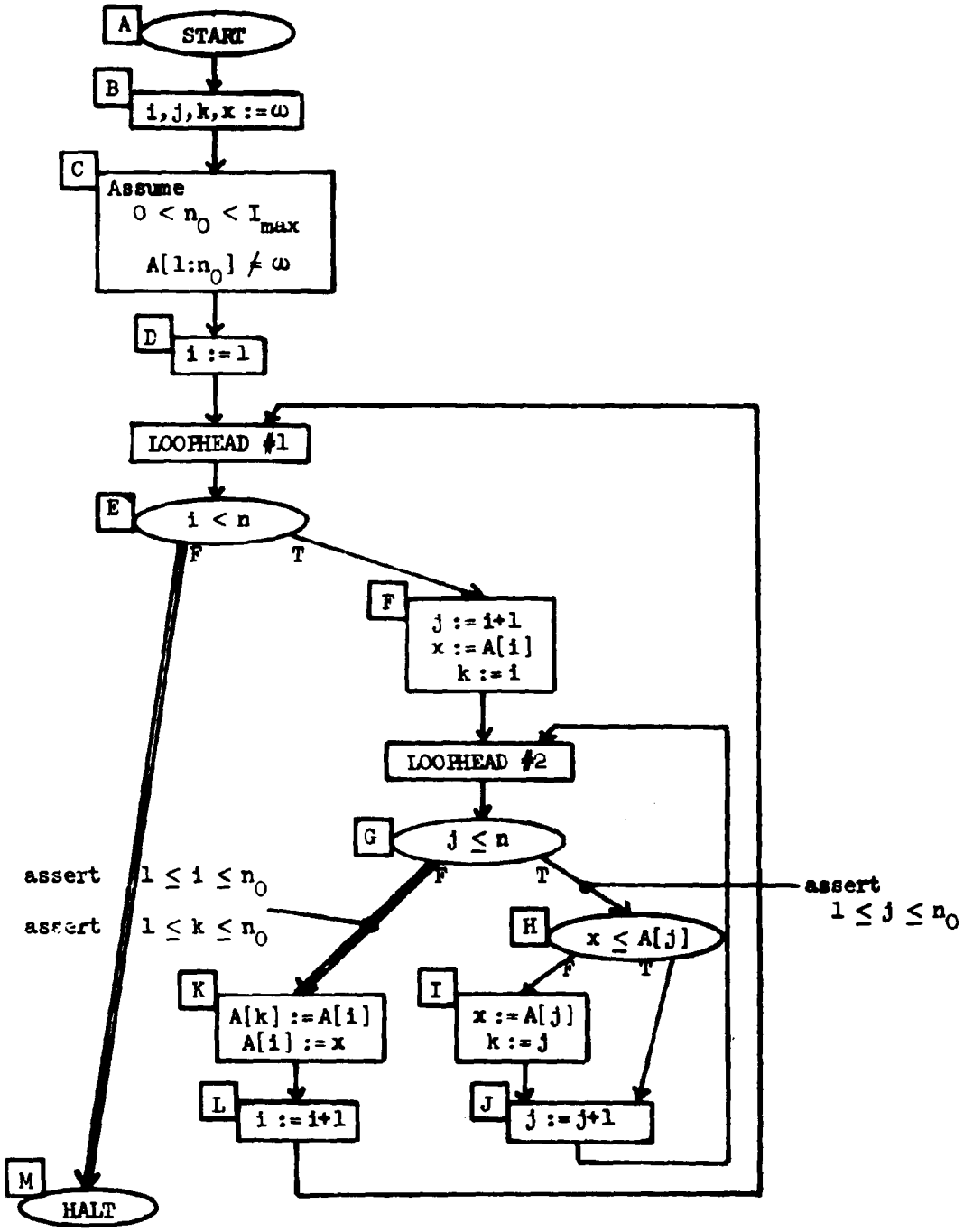

Figure A9.1. Flow graph for King's last example. We will only consider the proofs of the three assertions shown, since the other proofs are similar to those in earlier examples. To prove clean termination, we need never consider what is happening to A , i.e., that it is being sorted.

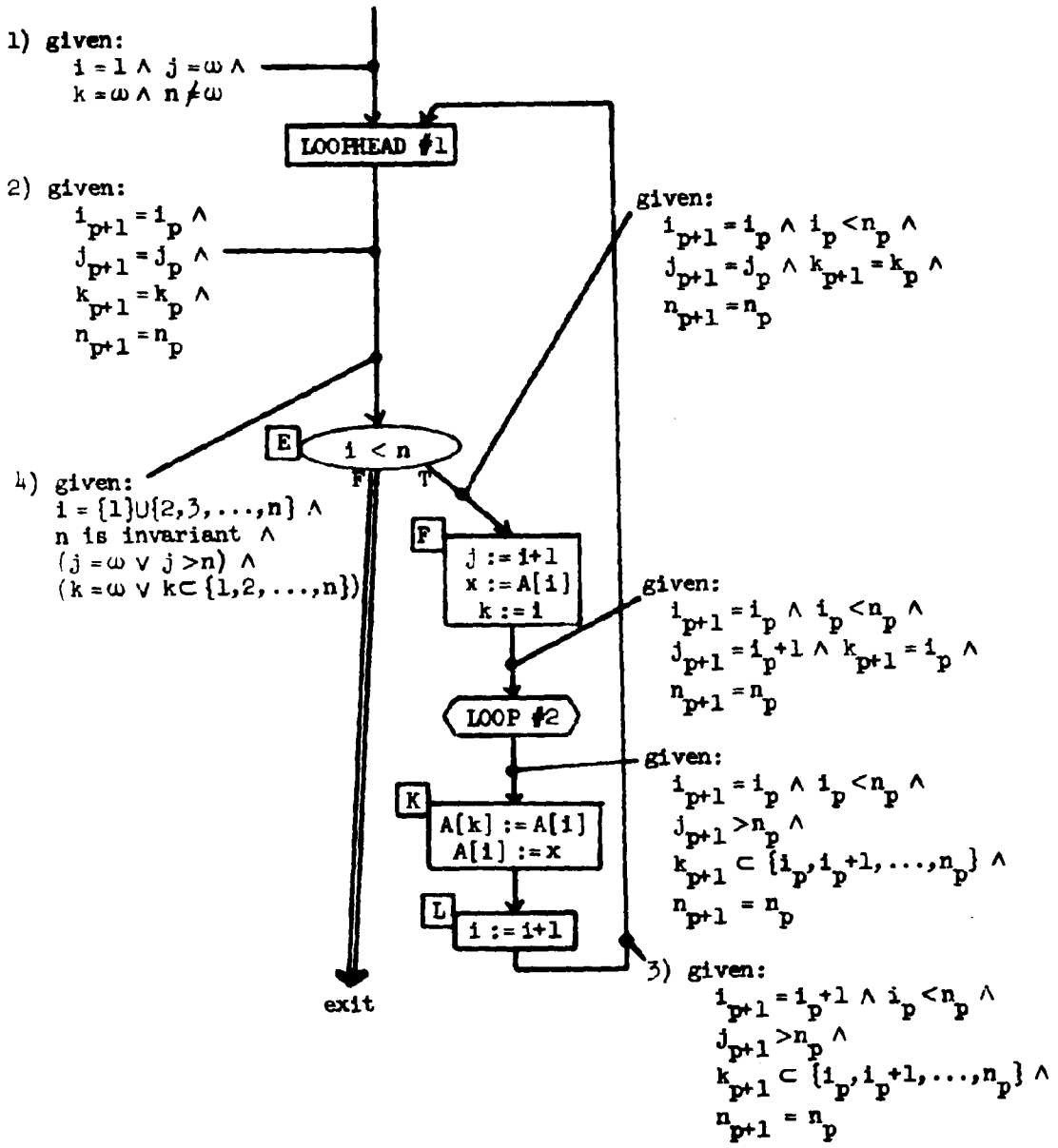

Figure A9.2. Gathering of given information for i , j , k , and n on first pass through outer loop. The processing of the inner LOOP #2 on this first pass is detailed in Figure A9.3. The induction step between the first pass and the second pass through the outer loop determines the synthesized information labeled 4) . Note that j does not have a particularly useful value at the LOOPHEAD #1 node; it is the assignment in node F that is important.

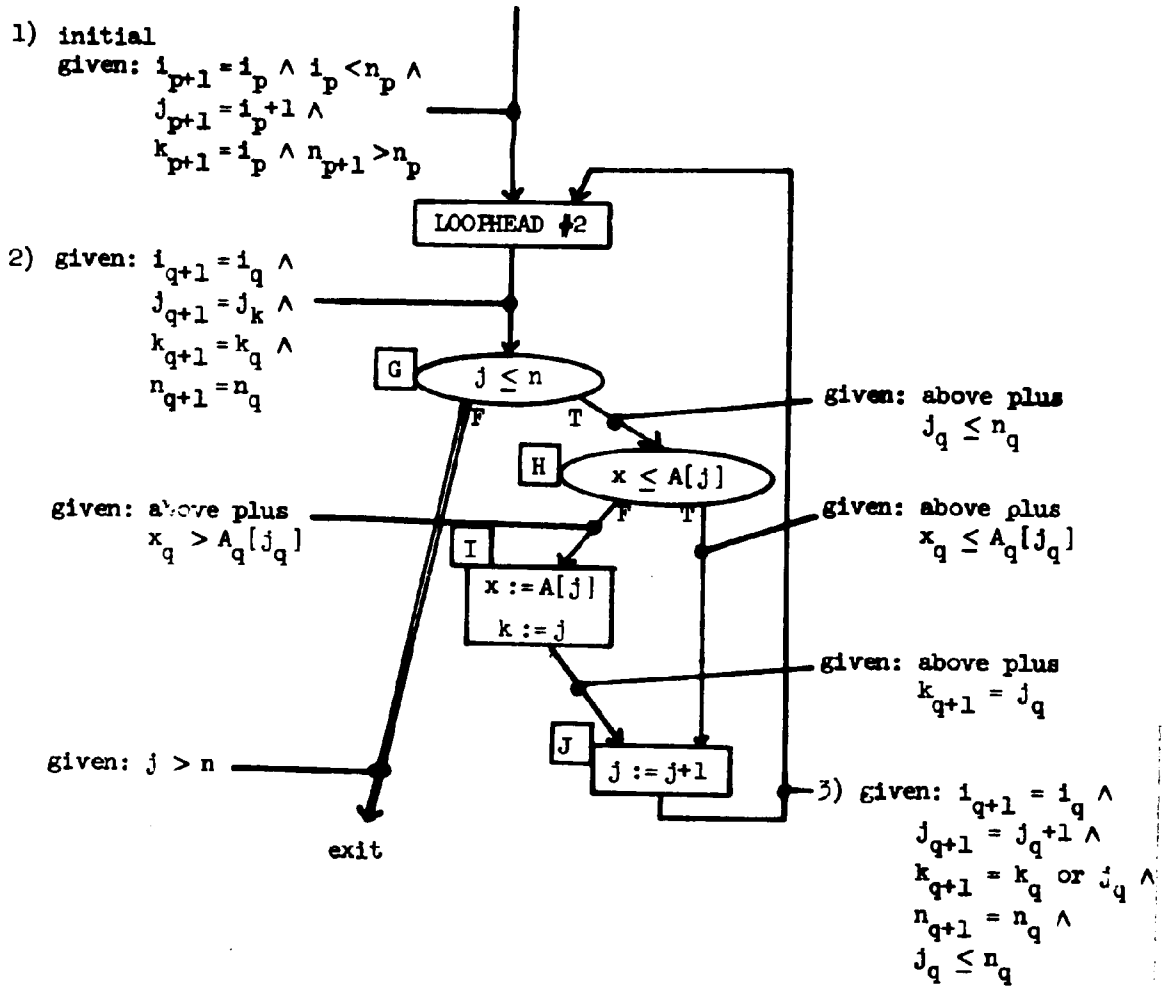

Figure A9.3. Gathering of given information for i , j , k , and n in inner loop, during first pass through outer loop. The recurrence relations attached after node J show that i and n are invariant in the loop, that j is monotonically increasing, and that k is some subset of the values that j takes on. Combining these recurrence relations with the initial input conditions from the first pass through the outer loop, we find that, at the LOOPHEAD #2 node:

4) $i_{p+1} = i_p \wedge i_p < n_p \wedge n_{p+1} = n_p \wedge$
$j_{p+1} = \{i_p+1, i_p+2, \ldots, n_p+1\} \wedge$
$k_{p+1} \subset \{i_p, i_p+1, \ldots, n_p\}$.

Example 10. The 91 Function.

The program we consider is a derivative of the recursive 91 function. The iterative version we deal with requires most of the graph transformations described in Chapter 1 and most of the merged information mechanisms described in Chapter 4 for the successful proof of its termination.

As stated in [Manna et. al. 1972, pp. 32 and 43], the 91 function is

$$F(x) \Leftarrow \underline{\text{if}}\ x > 100\ \underline{\text{then}}\ x-10\ \underline{\text{else}}\ F(F(x+11)) \quad .$$

This function returns $x-10$ if $x > 100$ and 91 otherwise. The initial form we use comes from a mechanical transformation of the recursive definition into an iterative one using an explicit stack. The stack index is k, and the only content of the stack is how many calls of F are still to be done, so k itself is used as this counter.

We shall concentrate on proving that the loops in this program terminate, and shall ignore the other issues, such as overflow. The reader may wish to convince himself that i does not overflow, and that k might.

Figure A10.1 gives the initial, user-supplied flow graph. Using the methods described in Chapter 1, this graph is transformed into the one in Figure A10.2. The first loop in this graph terminates because i is monotonically increasing, and hence will eventually exceed 100. The figure shows the given information available on initial entrance to the major loop, LOOP #2.

Using p and q as the iteration subscripts in loops #2 and #3 respectively, we find on the first pass through LOOP #2, that we enter the LOOPHEAD #3 node with:

Example 10. 91 Function

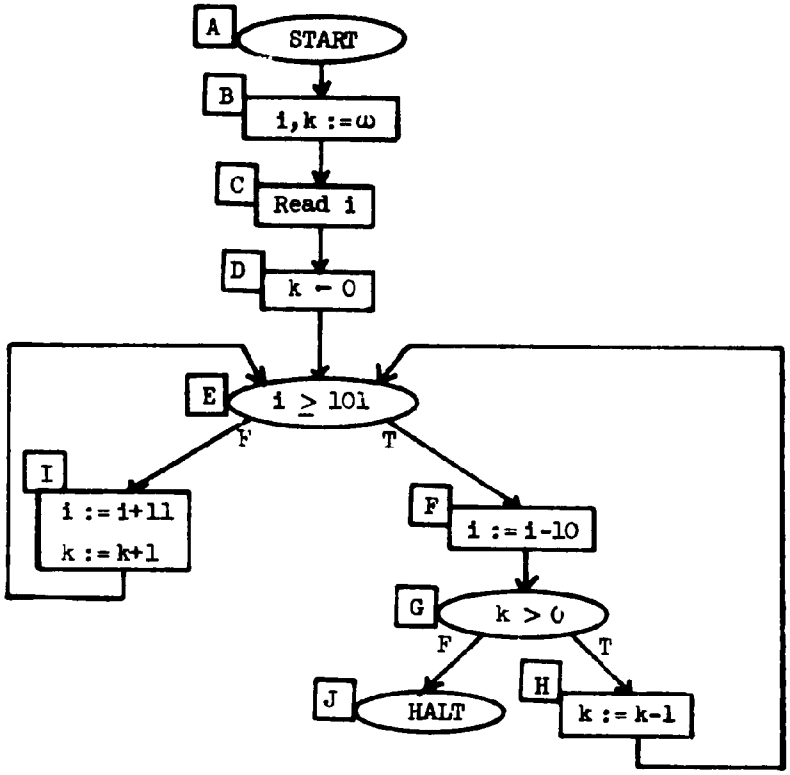

Figure A10.1. Flow graph of the 91 function before any of the Chapter 1 graph manipulations have been performed. We will make the loop E - I - E into an inner loop with exit to F , then make the loop E - F - G - H - E into an outer loop with exit to J . We will then permute the nodes of the outer loop so that it has a leading exit test.

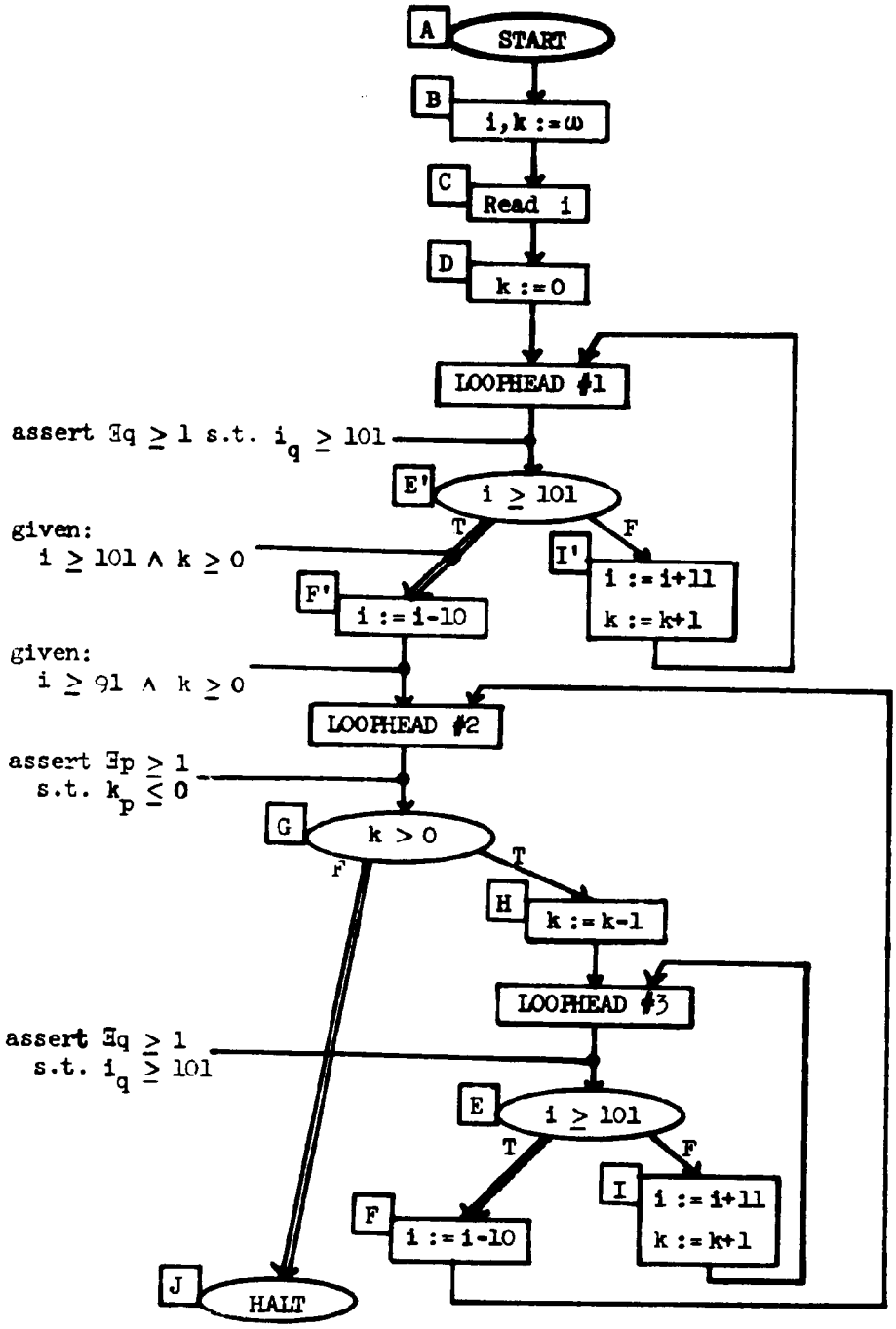

Figure A10.2. Structure of the flow graph in Figure A10.1 after separating the loops and putting them in leading test form. The loop termination assertions are shown, along with the initial entry conditions for LOOP #2 .

$$i_{p+1} = i_p \wedge k_{p+1} = k_p - 1 \wedge k_p > 0 \quad .$$

The subsequent induction step for LOOP #3 is shown in Figure A10.3. We then apply our knowledge that

$$i_{p+1} \geq i_p \wedge k_{p+1} \geq k_p - 1 \wedge k_p > 0$$

to nodes E and I on a second pass through LOOP #3, and then exit to node F, carrying the information:

$$i_{p+1} \geq i_p \wedge k_{p+1} \geq k_p - 1 \wedge k_p > 0 \wedge i_{p+1} \geq 101 \quad .$$

Passing through node F, we find that:

$$i_{p+1} \geq i_p - 10 \wedge i_{p+1} \geq 91 \wedge k_{p+1} \geq k_p - 1 \wedge k_p > 0 \quad .$$

The induction step for the outer loop (LOOP #2) is shown in Figure A10.4. We discover there that $i \geq 91$ at the LOOPHEAD #2 node during <u>all</u> iterations of the outer loop. With this tighter information about i, we start a third pass through LOOP #3 by re-doing the loop induction, as shown in Figure A10.5. We discover that $i \geq 91$ at the LOOPHEAD #3 node. We combine this information with the test $i \geq 101$ to find that on entry to node I,

$$91 \leq i \leq 100 \wedge k \geq 0 \quad ,$$

and hence after node I, that

$$102 \leq i \leq 111 \wedge k \geq 1 \quad .$$

This tight restriction on i during all but the first iteration of LOOP #3 allows us to elide the test $i \geq 101$ and in fact get rid of LOOP #3 entirely, as shown in Figures A10.6 and A10.7.

On exit from node I in this newly-structured graph, we know that $k \geq 1$, so we can merge this with the $k \geq 0$ on the arc from E to F to get

$$k \geq 0 \quad \text{(Refinement)}$$

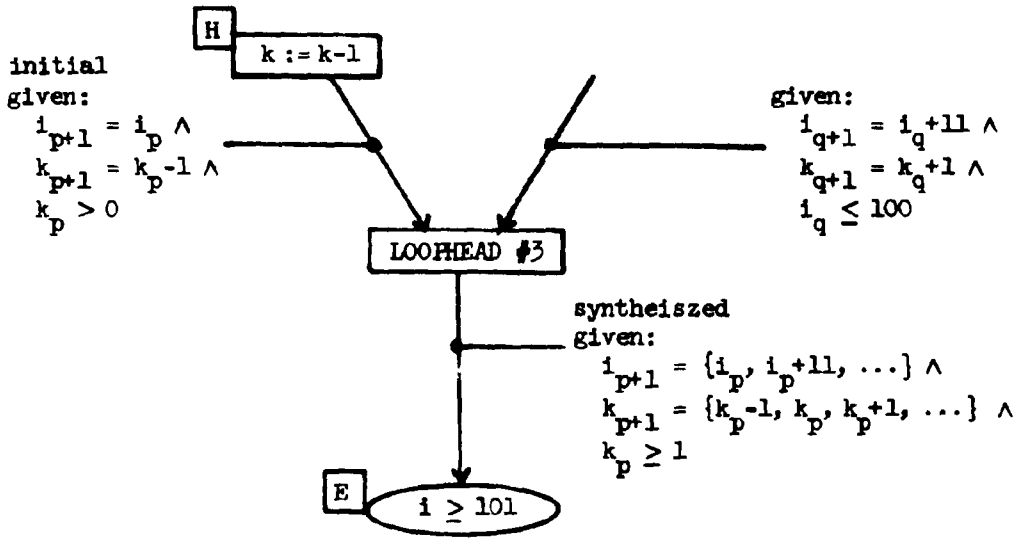

Figure A10.3. Induction step in LOOP #3 . Since we are within the first pass through LOOP #2 , we are developing ranges for the outer loop induction variables. Since we know nothing at this point about i_p , we cannot say that $i_{p+1} \leq 111$ inside LOOP #3 ; even though that is true when coming around the loop, it may not be true on initial entry.

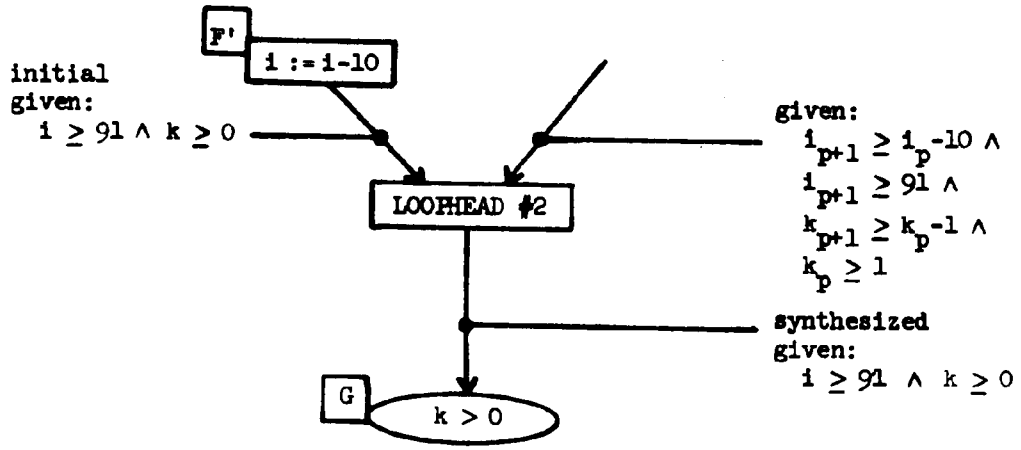

Figure A10.4. Induction step in LOOP #2 . The fact that $i \geq 91$ inside the loop will be crucial in our third-pass processing of LOOP #3 .

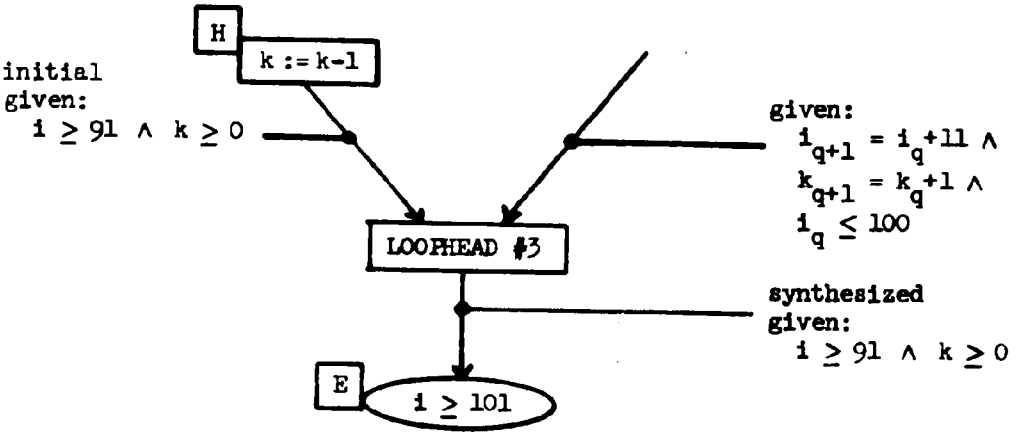

Figure A10.5. Loop induction for starting the third pass through LOOP #3 . Now for the first time we know that $i \geq 91$ at the LOOPHEAD #3 node on <u>all</u> iterations of LOOP #3 .

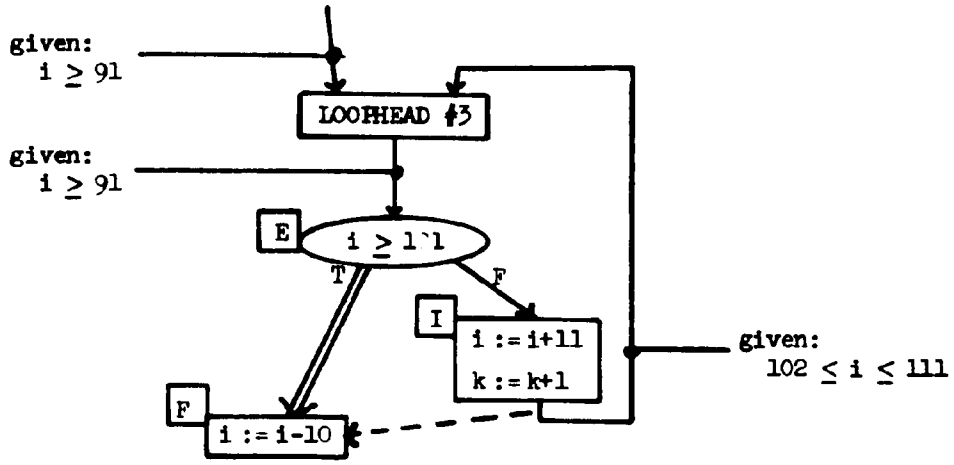

Figure A10.6. Elision of the test $i \geq 101$ along the path E - I - E . Since $102 \leq i$ when coming around the loop, the test of $i \geq 101$ is always true on the second iteration.

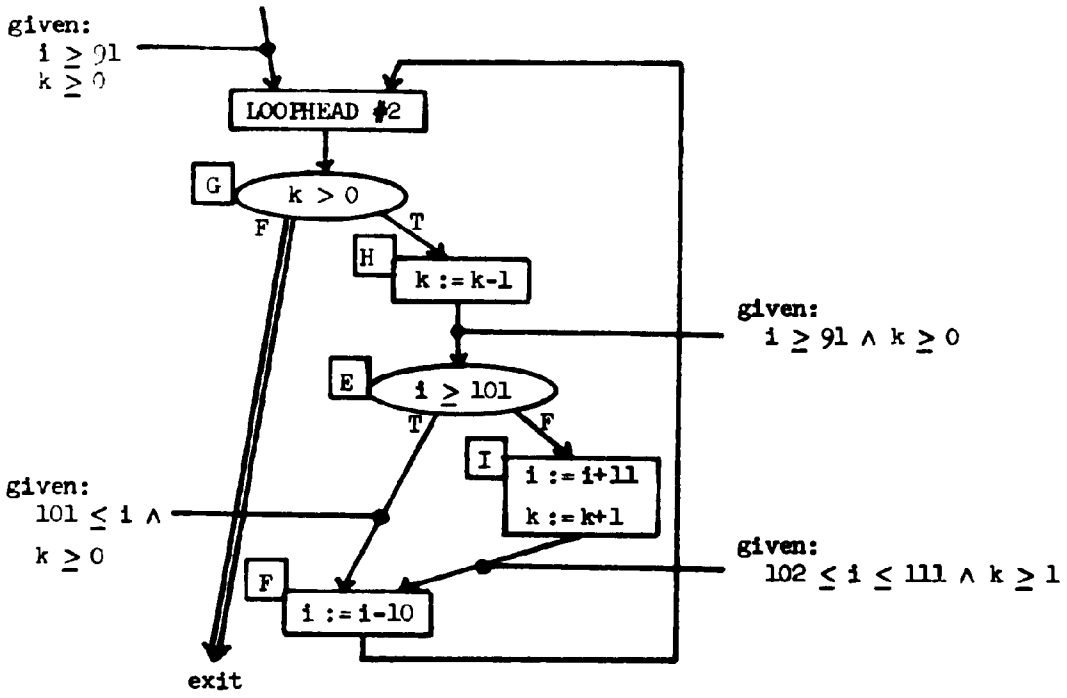

Figure A10.7. New structure of LOOP #2 after elimination of LOOP #3 . We now can elide the test of k on the path I - F - G .

on exit from node F , the refinement being that on one path we know the stronger condition $k \geq 1$. We can now attempt to elide the test at node G , and find the attempt successful along the path I - F - G , as shown in Figure A10.8, where the node F has been copied. Figure A10.9 shows the resulting nested loop structure, both of whose loops are easily seen to terminate.

We have now proved that all the loops in the iterative program for the 91 function terminate, by using only mechanical transformations of the flow graph and some simple theorem proving. The iterative program and its mechanical transformation from the recursive form are due to Donald Knuth, and have the property that if the iterative form terminates, so does the recursive one.

Combining any of the standard proofs of partial correctness of the 91 function with our proof of termination gives a proof of total correctness, with the only exposure being that k may overflow (or in the recursive form, the stack may overflow).

[I don't know if it is just a fluke that the mechanical test elision process was able to create an inner loop with k invariant, but it was certainly quite suspenseful the first time I worked all the way through this example. Originally, this was to be my example of how the thesis techniques could fail to prove loop termination.]

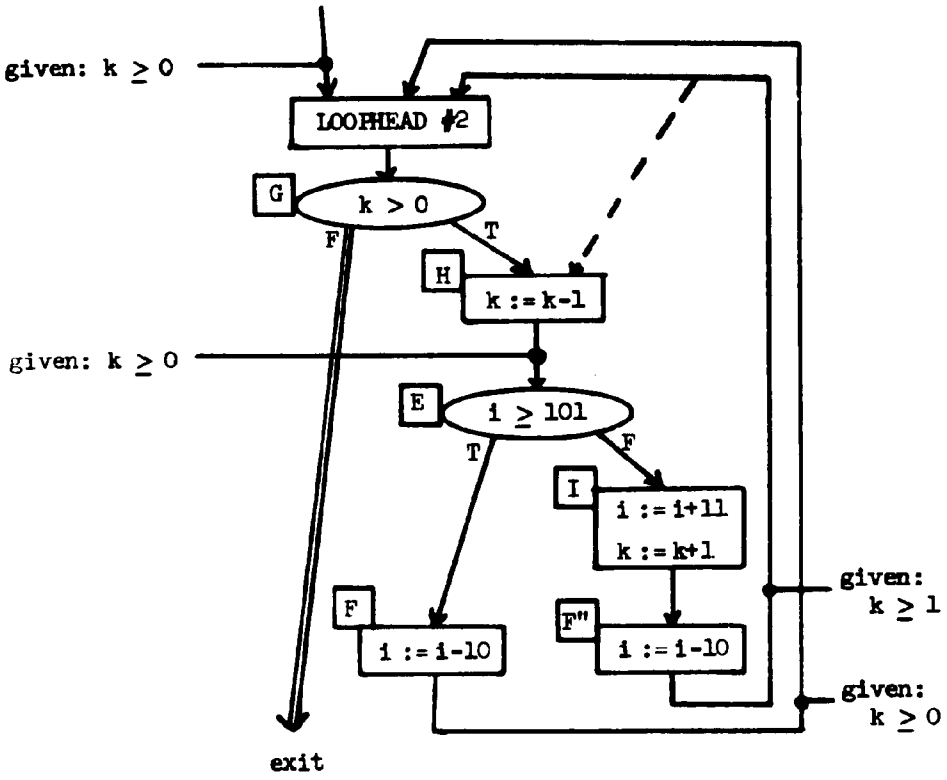

Figure A10.8. Elision of the test $k > 0$ along one path. When the structure of this new flow graph is analysed, we will have a new loop nested inside LOOP #2 . We have now made some significant progress, because k is invariant inside this inner loop.

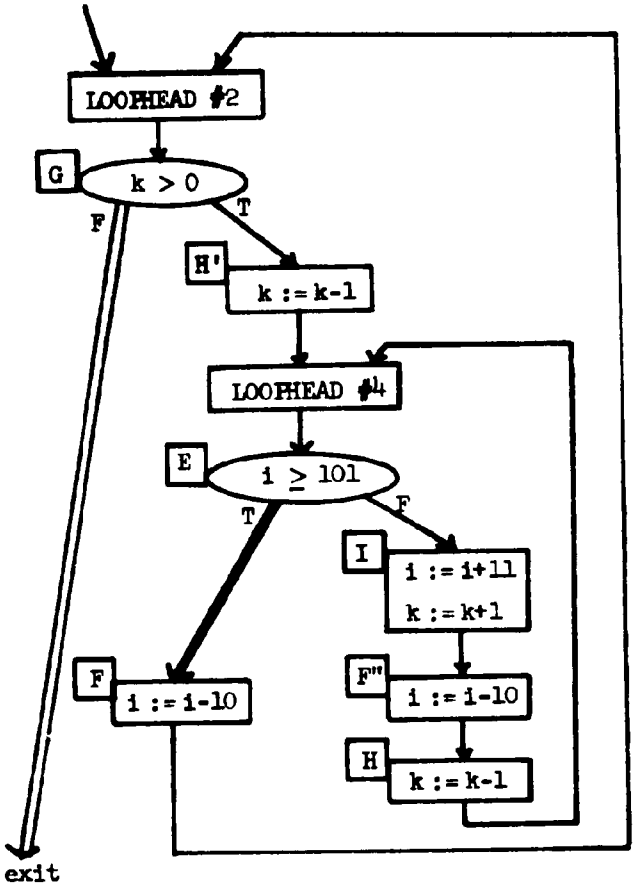

Figure A10.9. Final structure of nested loops #2 and #4. We cannot combine nodes I - F'' - H into a single i := i+1 , unless we can prove that no overflows occur. However, we do find the recurrence relations in LOOP #4 :

$$i_{q+1} := i_q + 1 \quad \wedge \quad k_{q+1} = k_q \quad .$$

Thus, LOOP #4 terminates because i is monotonically increasing. Since k is invariant in LOOP #4 , we find in the re-analysis of LOOP #2 that $k_{p+1} = k_p - 1$, and hence that LOOP #2 terminates because k is monotonically decreasing.

Appendix B. Node Visiting Algorithm from Chapter 4.

```
procedure VISIT (firstnode, lastnode);
    for n := firstnode to lastnode do
        begin
        1)  for each i in INCOMINGARCS(n) do
                PROVE (GIVEN(i) ⊃ ASSERTIONS(i));
        2a) if NODE(n) = LOOPHEAD then
              begin
              GIVEN (EXITARC(n)) := DUMMY BINDINGS V_{N+1} = V_N;
              VISIT (FIRSTNODEINSIDELOOP(n), LASTNODEINSIDELOOP(n));
              GIVEN (EXITARC(n)) := INDUCT (GIVEN (INITIALARCS(n)),
                                            GIVEN (LATCHBACKARCS(n)))
              end
            else begin
        2b)   if NODE(n) = TEST then
                for each i in INCOMINGARCS(n) do
                  begin
                  PROVE (GIVEN(i) ⊃ TESTEXPRESSION(n));
                  PROVE (GIVEN(i) ⊃ not TESTEXPRESSION(n))
                  if either is true then
                    elide the test and re-analyze the graph
                  end
        3)    g := MERGE (GIVEN(i) for each i in INCOMINGARCS(n));
              g := MERGE (g, ASSERTIONS(i) for each i in INCOMINGARCS(n));
              gprime := REFLECTASSIGNMENTS(g);
              if NODE(n) = TEST then
                begin
                GIVEN (TRUEEXIT(n)) := MERGE (gprime, TESTEXPRESSION(n));
                GIVEN (FALSEEXIT(n)) := MERGE (gprime, not TESTEXPRESSION(n))
                end
              else
                GIVEN (EXITARC(n)) := gprime
              end
        end
```

Bibliography

page referenced

[Allen 1970] 4, 9

Frances E. Allen, "A Basis for Program Optimization," IBM Research Report RC3138, T. J. Watson Research Center, Yorktown Heights, N. Y., November 1970, pp. 3-6.

[Allen and Cocke 1972] 4, 7, 9

Frances E. Allen and John Cocke, "Graph-Theoretic Constructs for Program Control Flow Analysis," IBM Research Report RC3923, T. J. Watson Research Center, Yorktown Heights, N. Y., July 1972, p. 28ff.

[Ashcroft and Manna 1972] 7, 25

Edward Ashcroft and Zohar Manna, "The Translation of 'Go To' Programs to 'While' Programs," Information Processing 71, North-Holland Publishing Company, 1972, pp. 250-255.

[Brent 1973] 5

Richard P. Brent, "Reducing the Retrieval Time of Scatter Storage Techniques," C.ACM 16, February 1973, pp. 105-109.

[Burstall 1970] 56

R. M. Burstall, "Formal Description of Program Structure and Semantics in First Order Logic," Machine Intelligence 5, Edinburgh University Press, 1970, pp. 79-98.

[Clint and Hoare 1972] 56

M. Clint and C. A. R. Hoare, "Program Proving: Jumps and Functions," Acta Informatica 1, 1972, pp. 214-224.

[Cocke and Schwartz 1970] 9

John Cocke and Jacob T. Schwartz, "Programming Languages and Their Compilers: Preliminary Notes," Courant Institute of Mathematical Sciences, New York University, N. Y., April 1970, pp. 442-461.

[Dahl and Hoare 1972] 59

Ole-Johan Dahl and C. A. R. Hoare, "Hierarchical Program Structure," in Structured Programming, Academic Press, New York, 1972, pp. 175-220.

[Deutsch 1973] 54, 55, 56, 66
L. Peter Deutsch, "An Interactive Program Verifier," Ph.D. Thesis, Computer Science Department, University of California Berkeley, June 1973.

[Earnest et al. 1972] 16
C. P. Earnest, K. G. Balke, and J. Anderson, "Analysis of Graphs by Ordering of Nodes," J.ACM 19, January 1972, pp. 23-42.

[Elspas et al. 1972a] 56
Bernard Elspas, M. W. Green, Karl N. Levitt, and Richard J. Waldinger, "Research in Interactive Program-Proving Techniques," S.R.I., Menlo Park, Calif., May 1972.

[Elspas et al. 1972b] 54, 56
Bernard Elspas, Karl N. Levitt, Richard J. Waldinger, and Abraham Waksman, "An Assessment of Techniques for Proving Program Correctness," Computing Surveys 4, June 1972, pp. 97-147.

[Floyd 1964] 5
Robert W. Floyd, "Algorithm 245 -- Treesort 3," C.ACM 7, December 1964, p. 701.

[Floyd 1967] 17, 54
Robert W. Floyd, "Assigning Meanings to Programs," Proceedings of a Symposium on Applied Mathematics, American Mathematical Society 19, 1967, pp. 19-32.

[Floyd and Rivest 1973] 5
Robert W. Floyd and Ronald L. Rivest, "Bounds on the Expected Time for Median Computation," Combinatorial Algorithms, edited by Randell Rustin, Algorithms Press, 1973, pp. 69-76.

[Fritsch et al. 1973] 5
F. N. Fritsch, R. E. Shafer, and W. P. Crowley, "Algorithm 443 -- Solution of the Transcendental Equation $we^w = x$," C.ACM 16, February 1973, pp. 123-124.

[Gerhart 1972] 54, 55, 60
Susan L. Gerhart, "Verification of APL Programs," Ph.D. Thesis, Carnegie-Mellon University, November 1972, 216 pp.

[Good 1970] 56
Donald I. Good, "Toward a Man-Machine System for Proving Program Correctness," Ph.D. Thesis, University of Wisconsin. Also Computation Center Memo TSN-11, University of Texas, Austin, Texas, June 1970, 179 pp.

[Good and London 1970] 59
Donald I. Good and Ralph L. London, "Computer Interval Arithmetic: Definition and Proof of Correct Implementation," J.ACM 17, October 1970, pp. 603-612.

[Hoare 1961] 56
C. A. R. Hoare, "Algorithm 65, FIND," C.ACM 4, July 1961, pp. 321-322.

[Hoare 1969] 56
C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," C.ACM 12, October 1969, pp. 576-580, 583.

[Hoare 1971a] 56
C. A. R. Hoare, "Proof of a Program: FIND," C.ACM 14, January 1971, pp. 39-45.

[Hoare 1971b] 54, 56
C. A. R. Hoare, "Procedures and Parameters: An Axiomatic Approach," Symposium on Semantics of Algorithmic Languages, Springer-Verlag, 1971, pp. 102-116.

[Hull et al. 1972] 59
T. E. Hull, W. H. Enright, and A. E. Sedgwick, "The Correctness of Numerical Algorithms," Proceedings of an ACM Conference on Proving Assertions About Programs, SIGPLAN Notices, January 1972, pp. 66-73. (Also SIGART Notices, January 1972.)

[Igarashi et al. 1973] 56
Shigeru Igarashi, Ralph L. London, and David C. Luckham, "Automatic Verification of Programs I: A Logical Basis and Implementation," Computer Science Department Report CS 365, AIM 200, Stanford University, May 1973, 53 pp.

Bibliography

[King 1969] 3, 5, 54, 66, 100

James C. King, "A Program Verifier," Ph.D. Thesis, Carnegie-Mellon University, National Technical Information Service, Springfield, Virginia 22151, #AD 699248, September 1969, 255 pp.

[Knuth 1973a] 5

Donald E. Knuth, "A Review of 'Structured Programming'," Computer Science Department Report CS371, (Clearinghouse # PB 223572/A), Stanford University, June 1973, 25 pp.

[Knuth 1973b] 5, 16

Donald E. Knuth, The Art of Computer Programming, Volume 1 - Fundamental Algorithms, Addison-Wesley, Reading, Mass., 1973.

[Knuth and Floyd 1971] 62

Donald E. Knuth and Robert W. Floyd, "Notes on Avoiding 'Go To' Statements," Information Processing Letters 1, 1971, pp. 23-31, 177.

[London 1970a] 54

Ralph L. London, "Bibliography on Proving the Correctness of Computer Programs," Machine Intelligence 5, Edinburgh University Press, 1970, pp. 569-580.

[London 1970b] 3, 5

Ralph L. London, "Certification of Algorithm 245[M1] Treesort 3: Proof of Algorithms -- A New Kind of Certification," C.ACM 13, June 1970, pp. 371-373. (Also see [Redish 1971].)

[London 1972] 54, 56

Ralph L. London, "The Current State of Proving Programs Correct," Proceedings of ACM National Conference 27:1, ACM, August 1972, pp. 39-46.

[Malcolm and Palmer 1974] 59

Michael Malcolm and John Palmer, "A Fast Method for Solving a Class of Tridiagonal Linear Systems," C.ACM 17, January 1974, pp. 14-17.

[Manna 1969] 56

Zohar Manna, "The Correctness of Programs," Journal of Computer and System Sciences 3, May 1969, pp. 119-127.

[Manna et al. 1972] 5, 54, 56, 66
Zohar Manna, Stephen Ness, and Jean Vuillemin, "Inductive Methods of Proving Properties of Programs," Proceedings of an ACM Conference on Proving Assertions About Programs, SIGPLAN Notices, SIGART Notices, January 1972, pp. 27-50. (Las Cruces, New Mexico, Conference.)

[Manna and Pnueli 1973] 56
Zohar Manna and Amir Pnueli, "Axiomatic Approach to Total Correctness of Programs," Computer Science Department Report CS 382 (Clearinghouse #AD 767335), Stanford University, July 1973, 25 pp.

[Naur 1963] 20
Peter Naur (Editor), "Revised Report on the Algorithmic Language ALGOL 60," C.ACM 6, January 1963, pp. 1-23.

[Redish 1971] 3
K. A. Redish, "Comment on London's Certification of Algorithm 245," C.ACM 13, January 1970, pp. 50-51.

[Reingold 1973] 5
Edward M. Reingold, "A Nonrecursive List Moving Algorithm," C.ACM 16, May 1973, pp. 305-307.

[Sites 1972] 61
Richard L. Sites, "Algol W Reference Manual," Computer Science Department Report CS 230 (Clearinghouse #PB 203601), Stanford University, February 1972, 141+ pp.

[Sites 1974] . . 3, 5, 17, 53, 66, 103
Richard L. Sites, "Some Thoughts on Proving Clean Termination of Programs," Computer Science Department Report CS 417, Stanford University, May 1974, approximately 60 pp.

[Smith 1972] 56
J. Meredith Smith, "Proof and Validation of Program Correctness," The Computer Journal 15, pp. 130-131.

[Waldinger and Levitt 1973] 55
Richard J. Waldinger and Karl N. Levitt, "Reasoning About Programs," ACM Symposium on Principles of Programming Languages, ACM, October 1973, pp. 169-182.

[Wegbreit 1974] 56

Ben Wegbreit, "The Synthesis of Loop Predicates," C.ACM 17, February 1974, pp. 102-112.

[Yohe 1970] 59

J. M. Yohe, "Best Possible Floating-Point Arithmetic," Mathematics Research Center Summary Report No. 1054, University of Wisconsin, March 1970.

Index

Notation