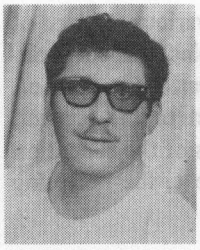


Nissim Francez was born in Bulgaria in 1944. He received the B.A. degree in mathematics and philosophy from the Hebrew University, Jerusalem, in 1965, and the M.S. and Ph.D. degrees, both in applied mathematics (computer science), from the Weizmann Institute of Science, Rehovot, Israel, in 1971 and 1976, respectively. His Ph.D. dissertation, under the supervision of Prof. A. Pnueli, was on the specification and verification of cyclic (sequential and concurrent) programs.

Between 1970 and 1976, he taught courses in high-level programming languages (Pascal, Fortran, Lisp, and SNOBOL) at both Beer Sheva and Tel-Aviv Universities. He has also taught courses in the introduction to system programming (using MIX) and compiler construction at the same universities. After completing work on his Ph.D., he was a Research Fellow at Queen's University, Belfast, on an SRC grant, where he participated (with W. P. de Roever) in a research project conducted by C. A. R. Hoare on the semantics of communicating sequential processes. In September 1977 he submitted a research proposal for the continuation of these studies to the National Science Foundation. He is currently teaching at the University of Southern California, Los Angeles. He has written a Fortran programming textbook with G. Amir, and has a book on Pascal in preparation. His main areas of interest are programming languages, semantics and methodology, concurrent program verification, and general theory of programming and computation.

Dr. Francez is a member of the Association for Computing Machinery and SIGPLAN.

Program Optimization Using Invariants

SHMUEL KATZ

Abstract—**Optimizing a computer program is defined as improving the execution time without disturbing the correctness. We show how to use invariants from a proof of correctness in order to change the statements in and around the program's loops. This approach is shown to systematize existing optimization methods, and to sometimes allow stronger optimizations than are possible under the standard transformation approach.**

Index Terms—**Invariants, program optimization, proof of correctness, transformations of programs.**

I. Introduction

FOR MANY YEARS compilers have contained sections which are supposed to "optimize" the code produced from computer programs. As has often been noted, this term is a misnomer because a really "optimal" solution to the "optimization" problem would involve throwing away the original program and producing in its place the best possible program to perform the desired task.

In view of the present state of program synthesis, we adopt a more standard (and considerably less ambitious) definition as the goal of optimization. Optimization is intended to improve the execution time of a given program by changing or moving some of the statements, without disturbing the correctness of the program. The same crucial relationships will be maintained among the variables, but the computations and tests will be altered in order to reduce the time required for computation.

A wide variety of techniques are currently grouped under the term optimization. Among these are various machine-dependent operations (including register allocation) which use special characteristics of a given computer and are best applied to machine code. We concentrate on the other large class of techniques, namely, various program transformations which are independent of the machine code. These are either applied to an intermediate-level program (similar in complexity to a flowchart language) before actual machine code is generated or directly applied to source-level programs.

In this paper, a method is presented for performing optimizations of the latter type with the aid of a proof of correctness of the program. That is, in addition to the program, the user has provided 1) an *input specification*, defining the acceptable input values which the program is intended to treat, and 2) an *output specification*, defining the desired relationship between the input and the output. Then intermediate "invariant assertions" have been attached to prechosen points in the program. These assertions allow proving the correctness of the program with respect to its specification. Note that by "correctness" we mean that for every legal input the program will terminate, *and* the output will satisfy the output specification (this is often called *total* correctness).

An invariant assertion (or *invariant*[1] for short) at a point A is any claim about the variables which is always true when the control of the program reaches point A. The following section contains the definitions needed to give a precise criterion for proving an assertion to be an invariant.

Manuscript received November 1,1976; revised March 15, 1978.

The author is with the IBM-Israel Scientific Center, Technion City, Haifa, Israel.

[1] Note that we use the word "invariant" in the sense common to program verification and *not* as sometimes used in articles on optimization, where an "invariant expression" means an expression containing only variables unchanged in the loop.

0098-5589/78/0900-0378$00.75

A situation in which we would like to perform optimization of a program and also have available the program specification and a proof of correctness—including the invariants used in that proof—could arise from at least two sources.

1) The program may have been developed in a top-down manner by stepwise refinements using structured programming techniques (see, e.g., [16]). We assume a stage at which a complete but unoptimized program has been obtained, and that at each stage the correctness has been proven—perhaps somewhat informally—by demonstrating the needed invariants. In this case, the invariants are available, even though they may not be organized in the manner described in this paper. Moreover, the final optimization stage can be done at least semi-automatically, even though earlier stages of the refinement do not seem to be as amenable to automatization.

2) The program (either developed by stepwise refinements or in any more haphazard manner) may have undergone logical analysis, preferably automatically, so that some invariants have been extracted from the program independently of the output specification. Elements of such a system exist, and several such systems are proposed or under construction (e.g., [5], [12]). If the program is incorrect, a logical analysis system would be used to prove this and to help diagnose and correct the errors. If it is correct, partial correctness and termination would be proved for the given specification, also using the invariants. Once the program has been proved, a logical analysis system would pass to an optimizer, with the invariants already organized as indicated in the following section.

It should be noted that no present system is capable of conducting a logical analysis of large-scale programs. This is primarily due to the difficulty of generating invariants for such programs without aid from the programmer. In fact, one of the goals of this work is to show the relevance of invariants for problems besides correctness, and thus help act as a spur both for the development of better invariant-generating techniques and for cooperation from the programmer on annotating his program with key invariants.

In any case, the time-consuming and difficult task of finding the proper invariants and the proof would be done primarily for other reasons, and not merely for optimization. Thus it is worthwhile to take advantage of the added information which is available "free of charge," as an aid in optimization. In Section II, some definitions and facts about invariants are presented, and their organization for the purpose of optimization is described. Section III examines some well-known optimizations which are facilitated by using invariants, while Section IV demonstrates optimizations which are virtually impossible unless an appropriately organized correctness proof is available. In the conclusion (Section V), the techniques described below are compared with the usual optimizations found in compilers and with related work on transformations which preserve correctness.

II. Invariants and their Organization for Optimization

First, some definitions and facts related to invariants are briefly summarized, using the terminology of [10].

For convenience of explanation, properly structured *blocked programs* are assumed (although this is not really essential to the ideas). That is, the programs treated are divisible into (possibly nested) "blocks" in such a way that every block has at most one top-level loop (in addition to possible lower level loops which are already contained in inner blocks). The blocks considered have one entrance and may have many exits. Algorithms for identifying such blocks can be found in [1]. The top-level loop of a block can contain several branches, but all paths around the loop must have at least one common point. For each loop, one such point is chosen as the *cutpoint* of the loop. (A few comments on treating nested blocks precede Example 3.)

A *counter* can be associated with the cutpoint of the loop. The counter is initialized before entering the block so that its value is zero upon first reaching the cutpoint, and is incremented by one exactly once somewhere along the loop before returning to the cutpoint. In the continuation, a *local* initialization of each counter is assumed immediately before the entrance to its block.

Counters can be useful both for generating invariants and for proving termination. They denote relations among the number of times various paths have been executed and also help express the values assumed by the program variables. Thus $y_i(n)$ denotes the value of y_i the (n+1)-th time the cutpoint is reached since the most recent entrance to the block. It should be noted that it is unnecessary to add the counters physically to the body of the program. Their location may merely be indicated, since their behavior is already fixed.

It is also sometimes convenient to add auxiliary cutpoints at the entrance and exit of a block. In addition, a special cutpoint is added on each arc immediately preceding a HALT statement. Such cutpoints are HALT-points of the program.

In the following definitions, $\bar{x}$ denotes the input values, and $\bar{y}$ the values of the program variables at the cutpoint being considered.

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

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

As is well known, assertions at a complete set of cutpoints can be shown to be invariants by an induction on the computation: for every path α from cutpoint i to cutpoint j, it must be proved that the assertions at i, say $q_i(\bar{x}, \bar{y})$, and the path condition $R_\alpha(\bar{x}, \bar{y})$ together logically imply the assertions at j with $r_\alpha(\bar{x}, \bar{y})$ substituted for the program variables; i.e., $q_j(\bar{x}, r_\alpha(\bar{x}, \bar{y}))$.

As is noted in the Introduction, discovering the appropriate assertions to be used in these proofs is the most difficult aspect of program verification, and is best done by the programmer himself. Nevertheless, numerous techniques have been

developed to obtain invariants directly from the program. For example, the path functions $r_\alpha(\bar{x}, \bar{y})$ define difference equations expressing the changes in the variables for one pass around a loop. The solutions of these equations yield invariants. It is also possible to examine what was established by the tests made as the paths are followed. This is in the path condition $R_\alpha(\bar{x}, \bar{y})$. By their construction, the results of these techniques must be invariants, and they are therefore termed "algorithmic" methods.

Another approach seeks to "guess" invariants by using heuristics to identify likely or desirable candidates. These could be based on the desired specification, on existing invariants, or on various indications in the program. Any candidates so generated must be checked. A deeper look into the invariant-generating techniques, and proving properties of programs with invariants, can be found in, e.g., [7], [8], [11], [12], [15]. Better techniques are still needed, and the programmer should be further motivated to provide the nontrivial invariants.

Although a conjunction of invariants is clearly an invariant in itself, we use the term invariant to refer to the individual conjuncts. Thus, after finding the assertions, we will have conjunctions of invariants at each cutpoint (including the output specifications at the HALT-points) which satisfy the logical implications previously described.

In order to effectively use the invariants and the proof of a program for optimization, we need to record the source of each invariant; i.e., precisely how (and which) program statements and/or other invariants were used in its derivation and/or proof.

This can be done in a table which notes for each invariant (on the one hand) the other invariants and the statements on the path(s) from previous cutpoints which are used in its derivation or proof, as well as information about the specific proof or derivation technique used. On the other hand, the uses of that invariant for proving other invariants must also be noted. The invariants can either be organized in separate tables for each cutpoint or in one large table where each invariant is also associated with the cutpoint at which it is true. By convention, the order of the statements on the path is indicated by their order from left to right in the table.

For simple programs with only one or two loops, the table may be represented pictorially as a directed graph having an arc from each statement or invariant A to any other invariant B which directly used A in its derivation or proof. Thus A could be an invariant true at a previous cutpoint, an assignment statement which affected the variables in B along a path from a previous cutpoint, or a test which was used in forming $R_\alpha(\bar{x}, \bar{y})$, which in turn helped establish B. The program statements will be the sources of this graph, and the specification along with a bound on the counters (proving termination) will be its sinks. In the continuation, if there is a path from A to B, then A is B's *ancestor*, and B is A's *son*. The graphic representation is termed a *derivation graph*.

In the case of an assertion B which must be proved to be an invariant by using an automatic theorem-prover or a proof-checker, the ancestors can be obtained by adding a module which records each invariant or statement $A_1, \cdots, A_n$ used in B's proof, and for each A_i records that it was used to establish B. A similar mechanism can be associated with the algorithmic invariant-generating techniques. For an informal proof, the prover would obviously have to provide the needed graph.

Self-arcs will not be included in the graphs we consider. Such arcs would arise whenever an invariant is used in an induction to help prove itself on the next pass around a loop, but for optimization purposes should be ignored. Under this assumption, and if a minimal number of cutpoints are used, there will be very few circuits in the graph, often none at all. Each closed circuit will involve only a few nodes and should be considered as one unit for the purposes of optimization. Known techniques of graph theory can be used to identify and treat the circuits as a unit. In the continuation, the treatment of closed circuits is not discussed in detail.

For clarity, we refer to various operations on this graph, but it should be clear that the tabular representation is the one which actually would be used in an implementation. Once the program has been proved correct and the graph has been generated, the invariants used in the correctness proof can be identified. That is, all ancestors of the specification will be marked. The basic optimization procedure will then be to "cut" the graph at invariants which we will decide are essential, and try to compute these invariants—or other invariants with an equivalent effect—in a more efficient way. Then any ancestor statement not used in either the new derivation, or in the derivation of another invariant needed in the correctness proof, can be removed from both the graph and the program. The precise methods used to obtain the invariant in a new way are described in the following section.

This cutting of the graph will, in effect, define a *level of optimization*. The nearer the cut is to the sources of the graph (i.e., to the program statements), the more local the optimization, while the nearer the cut is to the sinks of the graph (i.e., the vital invariants for proving the specification), the more global the optimization. It is usually best to directly find the invariants closest to the sinks for which we can optimize.

Before describing the optimizations considered, a simple example is given to demonstrate the table, the graph representation, and the levels. For the moment, the full justification of the optimizations performed is not included.

Example 1: From the Fortran statements (obviously a segment of a program)

```
      K = 0
      DØ 3 I = 1,1000
    3 K = L+1+I+K
      PRINT K
```

we might obtain the intermediate segment shown in Fig. 1. In Fig. 2, we show in solid lines the derivation graph (in this example, a tree) at A for this segment. For this tree, no other cutpoints inside the loop except A are used. The dotted lines indicate the tree of invariants at C after the loop.

In Fig. 3, the table representation is given if *two* cutpoints were chosen inside the loop, one after the assignment to T, and one at A, before the test. In this case, the graph would be more difficult to draw, but conceptually there is no change. The numbers to the left of the statements and invariants are identifiers which clearly would be replaced by pointers.

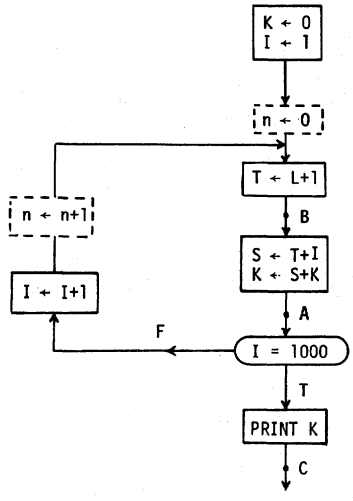


Fig. 1. Simple intermediate program.

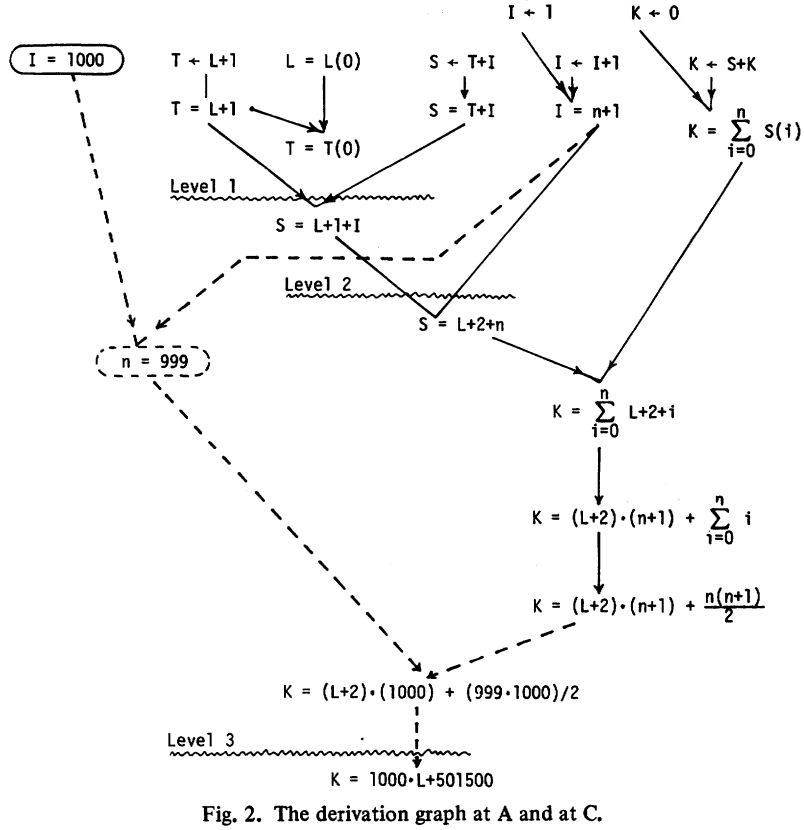


Fig. 2. The derivation graph at A and at C.

In the remainder of this example, the tree of Fig. 2 is used. By "cutting" this tree at various distances from the program statements, we obtain differing strengths of optimizations. Considering the line denoted in Fig. 2 as level 1, we would like to compute the invariant at A, S=L+1+I, in a different way. This can be done trivially by inserting S ← L+1+I just before A. All the statements above the invariant which are not used in other invariants may now be removed (in this example, T ← L+1 and S ← T+I).

If level 2 is instead considered, we would like to compute S=L+n+2 differently. However, since this invariant implies that S is linear in the counter n, and L is unchanged in the loop, this could be achieved by initializing S to L+2 before entering the loop, and then increasing S by 1 at each iteration,

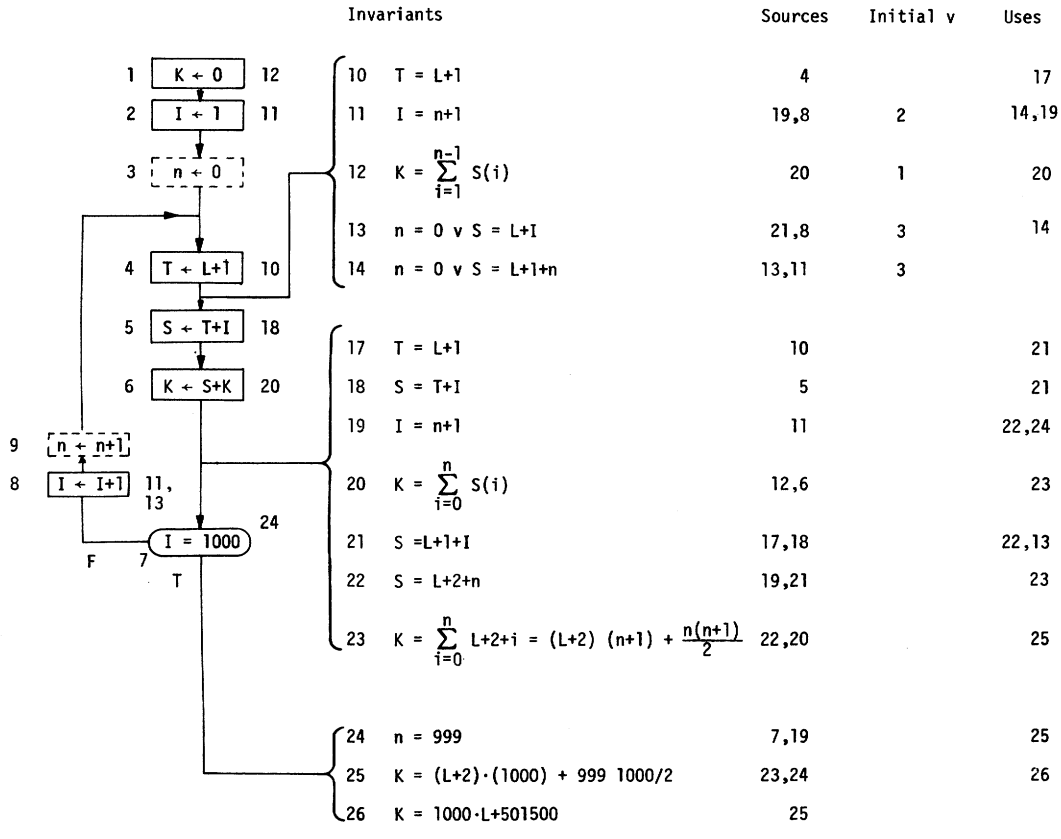


Fig. 3. The table for cutpoints A, B, and C.

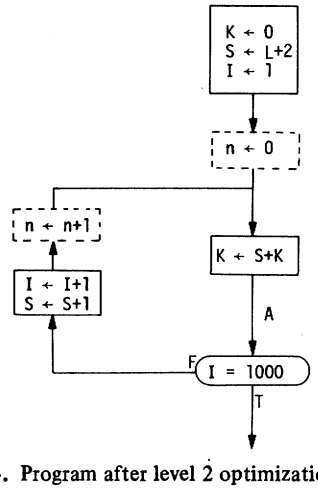


Fig. 4. Program after level 2 optimization.

just as n is increased. Most of the old statements above the invariant S=L+n+2 can then be removed. The resulting program is shown in Fig. 4 and the new derivation tree in Fig. 5.

Finally, if we optimize at level 3, we see that K = 1000 · L + 501500 is obtained by the simple assignment statement K ← 1000 · L + 501500, and that the entire loop is then extraneous. This assignment could lead to overflow, and involves one multiplication. However, the overflow would occur anyway in the original loop if it would occur here, and one multiplication seems better than a few thousand additions and 1000 tests.

Note that in this example the various levels of optimization could either have been treated consecutively or independently, and that it is most worthwhile to start at level 3, since then the other levels need not be considered at all.

III. Typical Optimizations using Invariants

Most of this section demonstrates that several well-known optimizations used in compilers can be easily applied using the information already in the invariants, rather than the various information-gathering algorithms generally employed. The fol-

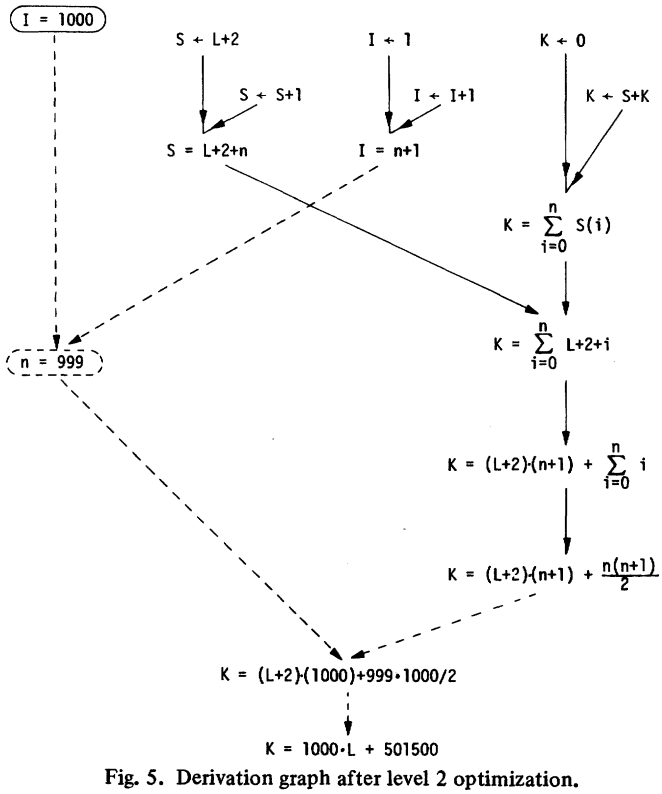


Fig. 5. Derivation graph after level 2 optimization.

lowing section describes optimizations stronger than those possible in standard compilers. Note that the optimizations treated are not language-dependent.

The point to be made in this section is that the applicability of seemingly quite different optimizations can be determined from a simple examination of the invariants. In fact, some optimizations are "automatically" invoked by the very method for employing the invariants, and do not have to be checked for independently.

The basic "automatic" tool for improving programs using the derivation graph will be eliminating redundant or "dead" (unused) statements. Simply, any statement not an ancestor of any marked invariant in the derivation graph can immediately be removed from both the graph and the program, since it has no effect on the correctness of the program.

Of course, in a program written with any care at all, there should not be any such extraneous statements in the original graph. However, after a cut has been made, and new statements inserted as the ancestors of an invariant (using the following optimizations), the old immediate ancestors are no longer needed for that invariant, and the relevant links are removed from the graph. If the statements which were ancestors are thereby no longer needed for any invariant involved in the correctness proof (i.e., are "dead"), they can be removed. Since statements will only be removed when they become "dead," we have a double-check on the legality of other optimizations (which may seem to preserve correctness, but in unusual situations might not). In the continuation, this tool will be called the *elimination criterion.*

Of course, a regular optimizer will often eliminate statements which have become syntactically dead (that is, they cannot be reached in any way). Here, in addition, statements can be removed which are logically redundant or will not be reached because of logical relations, even if syntactically they appear to be necessary. Thus the existing proof shapes and identifies which statements can be eliminated. Naturally, for *any* proof syntactically dead statements would be eliminated.

The optimizations described below will all use an identical *replacement procedure.* They all discover potential optimizations to compute, say, a variable v in a new way based on an invariant p (involving v) which is true at the cutpoint of the loop in question.

Then the newly generated statements are inserted in the graph as p's ancestors, and the old immediate ancestors ("fathers") are disconnected from p. If the elimination criterion can then be used to remove the old ancestor statements, the particular optimization is complete. If not, some of the ancestor statements are still needed for another invariant, say q, involved in the correctness proof. In this case, we try to derive q in a new way, based on p and/or other invariants, perhaps adding new statements, so that the old ancestors of p are not used. If this cannot be done, we must insert a new variable name, say t, in place of v in the original ancestor statements of p, and also replace v by t in q (and the other remaining derived invariants). Generally, the old statements can be in themselves optimized.

Intuitively, the relatively rare cases where a new variable is added occur when a variable fulfills two or more roles in the loop (e.g., is similar to a counter at one point in the loop, but is a constant at another point in the loop). In fact, it is generally good programming practice to delegate the roles of a variable to different variables, which can then be optimized separately.

Now some optimizations which lend themselves to the procedure described are examined in more detail. The common compiler optimizations described below loosely follow [1].

a) *Constant propagation* involves replacing a variable by its constant value, and performing at compile time operations with constants. This optimization arises naturally from the derivation graph, since such substitutions occur during the search for simpler invariants. Consider an expression in an invariant with a constant c which was derived by using an invariant v=c. The expression can usually be derived directly by substituting the constant c for the variable v in the *other* invariants or statements used to derive the expression. Then the modified invariants can be "pushed" back up the tree, until the relevant ancestor statements have been modified. The link from v=c is then removed, and the elimination criterion may be relevant. Similarly, algebraic simplification is employed to replace by its value an expression involving constants, and this too can be pushed back up to the relevant statement.

An example is shown in Fig. 6. Once the constant propagation has been done, if only d=7*k is used in the proof of correctness, d = v*k and v=7 are extraneous invariants, and they and their ancestor v ← 7 may be removed. Of course, the ancestor d ← 7*k is still needed for the marked invariant d=7*k.

This type of optimization can also be used for an invariant of the form v= **if** $\cdots$ **then** c_1 **else** c_2 arising from branching tests around the loop. No matter which statements are origi-

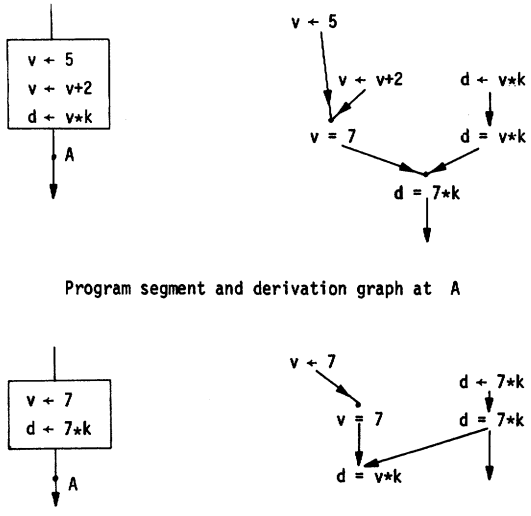


Fig. 6. Constant propagation example.

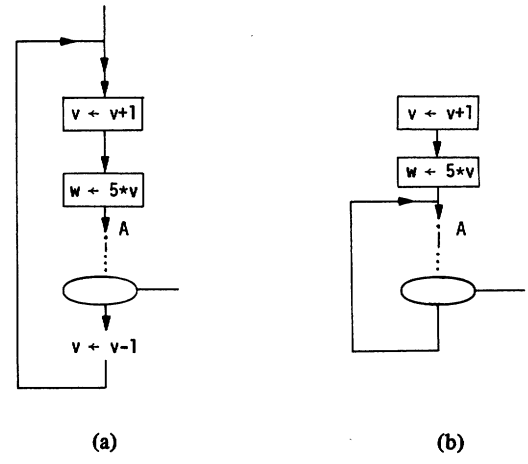


Fig. 7. Removing computation of constants with respect to loops.

nally used to compute c_i, it can be obtained alternatively by inserting $v \leftarrow c_i$ on the segment of the loop which is reached if the condition for v to be equal to c_i is true, and by using the replacement procedure with the old ancestors.

b) *Moving statements outside of loops* is probably one of the most beneficial optimizations. We call a variable v *constant with respect to the loop* if an invariant v=v(0) is true at the cutpoint of the loop. Recall that v(0) denotes the value of v the first time the *cutpoint* is reached (and *not* necessarily the value at the entrance to the loop). Thus v=v(0) is an invariant if at the cutpoint v is always equal to its value the first time the cutpoint is reached. The change in v along the paths from an entrance of the loop to the cutpoint can be computed as the path functions of that path. If this change is denoted as entr(v), we can add $v \leftarrow$ entr(v) before that entrance to the loop (or add nothing if the value is not changed between the entrance and the cutpoint). The replacement procedure will then be followed. Note that if the old assignments to v cannot be removed, v must be renamed in them to avoid interference with the new way of computing the invariant v=v(0).

Once such an invariant v=v(0) has been found, other appearances of v in invariants can be inspected to discover *expressions* which are constant with respect to the loop. For such an expression a new variable t can be defined, equal to the expression, and since t=t(0), its calculation can be removed from the loop.

Example: If we had the segment seen in Fig. 7(a) where v and w appear only as indicated and are used only at A, then v=v(0) at A and, entr(v) is v+1. Moreover, w=5*v at A; i.e., w=5*v(0)=w(0). Thus we may add $v \leftarrow v+1$ and $w \leftarrow 5*v$ before the loop and remove the original statements (again using the replacement procedure), obtaining the segment of Fig. 7(b).

c) *Reduction in strength* involves computing variables by using weaker operators in place of stronger ones; e.g., addition in place of multiplication.

The conditions for applying this optimization are particularly easy to identify by using invariants. Any invariant which connects a variable linearly to the loop counter, such as $v=c_1 \cdot n+c_2$, can be computed by initializing to c_2 and then incrementing v by c_1, i.e., inserting $v \leftarrow v+c_1$ to the loop (of course, the previous statements used to compute v must be removed or changed using the replacement procedure). In Example 1, such an optimization was performed. An example is shown in Fig. 8. Once the optimization has been done, should only the invariant involving L be needed in the proof, the invariants and statements with I, K, and J would be removed by the elimination criterion.

Other reductions in strength, such as multiplication in place of raising to a power, are also easy to identify, and thus it does not cost much to check for even these more unusual cases (which are sometimes ignored in compilers). For example, an invariant $v=c_1 \cdot c_2^n$, for c_1, c_2 constants with respect to the loop, can be computed by initializing to $v \leftarrow c_1$ and inserting $v \leftarrow c_2 \cdot v$ in the loop in place of any other computations of v.

d) *Replacing test statements by equivalent tests* which use different variables is a common optimization. In this way, we can sometimes remove computations used only to allow making the original tests. In order to do this, we can inspect whether the invariants from the test, and the invariant which becomes true upon exit from the block, could be obtained by testing other variables. In particular, if an invariant involving a variable linear in the counter is used *only* to allow a test of the counter, there may be other variables also linear in the counter which could be used instead.

Example 1 (continued): From the invariant tree of Fig. 5, it is clear that the test I=1000 and the invariant I=n+1 mean that we are testing whether n=999. Since after doing level 2 optimization, I=n+1 is used *only* in this test, we check whether another variable could be used instead, and see that S=L+2+n. Thus n=999 is equivalent to S-(L+2)=999; i.e., to S=L+1001. We may test S=L+1001, and remove the iteration of I, which is now unnecessary (see Fig. 9).

Since L=L(0), L+1001 can be identified as a constant ex-

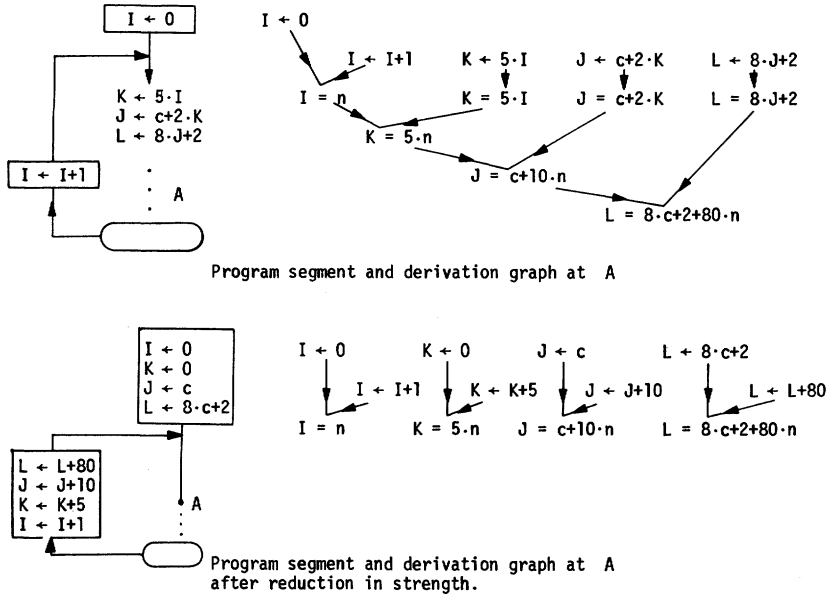


Fig. 8. Reduction in strength example.

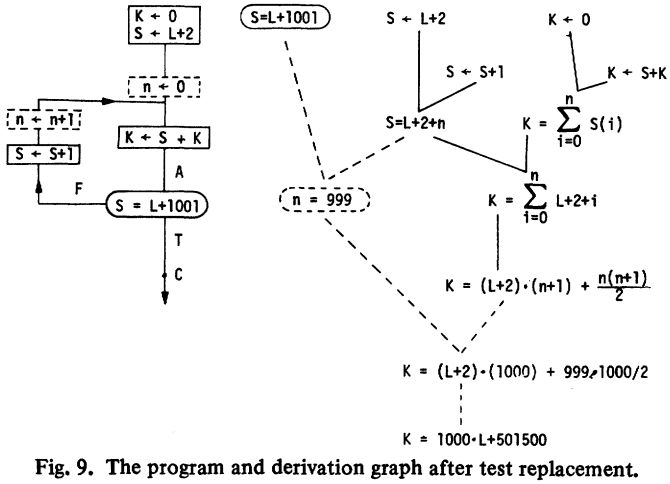


Fig. 9. The program and derivation graph after test replacement.

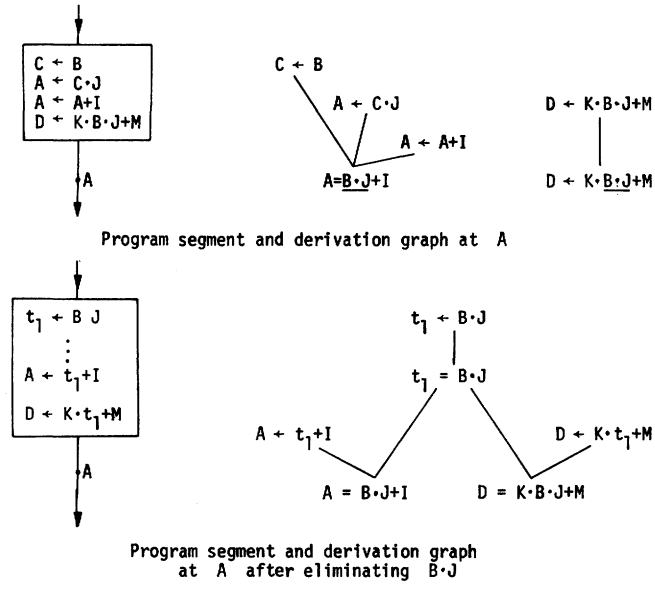


Fig. 10. Subexpression elimination example.

pression, using optimization b), and be replaced by t, with an invariant t=L+1001 at A and at C, having as its ancestor the assignment t ← L+1001 before the loop.

e) *Common subexpression elimination* involves introducing new temporary variables which are equal in value to subexpressions which appear in several statements (or several times in one statement). In our framework, common subexpressions in *invariants* are eliminated even if the statements used to derive the invariants with the common subexpression do not look similar. Using the invariants at the cutpoints also avoids the problem of a special algorithm to guarantee that statements which appear to have identical subexpressions actually do not (because some of the variables involved were changed between the statements containing the subexpressions). The information from pattern matching which identifies common subexpressions would be available from the invariant-generating process, since during that process an attempt is made to combine algebraically invariants into new invariants by eliminating such expressions.

The temporary variable is computed before the point where the subexpression is first used, and replaces all uses of the subexpression. The computation of the temporary variable can then sometimes be further optimized using a)-d). An example is shown in Fig. 10.

IV. Proof-Linked Optimizations

So far, some well-known compiler optimization techniques have been rephrased in terms of invariants. However, it is sometimes possible to perform optimizations which are impossible unless the program specification and its proof are available. This occurs in three situations.

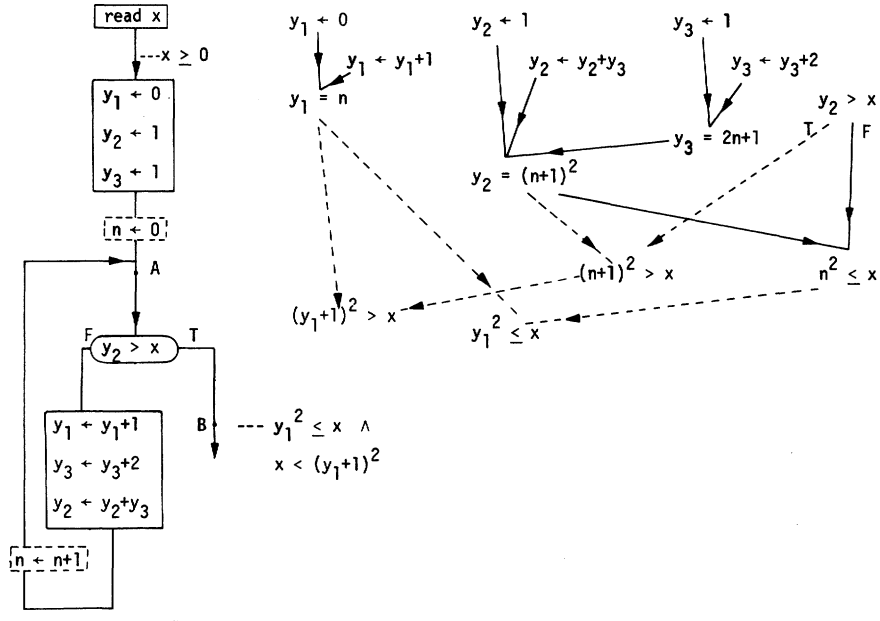


Fig. 11. Integer square-root program and derivation graph.

1) The output specification is "loose;" i.e., could be satisfied by many values. The original (correct) program will compute one of these values. However, it may be possible to compute another of these acceptable values more efficiently. The danger of such optimizations is, of course, that the programmer really wanted the value computed by the original program, but gave too vague a specification. Thus this class of optimizations should only be utilized under some sort of "approval" from the user. Such situations can arise in numerical algorithms where the result need only be within some specified range (say, within ϵ of an actual root), or in a sorting algorithm, where the final order of equally valued elements is unimportant.

2) An essential invariant q at some level is recognized as being too strong for the output specification. That is, a weaker (more general) invariant q′ of the same form as the original q suffices to establish the needed invariants at a lower level. The weaker q′ can then be substituted for q, and the ancestors modified accordingly.

As an extreme example, if $y = (r+z)/2 \wedge r<z$ are invariants, but only $r \leqslant y<z$ is really needed to prove correctness, then any more convenient relation which fulfills the needed inequalities could be used in place of the invariants. One such possible invariant, $r = y \wedge r<z$, would probably significantly simplify the ancestor statements. Although these optimizations could theoretically involve radically changing the algorithm, only relatively obvious cases could probably be recognized automatically.

3) The input specification is "tight"; i.e., a general algorithm is used with inputs guaranteed to fulfill additional properties to those really required by the algorithm. This is a common situation, and often allows optimizations. For example, a program for matrix manipulation might have special tests and treatment for empty matrices, or matrices of one element, while if the input specification guarantees that $m \geqslant 2$ and $n \geqslant 2$ for an $m \times n$ matrix, the special sections and the tests can be removed. In general, such optimizations will be done automatically using the elimination criterion, since the statements will be seen to be unnecessary to prove the essential invariants, and will be removed.

Optimizations dependent on the proof or the specification are naturally sensitive to the particular proof used to show correctness. Optimizations which are more "tailor-made" to the specific case are correspondingly harder to discover systematically.

Example 2: Consider the integer square-root program and its derivation graph seen in Fig. 11. Again, the solid lines are the graph at A, and the dotted lines the graph at B. This program is a well-known example whose invariants can be found by automatic analysis. An idea similar to that of optimization d) can be applied to optimize by pushing computation **forward** out of the loop past its exit test. This is generally not done in standard optimizations. From the graph, it is easy to see that the invariant $y_1=n$ is really only needed after the loop in order to help establish that the specifications are invariants. If $y_1=n$ could be made an invariant after the loop in some other way, its computation inside the loop could be eliminated. But $y_3=2n+1$ is another invariant, and shows that y_3 is linear in n. Solving that invariant for n, it is clear that $n=(y_3-1)/2$ both at A and B. Thus $y_1=n$ can be alternatively achieved by inserting the assignment

$$y_1 \leftarrow (y_3-1)/2$$

before B. (Since $y_3=2n+1$, clearly y_3-1 is even and the above "division" can be accomplished by a simple shift-right operation on a binary computer.) The previous assignments to y_1 would then be removed by the elimination criterion. This optimization is only possible because the role of y_1 in establishing the specification is clearly seen from the graph. The resulting program and its graph are shown in Fig. 12.

Example 3: In this example a simple nested-loop program is

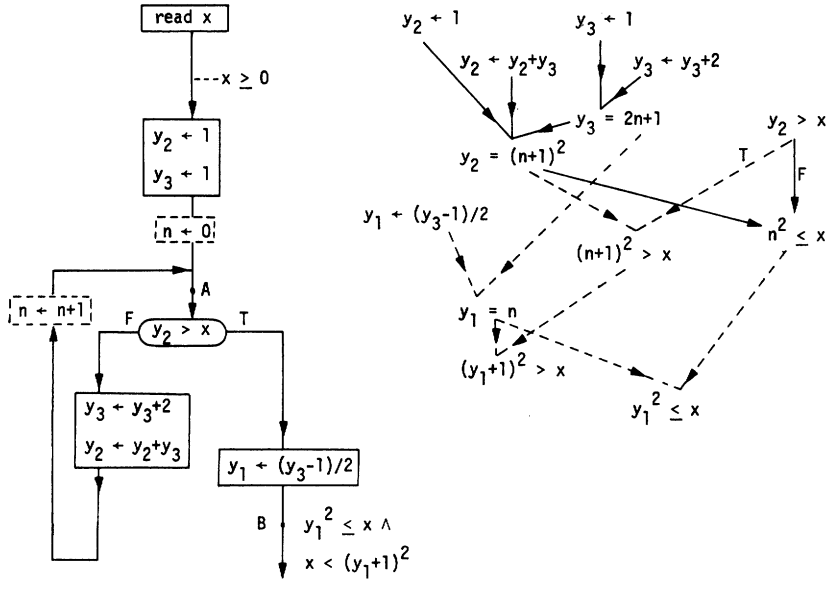
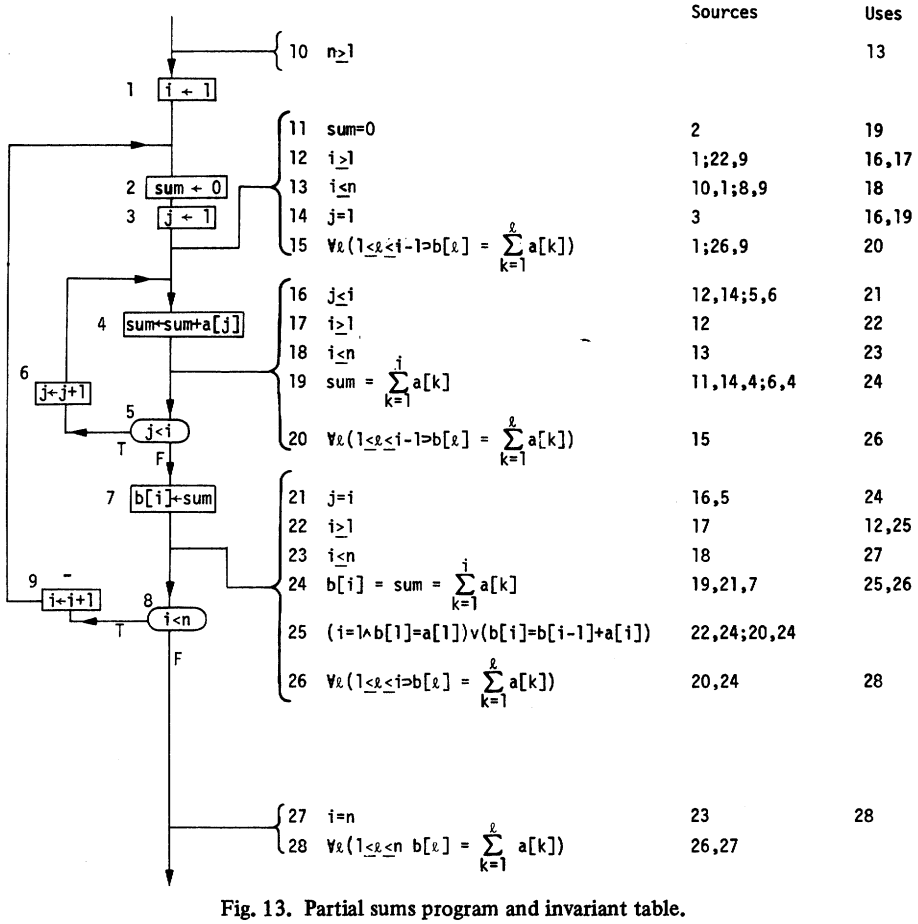


Fig. 12. Program and derivation graph after pushing out of loop.



		Sources	Uses
10	$n \geq 1$		13
11	sum=0	2	19
12	$i \geq 1$	1;22,9	16,17
13	$i \leq n$	10,1;8,9	18
14	j=1	3	16,19
15	$\forall \ell(1 \leq \ell \leq i-1 \supset b[\ell] = \sum_{k=1}^{\ell} a[k])$	1;26,9	20
16	$j \leq i$	12,14;5,6	21
17	$i \geq 1$	12	22
18	$i \leq n$	13	23
19	$sum = \sum_{k=1}^{i} a[k]$	11,14,4;6,4	24
20	$\forall \ell(1 \leq \ell \leq i-1 \supset b[\ell] = \sum_{k=1}^{\ell} a[k])$	15	26
21	j=i	16,5	24
22	$i \geq 1$	17	12,25
23	$i \leq n$	18	27
24	$b[i] = sum = \sum_{k=1}^{i} a[k]$	19,21,7	25,26
25	$(i=1 \wedge b[1]=a[1]) \vee (b[i]=b[i-1]+a[i])$	22,24;20,24	
26	$\forall \ell(1 \leq \ell < i \supset b[\ell] = \sum_{k=1}^{\ell} a[k])$	20,24	28
27	i=n	23	28
28	$\forall \ell(1 \leq \ell \leq n\ b[\ell] = \sum_{k=1}^{\ell} a[k])$	26,27	

Fig. 13. Partial sums program and invariant table.

considered. More than one cutpoint is usually required in order to prove correctness of programs with nested loops. In effect, at each cutpoint an entire table of invariants is built up, with interconnections among the entries in each table.

In this situation optimization should be done first on innermost blocks considered as separate entities. Only then should the resultant outer loops be treated. It is often convenient to add cutpoints at the entrances and exits of inner blocks, in order to "isolate" the block from the remainder of the outer loop.

The program in Fig. 13 is an intermediate version with the invariant table of the Pascal segment

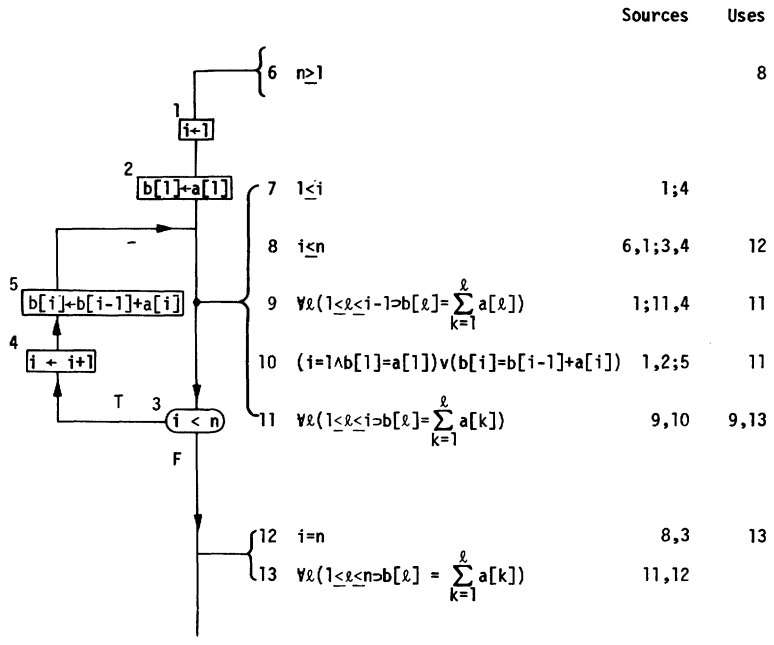


Fig. 14. Optimized partial sums and invariant table.

```
begin for i := 1 to n do
        begin sum := 0;
           for j := 1 to i do sum := sum+a[j];
           b[i] := sum
        end
end
```

The input specification is $n \geq 1 \wedge n \in \{\text{integers}\}$ and a and b are vectors of integers. The segment is intended to compute in b the "partial sums" of a, i.e.,

$$\forall i(1 \leq i \leq n \supset b[i] = \sum_{j=1}^{i} a[j]).$$

The optimization demonstrated on this program illustrates the importance of which proof is used to show correctness.

The invariants listed in Fig. 13 are not particularly difficult to generate. The ones used in the original proof can be obtained either by "pulling" the desired output specification "backward," or by analyzing the difference equations of the inner, and then, the outer loops.

Note that the invariant $b[i] = b[i-1] + a[i]$, which more or less "falls out" of an analysis of the difference equations of the outer loop, is not used. If, however, we substituted the sources 20 and 25 in the proof of invariant 26, (instead of 20 and 24 directly) we would have

$$\forall \ell(1 \leq \ell \leq i-1 \supset b[\ell] = \sum_{k=1}^{\ell} a[k]) \wedge$$

$$((i=1 \wedge b[1] = a[1]) \vee (b[i] = b[i-1] + a[i])) \supset$$

$$b[i] = \sum_{k=1}^{i} a[k] \wedge \forall \ell(1 \leq \ell \leq i \supset b[\ell] = \sum_{k=1}^{\ell} a[k]).$$

This alternative proof still works for the program as it is, and it allows optimization. Simply enough, $(i = 1 \wedge b[1] = a[1]) \vee (b[i] = b[i-1] + a[i])$ can be achieved (much as in reduction in strength) by initializing $b[1]$ to $a[1]$, and then iterating by

$$b[i] \leftarrow b[i-1] + a[i].$$

All the previous ancestors could then be removed. The resulting segment does not need the inner loop at all, and is shown with its proof in Fig. 14.

V. Conclusion

The present paper attempts to link the worlds of program verification and proofs of equivalence between programs with practical considerations of optimizing compilers.

In the context of program verification, Gerhart [9] has expounded the idea of proving systematically that various transformations preserve correctness. Other work in this area has been largely devoted to converting recursive programs to more efficient iterative ones (see [4], [6], [13]). These works do not primarily use the existing proof and invariants to determine the form the transformation will take. That is, the analysis of the program in order to decide whether a certain optimization is applicable is done elsewhere, and the invariants (or some other proof construct) are used only to check that the transformation preserves correctness. The statements which are to be replaced, for example, are not determined by the existing proof, as is done here.

It should be noted that, as is shown in Example 3, an alternative correctness proof of a given program might have a different set of essential invariants, and therefore lead to a different optimization. In some sense, the proof which leads to the greatest gains in execution time will be the most elegant, because the minimum needed information is used at each stage, and extraneous computation is thereby clearly identi-

fied. Even a less than optimal proof should at least allow the optimizations done in standard compilers.

The technique presented here can also be modified to provide formal justification of general transformations by proving that *any* invariants at cutpoints of a given schema will be preserved in another given schema if the invariants and statements satisfy the conditions for applying the transformation.

In the context of the "real" world, the natural question which arises is: "What is gained by basing optimization on invariants, when compilers have been optimizing for years without generating any invariants?"

A few answers to this question follow.

1) Invariants systematize the established techniques because the correctness criterion is clear. "Overzealous" optimizations, which can introduce errors, for example, by moving code out of its context, are easier to avoid since we know exactly what must be maintained, and are able to always check whether we are really maintaining it.

As noted by Allen and Cocke [3], the assurances that a given transformation does not disturb the correctness of the original program are currently built into the algorithm implementing the transformation, and are an ad hoc collection of considerations. Implementors of optimizations have occasionally overlooked problematic situations, and "illegal" optimizations have resulted.

2) Even when it is clear which transformations are legal, information must be gathered from the program in order to ascertain whether the conditions exist which allow applying a transformation. For example, certain transformations require identifying the "induction variables;" i.e., those variables incremented by a constant value at each iteration. It is also valuable to discover variables which are constant (unchanged) in a loop. There are separate algorithms to find each of these and many other characteristics, and the algorithms are often guaranteed to find only relatively obvious cases of whichever characteristic is being considered. Using invariants can make the information-gathering process easier and more uniform.

3) As seen in the previous section, the availability of a correctness proof and the organization based on invariants sometimes allows more radical optimizations than are possible using "blind" transformations. In particular, the optimization can be tailor-made for the information revealed by the proof.

In fact, a closer look at present compiler optimizations will show that they are based on various techniques for finding very specific types of invariants and using them to optimize. Realizing this should help to standardize optimizations and clarify problems in developing more powerful optimizations.

To date, the optimization technique suggested here has not been adequately implemented. It is hoped that this will be rectified as part of logical analysis systems being developed. Of course, it should be recalled that in a logical analysis system the invariants would be available anyway because of their other applications, and so the considerable price of their generation—either manually or automatically—would not be "charged" to any possible optimizations. Until then, the technique is still applicable for hand changes to programs, and as a justification of the transformations method.

References

[1] F. E. Allen, "Program optimization," *Annu. Rev. Automat. Programming*, vol. 5. New York: Pergamon, 1969.

[2] —, "A basis for program optimization," *Proc. IFIPS*, vol. 1 (Ljublijana, Yugoslavia, Aug. 1971).

[3] F. E. Allen and J. Cocke, "A catalogue of optimizing transformations," in *Design and Optimization of Compilers*, R. Rustin, Ed. Englewood Cliffs, NJ: Prentice-Hall, 1972, pp. 1–30.

[4] R. Burstall and J. Darlington, "A transformation system for developing recursive programs," *J. Ass. Comput. Mach.*, vol. 24, pp. 44–67, Jan. 1977.

[5] T. E. Cheatham and B. Wegbreit, "A laboratory for the study of automating programming," in *Proc. 1972 Spring Joint Computer Conf.*, pp. 11–20.

[6] J. Darlington and R. Burstall, "A system which automatically improves programs," in *Proc. 3rd Int. Conf. Artificial Intelligence* (Stanford, CA, 1973), pp. 479–485.

[7] N. Dershowitz and Z. Manna, "Inference rules for program annotation," Stanford AI Lab. Memo AIM-303 or STAN-CS-77-631, Oct. 1977.

[8] B. Elspas, "The semiautomatic generation of inductive assertions for proving program correctness," SRI, Menlo Park CA, Res. Rep., July 1974.

[9] S. Gerhart, "Correctness-preserving program transformations," in *Proc. 2nd ACM Symp. Principles of Programming Languages* (Palo Alto, CA, Jan. 1975), pp. 54–65.

[10] S. Katz, "Logical analysis and invariants of programs," Ph.D. thesis, Weizmann Institute of Science, Rehovot, Israel, Aug. 1976.

[11] S. Katz and Z. Manna, "A heuristic approach to program verification," in *Proc. 3rd Int. Conf. Artificial Intelligence* (Stanford, CA, 1973), pp. 500–512.

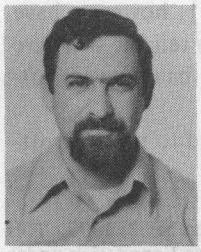
[12] —, "Logical analysis of programs," *Commun. Ass. Comput. Mach.*, vol. 19, pp. 188–206, Apr. 1976.

[13] D. Knuth, "Structured programming with GØTØ statements," *Ass. Comput. Mach. Computing Surveys*, vol. 6, Dec. 1974.

[14] Z. Manna, "The correctness of programs," *J. Comput. Syst. Sci.*, vol. 3, pp. 119–127, May 1969.

[15] B. Wegbreit, "The synthesis of loop predicates," *Commun. Ass. Comput. Mach.*, vol. 17, pp. 102–112, Feb. 1974.

[16] N. Wirth, *Systematic Programming*. Englewood Cliffs, NJ: Prentice-Hall, 1973.



Shmuel Katz was born in Los Angeles, CA, in 1945. He received the B.A. degree from the University of California, Los Angeles, in 1968, with majors in mathematics and English literature, the M.Sc. and Ph.D. degrees in computer sciences from the Weizmann Institute of Science, Rehovot, Israel, in 1972 and 1976, respectively.

Since 1976 he has been a Researcher at the IBM Scientific Center in Haifa, Israel, and has taught courses at the Technion (Israel Institute of Technology) in data structures and program verification. During 1977 to 1978, he was on a leave of absence at the University of California, Berkeley. His interests include topics in program verification, data structures, and programming languages. His research has centered on the discovery and application of invariants, the nature of incorrect programs and methods for correcting them, the systematic selection of data-structure representations, and the correctness of data-structure implementations.