PROGRAM OPTIMIZATION USING INVARIANTS

by

Shmuel Katz

IBM Israel Scientific Center
Technion City, Haifa, Israel

ABSTRACT

Optimizing a computer program is defined as improving the execution time without disturbing the correctness. We show how to use invariants generated from the program 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.

1. INTRODUCTION

For many years compilers have contained sections which are supposed to "optimize" the code produced from computer programs. As has been often 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 control structure of the original program usually will be left intact, and 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 presently 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, and are typically applied to an intermediate-level program (similar in complexity to a flowchart language), before actual machine code is generated.

In this paper, a method is presented for performing optimizations of the above type with the aid of a proof of correctness of the program. That is, in addition to the program, the user has provided (i) an input specification, defining the acceptable input values which the program is intended to treat, and (ii) an output specification

defining the desired relationship between the input and the output. Then, automatically or by hand, intermediate 'invariant assertions' have been attached to prechosen points in the program. These assertions allow proving the correctness of the program w.r.t. 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* 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.

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:

(i) The program may have been developed in a top-down manner by stepwise refinements using structured programming techniques (see, e.g., Wirth [1973]). We assume a stage at which a complete but unoptimized program has been obtained, and that at each stage the correctness has been proven 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.

(ii) The program (either developed by stepwise refinements, or in any more haphazard manner) may have undergone logical analysis, preferably automatically, so that the 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., Cheatham & Wegbreit [1972], Katz & Manna [1976]). 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 proven for the given specification, also using the invariants. Once the program has been proven, a logical analysis system would pass to an optimizer, with the invariants already organized as indicated in the following section.

In either of the above cases, 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 2, some definitions and facts about invariants are presented, and their organization for the purpose of optimization is described. Section 3 examines some specific optimizations

---

* 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.

which are facilitated by using invariants. In the conclusion (Section 4), the techniques described below are compared with the usual optimizations found in compilers, and with related work on transformations which preserve correctness.

2. INVARIANTS AND THEIR ORGANIZATION FOR OPTIMIZATION

First some definitions and facts related to invariants are briefly summarized. The presentation follows Katz [1976].

For convenience of explanation, blocked programs are assumed (although this is not really necessary 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 Allen [1971]. Every "structured program" e.g., program without goto statements (see Wirth [1973]) can be decomposed into such blocks.

The block structure allows treating the program first by considering inner blocks (ignoring momentarily that they are included in outer blocks), and working outwards. Thus for each block its top-level loop can be analyzed using information obtained from the inner blocks.

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

Counters attached to each block containing a loop are an essential tool in our techniques. Since each loop has a unique cutpoint, a counter is 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. Experience has shown that this is generally the most convenient choice.

The counters play a crucial role both for generating invariants and for proving termination. They are used to denote relations among the number of times various paths have been executed, and also to 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 can 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 will be called 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 w.r.t. $\phi(\bar{x})$ 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 λ from cutpoint i to cutpoint j, we define $R_\lambda(\bar{x},\bar{y})$ as the condition for the path λ to be traversed, and $r_\lambda(\bar{x},\bar{y})$ as the transformation in the $\bar{y}$ values which occurs on path λ. A set S of cutpoints of a program P is said to be complete if for each cutpoint i in S all the cutpoints on any path from START to i are also in S.

We now state a sufficient condition (proven in Manna [1969]) for showing that assertions at a complete set of cutpoints are actually invariants.

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

(a) For every path λ from the START statement to a cutpoint j (which does not contain any other cutpoint)

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

and (b) For every path λ from a cutpoint i to a cutpoint j (which does not contain any other cutpoint)

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

As noted in the Introduction, invariants can either be provided by the user, or be generated directly from the program. Among the main methods for this generation are the solution of difference equations which express the change in the variables for one pass around inner loops. These can be obtained from the path functions $r_\lambda(\bar{x},\bar{y})$. Another technique is to examine what was established by the tests made as the paths are followed. This is in the path condition $R_\lambda(\bar{x},\bar{y})$. 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 desireable 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 using the above Lemma.

The final result of the above processes should be conjunctions of invariants at each cutpoint (and including the output specifications at the HALT-points) which satisfy the Lemma. The detailed justification of the use of these invariants to prove partial

correctness, termination, or incorrectness is beyond the scope of this paper. A deeper look into the invariant-generating techniques, and proving properties of programs with invariants, can be found in, e.g., Elspas [1974], Katz & Manna [1973, 1976], Wegbreit [1974].

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 recording the specific 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 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 acyclic graph having the immediate sources as the fathers of each invariant. At the bottom of the graph is the specification, proven from the invariants. Since we talk about the 'ancestors' and 'sons', using terminology similar to trees, we call this graphic representation an invariant tree.

In the continuation, for clarity we refer to various operations on this tree, 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 proven correct and the tree has been generated, the invariants used in the correctness proof are marked. The basic optimization procedure will then be to 'cut' the tree 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, can be removed from both the tree 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 tree will in effect define a level of optimization. The nearer the cut is to the leaves of the tree (i.e., to the program statements) the more local the optimization, while the nearer the cut is to the roots of the tree (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 roots for which we can optimize.

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

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 Figure 1A. In Figure 1B, we show the invariant tree at A for this segment (in solid lines), generated by using only algorithmic techniques.

For this tree, no other cutpoints inside the loop except A are used. The dotted lines indicate the invariant tree at C after the loop. In Figure 1C, 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 tree would be harder 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.

In the remainder of this example, the tree of Figure 1B is used. By 'cutting' this tree at various levels we obtain differing strengths of optimizations. Considering the line denoted in Figure 1B 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).

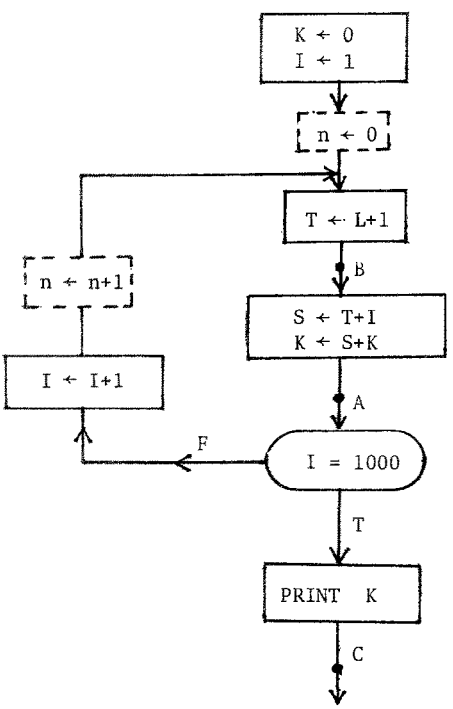


Figure 1A.
Simple intermediate program.

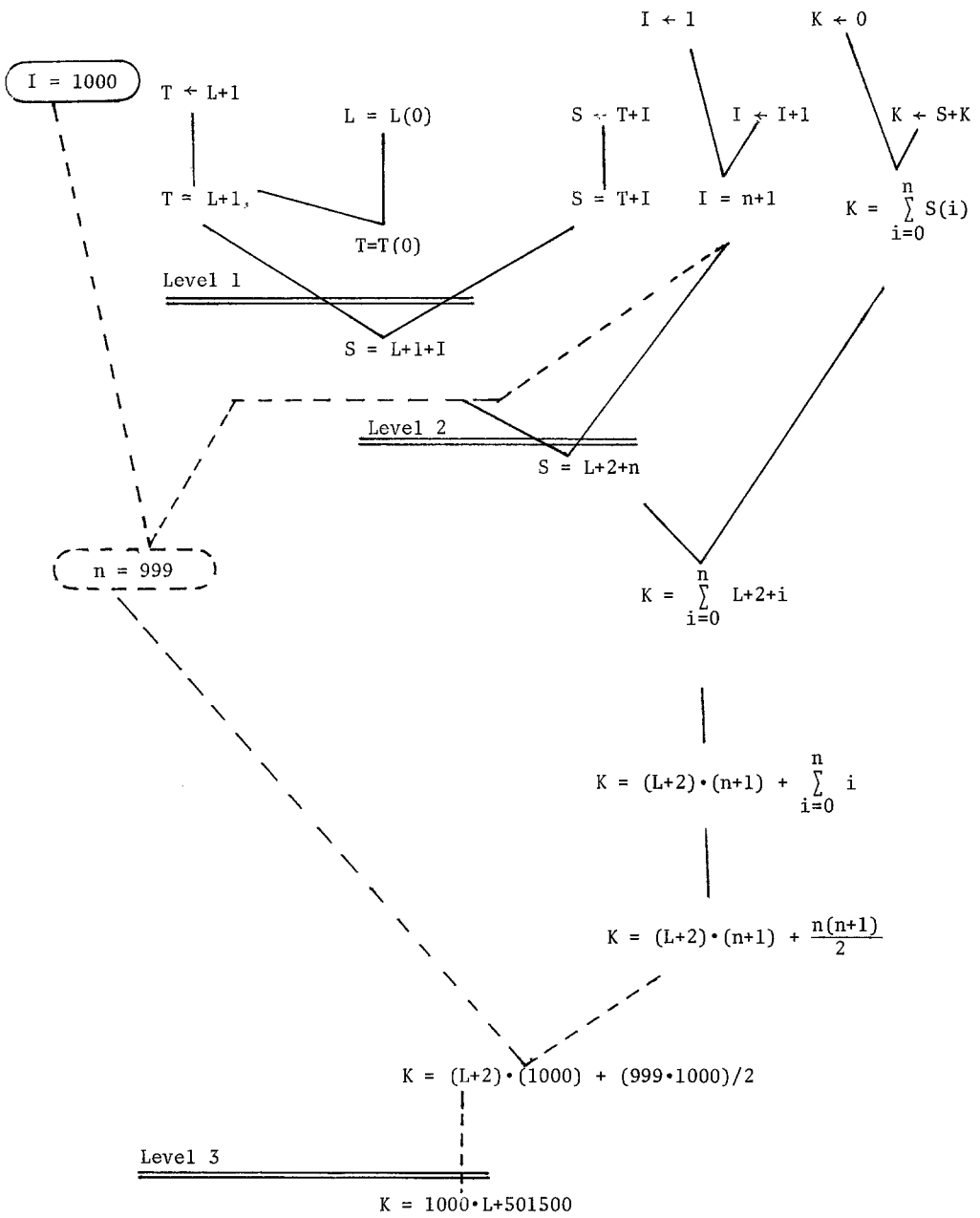
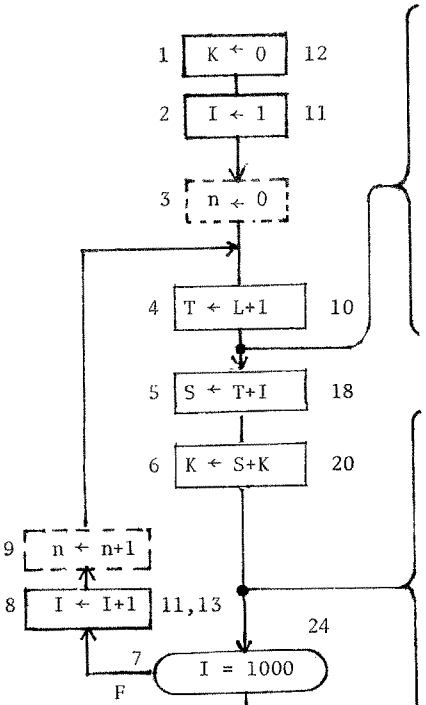


Figure 1B The invariant tree at A and at C.



	Invariants	Sources	Initial y	Uses
10	$T = L+1$	4		17
11	$I = n+1$	19,8	2	19
12	$K = \sum_{i=1}^{n-1} S(i)$	20	1	20
13	$n = 0 \vee S = L+I$	21,8	3	
14	$n = 0 \vee S = L+1+n$	22,11	3	
17	$T = L+1$	10		21
18	$S = T+I$	5		21
19	$I = n+1$	11		22,24
20	$K = \sum_{i=0}^{n} S(i)$	12,6		23
21	$S = L+1+I$	17,18		22,13
22	$S = L+2+n$	19,21		23,14
23	$K = \sum_{i=1}^{n} L+2+i = (L+2)*(n+1) + \frac{n(n+1)}{2}$	22,20		25
24	$n = 999$	7,19		25
25	$K = (L+2)(1000) + 999 \cdot 1000/2$	23,24		26
26	$K = 1000 \cdot L+501500$	25		

Figure 1C. The table for cutpoints A,B and C.

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, 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 Figure 1D, and the new invariant tree in Figure 1E.

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.

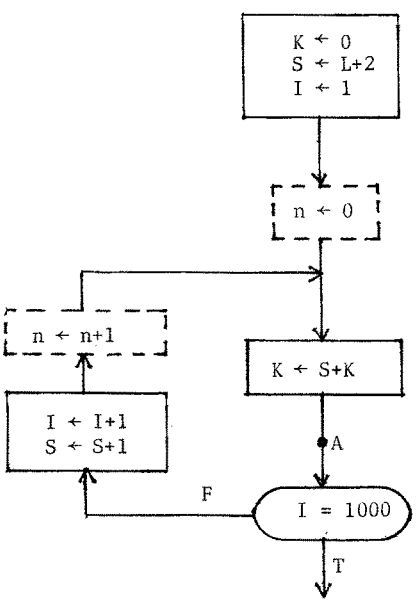


Figure 1D. Program after Level 2 optimization

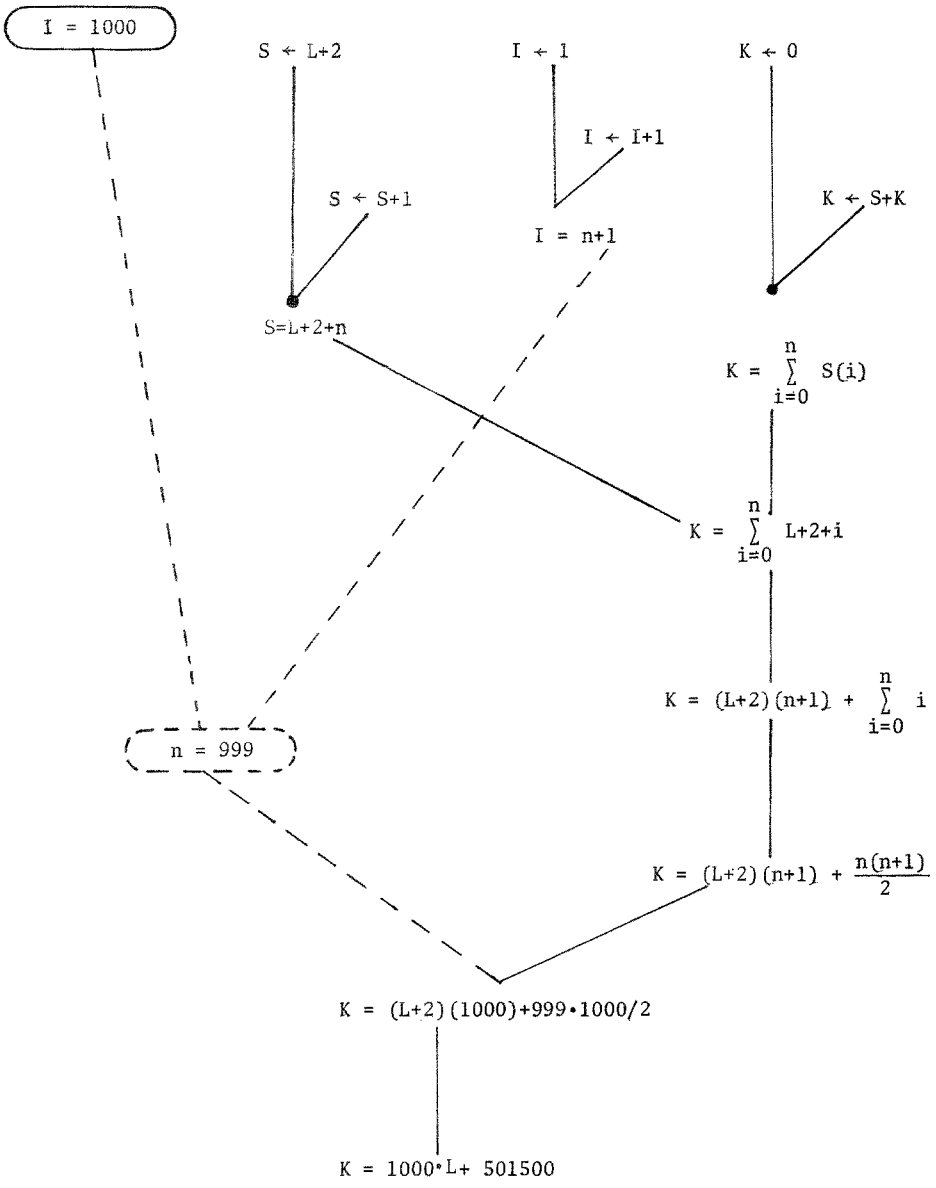


Figure 1E. Invariant tree after level 2 optimization

3. 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 then the various information-gathering algorithms generally employed. At the end of the section optimizations stronger than those possible in standard compilers are described. Note that the optimizations described are not language-dependent.

The basic tool for improving programs using the invariant tree will be eliminating redundant or 'dead' (unused) statements. Simply, any statement not an ancestor of any marked invariant in the invariant tree can immediately be removed from both the tree 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 tree. 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 tree. 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.

Note that a regular optimizer will often eliminate statements which have become syntactically dead (e.g., they cannot be reached in any way). Here statements can be removed which are logically redundant or because of logical relations will not be reached, even if syntactically they appear to be necessary.

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

Then the new generated statements are inserted in the tree as p's ancestors, and the old fathers are disconnnected 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.

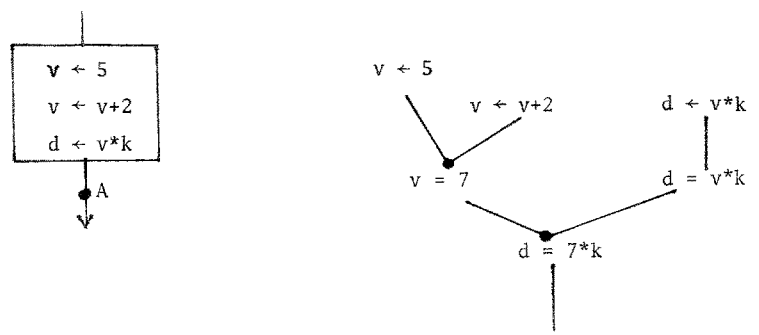
Intuitively, the relatively rare cases where a new variable is added occur when a

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.

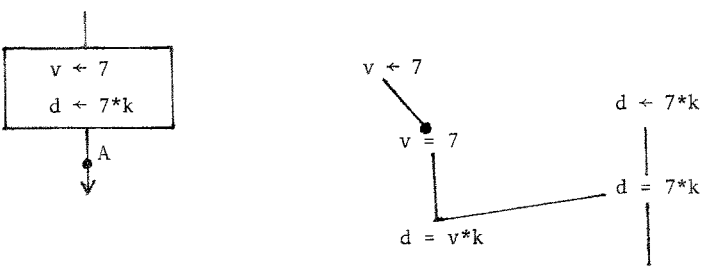
Now some optimizations which lend themselves to the procedure described are examined in more detail. The common compiler optimizations described below loosely follow Allen [1969].

(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 invariant tree, 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.

Example:



Program segment and invariant tree at A



Program segment and invariant tree at A after constant propagation.

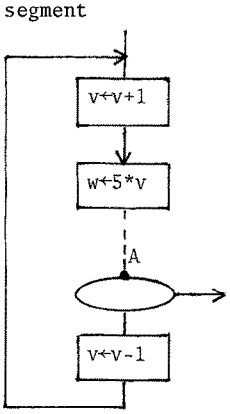
If only d=7*k is used in the proof of correctness, d = v*k and v=7 are extraneous invariants, and they and their ancestor v←7 may be removed.

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

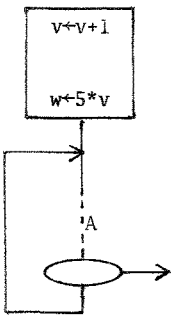
(b) Moving statements outside of loops is probably one of the most beneficial optimizations. We call a variable v constant w.r.t. the loop if an invariant v=v(0) is true at the cutpoint of the loop. Recall that v(0) denotes the value of v the first time the cutpoint is reached (and not necessarily the 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 have the segment



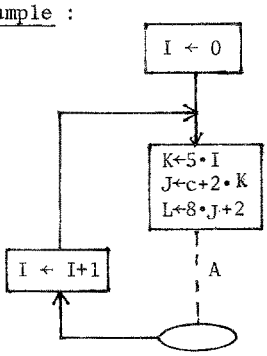
where v and w appear only as indicated and are used only at A, then v=v(0) at A and, entr(v) is v+1. Moreover, w=5*v at A, i.e., w=5*v(0)=w(0). Thus we may add v←v+1 and w←5*v before the loop and remove the original statements (again using the replacement procedure), obtaining

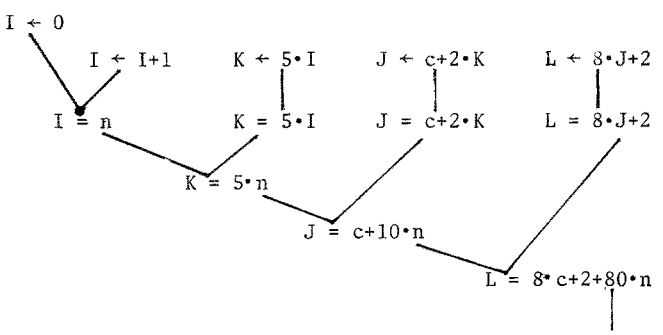


(c) Reduction in strength involves computing variables by using weaker operators in place of stronger one, e.g., addition in place of multiplication.

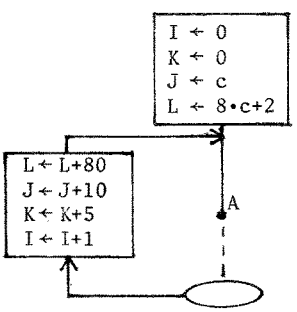
The conditions for applying this optimization are particularly easy to identify by using invariants. Any invariant which connects a variable linearly to the loop counter, e.g., $v=c_1 \cdot n+c_2$, can be computed by initializing to c_2 and then incrementing v by c_1, i.e., inserting $v \leftarrow v+c_1$ 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. More generally, if we have a difference equation $v = c_o$ + if ..... then $c_1 \cdot n$ else $c_2 \cdot n$ then, as in constant propagation, we can initialize to c_o and increment v by c_i on the segment reached if the i-th condition ($i \geq 1$) is true.

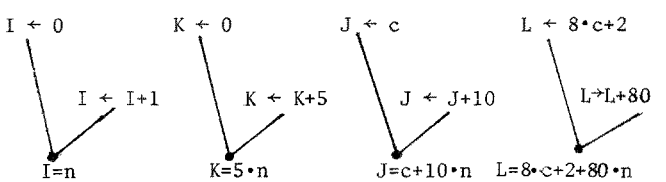
Example :





Program segment and invariant tree at A





Program segment and invariant tree at A after reduction in strength.

Clearly, if, for example,only the invariant involving L were needed in the proof, the invariants and statements with I,K,and J would be removed.

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 w.r.t. 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.

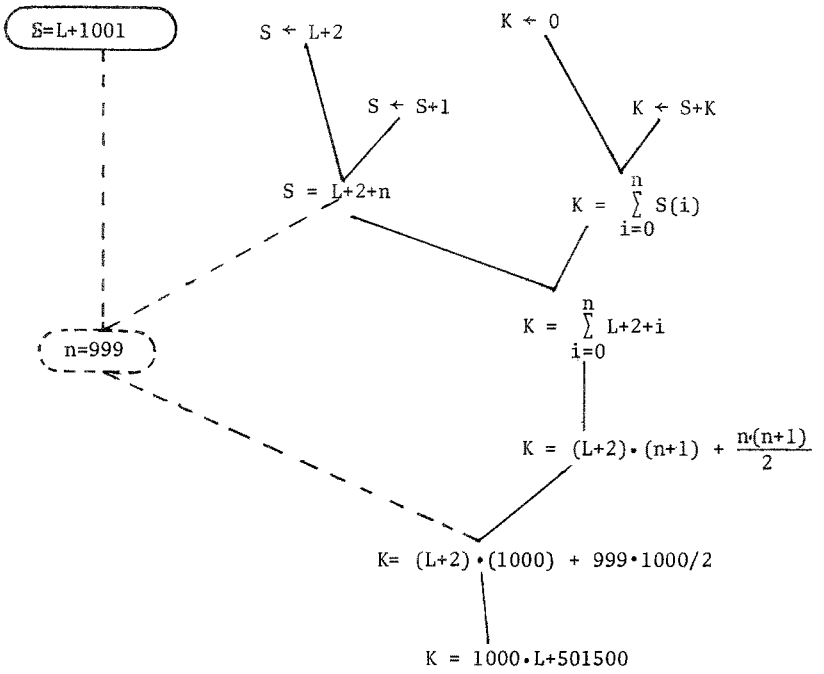
Example 1 (continued)

From the invariant tree of Figure 1E 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 Figure 1F).

Since L=L(0), L+1001 can be identified as a constant expression, 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 including 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).



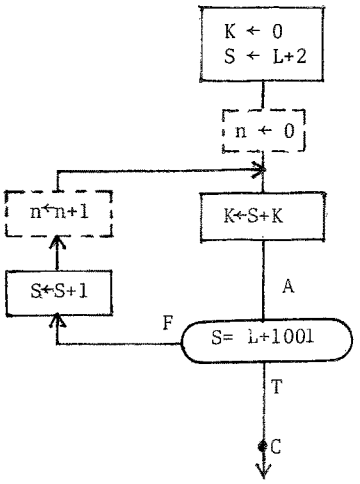
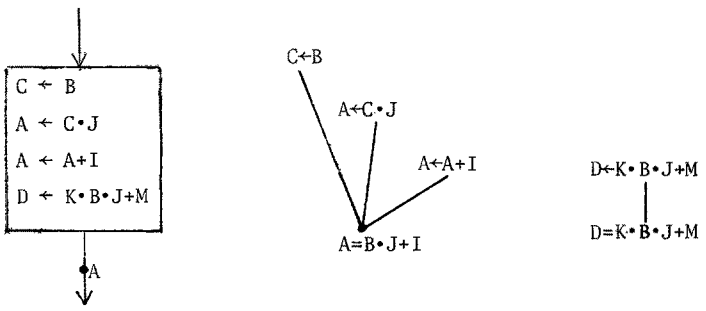
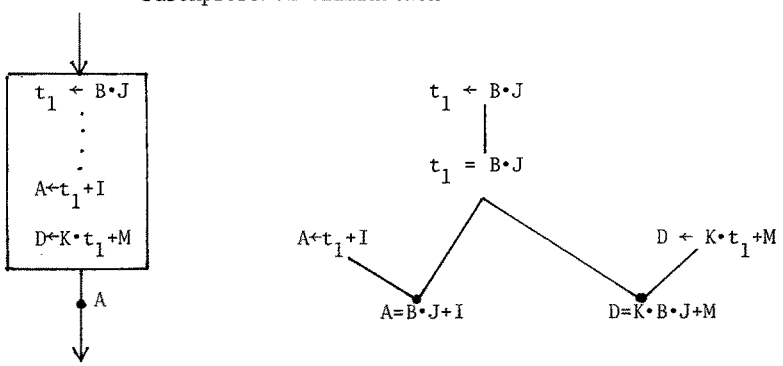


Figure 1F. The invariant tree and program after test replacement

Example:



Program segment and invariant tree at A before subexpression elimination



Program segment and invariant tree at A after eliminating B•J

So far we have discussed how some well-known compiler optimization techniques can be applied by using 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 :

(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 done 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 $r \leq y < z$ 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 possible such 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 $m \geq 2$ and $n \geq 2$ 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.

Example 2. In this example a simple nested-loop program is 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.

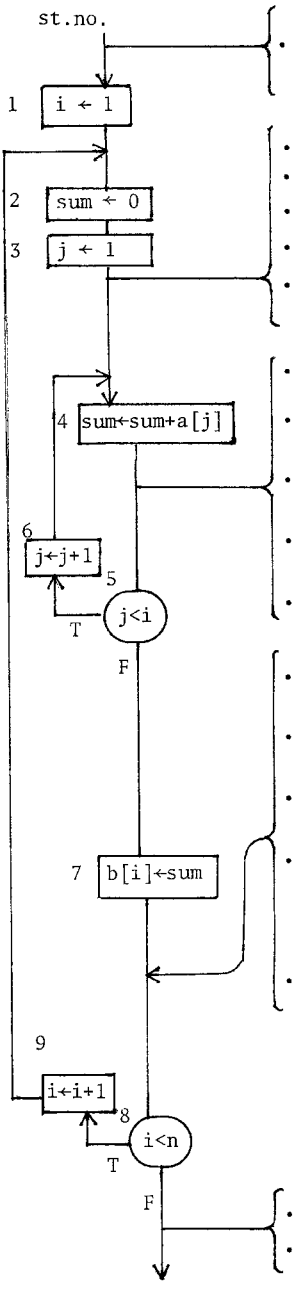
The program in Figure 2A is an intermediate version with the invariant table of the Pascal program

```
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 program 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. Moving from one proof to another is one of the more difficult tasks which we would like an optimizer based on invariants to attempt.



		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	$sum = \sum_{k=1}^{j} a[k]$	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	$\forall \ell(1 \leq \ell \leq i \supset b[\ell] = \sum_{k=1}^{\ell} a[k])$	20,24	28
• 27	i=n	23	28
• 28	$\forall \ell(1 \leq \ell \leq n \supset b[\ell] = \sum_{k=1}^{\ell} a[k])$	26,27	

Figure 2A. Partial sums program and invariant table.

The invariants listed in Figure 2A are not particularly difficult to generate. The ones used in the original proof are marked with a black dot next to the number identifying the invariant, and can be obtained either by 'pulling backwards' the desired output specification, or by analyzing the difference equations of the inner, and then, the outer loops.

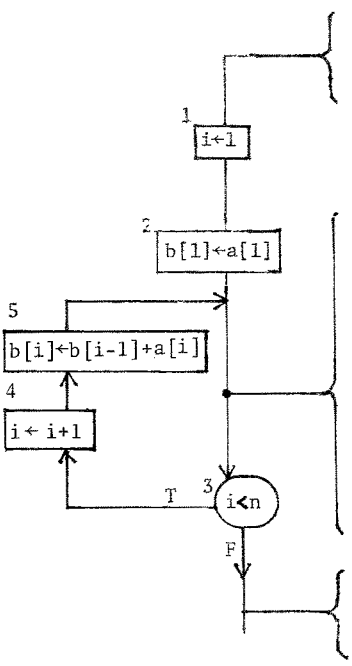
Note that the invariant $b[i] = b[i-1] + a[i]$, which more or less 'falls out' of an analysis of the difference equations of the outer loop, is not used. If, however, we substituted the sources 20 and 25 in the proof of invariant 26, (instead of 20 and 24 directly) we would have

$$\forall \ell(1 \le \ell \le i\text{-}1 \supset b[\ell] = \sum_{k=1}^{\ell} a[k]) \wedge ((i=1 \wedge b[1]=a[1]) \vee (b[i] = b[i\text{-}1] + a[i])) \supset$$
$$b[i] = \sum_{k=1}^{i} a[k] \wedge \forall \ell(1 \le \ell \le i \supset b[\ell] = \sum_{k=1}^{\ell} a[k]).$$

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

$$b[i] \leftarrow b[i\text{-}1] + a[i] \quad .$$

All the previous ancestors could then be removed. The resulting program does not need the inner loop at all, and is shown, with its proof, in Figure 2B.



		Sources	Uses
•6	$n \ge 1$		8
7	$1 \le i$	1;4	
8	$i \le n$	6,1;3,4	12
9	$\forall \ell(1 \le \ell \le i\text{-}1 \supset b[\ell] = \sum_{k=1}^{\ell} a[\ell])$	1;11,4	11
10	$(i=1 \wedge b[1]=a[1]) \vee (b[i]=b[i\text{-}1]+a[i])$	1,2;5	11
11	$\forall \ell(1 \le \ell \le i \supset b[\ell] = \sum_{k=1}^{\ell} a[k])$	9,10	9,13
12	$i=n$	8,3	13
13	$\forall \ell(1 \le \ell \le n \supset b[\ell] = \sum_{k=1}^{\ell} a[k])$	11,12	

2B. Optimized partial sums program with invariant table.

4. CONCLUSION

The present paper attempts to link two worlds: on the one hand, program verification and proofs of equivalence between programs, and on the other hand, practical considerations of optimizing compilers.

In the context of program verification, Gerhart [1975] 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 Darlington and Burstall [1973], Burstall and Darlington [1975], Knuth [1974]). Thses 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 2, an **alternative** correctness proof of a given program might have a different set of essential invariants, and therefore lead to an 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 identified. 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 are:

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 [1972], the assurances that a given transformation does not disturb the correctness of the original program are presently 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. Finally, as noted 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.

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 would not be 'charged' to any possible optimizations.

To date, the optimization technique suggested here has not been implemented. It is hoped (and planned) that this will be rectified as part of logical analysis systems being developed. Until then, the technique is still applicable for hand changes to programs, and as a justification of the transformations method.

REFERENCES:

Allen [1969] Allen F.E.: Program Optimization, Annual Review in Automatic Programming, Vol. 5, Permagon, Elmsford, N.Y., 1969.

Allen [1971] Allen F.E.: A basis for program optimization. Proc. IFIP 1971, Vol. 1, Ljublijana, Yugoslavia (August 1971).

Allen & Cocke [1972] Allen F.E. and Cocke J.: A Catalogue of optimizing transformations in Design and Optimization of Compilers (R. Rustin, ed.), Prentice Hall, 1972, pp.1-30.

Burstall & Darlington [1975] Burstall R., and Darlington J.: Some transformations for developing recursive programs. Proc. International Conference on Reliable Software, Los Angeles, April 1975.

Cheatham & Wegbreit [1972] Cheatham, T.E. and Wegbreit B.: A laboratory for the study of automating programming. Spring Joint Computer Conference, 1972, pp. 11-20.

Darlington & Burstall [1973] Burstall, R. and Darlington, J.: A system for the automatically improves programs. Proc. 3rd Intl. Conf. on Artificial Intelligence. Stanford, 1973, pp. 479-485.

Elspas [1974] Elspas, B.: The semiautomatic generation of inductive assertions for proving program correctness. Research report, SRI, Memlo Park, Calif. (July 1974).

Gerhart [1975] Gerhart, S.: Correctness-Preserving program transformations, Proc. 2nd ACM Symposium on Principles of Programming Languages, Palo Alto, January 1975, pp. 54-65.

Katz [1976] Katz, S.: Logical analysis and invariants of programs, Ph.D. thesis Weizmann Institute of Science, Rechovot, Israel, to appear, 1976

Katz & Manna [1973] Katz, S. and Manna Z.: A Heuristic approach to program verification. Proc. 3rd Intl. Conf. on Artificial Intelligence, Stanford, 1973, pp. 500-512.

Katz & Manna [1976] Katz, S. and Manna Z.: Logical analysis of programs, CACM, to appear, 1976

Knuth [1974] Knuth, D.: Structured Programming with GØTØ statements, ACM Computing Surveys, Vol. 6, No.4, December 1974.

Manna [1969] Manna, Z.: The correctness of programs, J. Computer and System Science, 3, 2, May 1969, pp. 119-127.

Wegbreit [1974] Wegbreit, B.: The synthesis of loop predicates. CACM 17, 2 (February 1974), pp. 102-112.

Wirth [1973] Wirth, N.: Systematic Programming, Prentice-Hall, 1973.