Stanford Artificial Intelligence Laboratory
Memo AIM-294

December 19 7 6

Computer Science Department
Report No. STAN-CS- 7 6-58 6

THE EVOLUTION OF PROGRAMS:
A SYSTEM FOR AUTOMATIC PROGRAM MODIFICATION

Nachum **Dershowitz and Zohar Manna**

Research sponsored by

United States Air Force
and
Advanced Research Projects Agency
ARPA Order No. 2494

COMPUTER SCIENCE DEPARTMENT
Stanford University

Stanford Artificial Intelligence Laboratory **December 19 7 6**
Memo AIM-294

Computer Science Department
Report No. STAN-B-7 6-58 6

THE EVOLUTION OF PROGRAMS: A SYSTEM FOR AUTOMATIC PROGRAM MODIFICATION

Nachum Dershowitz and Zohar Manna

ABSTRACT

An attempt is made to formulate techniques of program modification, whereby a program that achieves one result can be transformed into a new program that uses the same principles to achieve a **different goal.** For example, a program that uses the binary search paradigm to calculate the square-root of a number may be modified to divide two numbers in a similar manner, or vice versa.

Program debugging is considered as a special case of modification: if a program computes wrong results, it must be modified to achieve the **intended** results. The application of abstract program schemata to concrete problems **is also** viewed from the -perspective of modification techniques.

We have embedded this approach in a running implementation; our methods are illustrated with several examples that have been performed by it.

This research was supported in part by the Office of Scientific Research of the United States Air Force under Contract AFOSR -76-2909A and the Advanced Research Projects Agency of the Department of Defense under Contract MDA 903-76-C-0206 . The authors were also affiliated with the Department of Applied Mathematics of the Weizmann Institute of Science during the period of this research.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of Stanford' University of the V. S. Government.

***Reproduced** in the U.S.A. Available from the National Technical information Service, Springfield, Virginia 22161.*

I. INTRODUCTION

Typically, a programmer directs more of his effort at the modification **of** programs that **have** already been written than at the development of original programs. Even when nominally engaged in the construction of a **new** program, he is constantly recycling **"used"** programs and adapting basic programming principles that have already been incorporated into other programs.

Much automatic programming research has focused on the origination of programs, but very little of this work shows how to profit from past experience when approaching a new problem. In this paper, we wish to emulate this latter aspect of programming in the context of an automatic program development system. The essence of our approach lies in the ability to formulate an ***analogy*** between two sets of specifications, those of a program that has already been constructed and those of the program that we desire to construct. This analogy is then used as the basis for transforming the existing program to meet the new specifications.

We consider the debugging process as an important special case of program modification. In our approach, the properties of an incorrect program are compared with the specifications, and a modification (correction) sought that transforms the incorrect program into a correct one.

Abstract program schemata are often a convenient form for incorporating programming **knowledge;** they may embody basic techniques and strategies (such as the

generate-and-test paradigm or the binary search technique). The application of these schemata to programming tasks may also be considered within the framework of modification. A schema which achieves some abstract goal is modified (instantiated) to achieve a concrete goal on the basis of a comparison of the abstract specifications of the schema with the concrete specifications of the desired program.

The use of analogy in problem solving in general, and theorem proving in particular, is discussed by Kling [1971]. The modification of an already existing program to solve a somewhat different task was suggested by Manna and Waldinger [1975] as part of a program synthesis system. Also, the STRIPS (Fikes, Hart and Nilsson[1972]) and HACKER (Sussman[1973]) systems were to some extent capable of generalizing and reusing the robot plans they generated.

The next section elucidates the basic aspects of our approach to program modification with the aid of several relatively straightforward examples. More subtle facets of the techniques are illustrated in the third section. The methods described are amenable to automation, and have been implemented in QLISP (Wilber [1976]). All examples of modifications that we present ran successfully on our system; a sample run may be found in the Appendix.

II. OVERVIEW

Typically, program specifications are expressed in a high-level ***assertion language*** in terms ***of*** an ***output specification*** — detailing the desired relationship between the program variables upon termination, and an ***input specification*** — defining the set of "legal" inputs for which the program is expected to **work.** For program ***modification,*** one is given **a** known correct program with its input-output specification and the specification for **a** new program. Comparison of the two specifications suggests a transformation that is then applied to the given program. Even if the transformed program does not -exactly fulfill the specifications, it can serve as the basis for constructing the desired new program.

1. Basic Technique: Global Transformation

In the approach to program modification presented in this paper, we stress **trapsformations in which** all occurrences of a particular symbol throughout a program **are affected. Such** transformations are termed **"global",** in contrast with "local" **transformations which are** applied only to **a** particular segment of a program.

As a simple example, consider the following program (annotated with its output **specification) :**

```
y ← n
loop until y = 0
        A[y-1] ← if A[2y-1] ≤ A[2y] then A[2y-1] else A[2y] fi
        y ← y-1
        repeat
assert A[0] = min( A[n:2n])   .
```

Given an array A[*n:2n*], which is non-empty (i.e., *n* is non-negative), when this program terminates, *A[0]* will contain the minimum of the values of the *n+1* array elements *A[n], A[n+1],.... A[2n]*. This output specification is formally expressed in the final statement:

assert *A[0] = min(A[n:2n])* .

To modify this program to compute the maximum of the array, rather than the **minimum,** we compare this specification with the desired:

assert *A[0] = max(A[n:2n])*

and note that since *max(A) = -min(-A)* (where *-A* is equal to the array *A* with each element negated), this is equivalent to:

assert *-A[0] = min(-A[n:2n])* .

Thus, the transformation "*A becomes -A*" transforms the given specification into the desired.

Applying this transformation to the program affects only the conditional assignment:

A[y-1] ← if A[2y-1] ≤ A[2y] then *A[2y-1]* else *A[2y]* **fi** ,

which becomes:

-A[y-1] ← if -A[2y-1] ≤ -A[2y] then *-A[2y-1]* else *-A[2y]* **fi** .

It is **"illegal"** for the array $-A$ to appear on the left-hand side of an assignment; therefore, both sides of the assignment are multiplied by -1 . And since the test $-A[2y-1] \leq -A[2y]$ is equivalent to $A[2y] \leq A[2y-1]$, we obtain the statement:

$$A[y-1] \leftarrow \textbf{if } A[2y] \leq A[2y-1] \textbf{ then } A[2y-1] \textbf{ else } A[2y] \textbf{ fi},$$

yielding a program that computes the maximum. Note that the array $-A$ no longer appears in the program; only the original A is actually used.

2. Special Case: Program Debugging

Program ***debugging*** may be considered as a special case of modification: a program which computes wrong results must be modified to compute the desired (correct) results. If'we know what the "bad" program actually does, then we may compare that with the specifications of what it should do, and modify (debug) the incorrect program accordingly.

As **an** example, consider a program intended to compute the integer square-root z of the non-negative integer c , that is, c should lie between the squares of the integers z and $z+1$:

$$\textbf{assert } z^2 \leq c < (z+1)^2, z \in \mathbf{N},$$

where N is the set of natural numbers. The given program is:

$(z, s, t) \leftarrow (1, 0, 3)$
loop until c < s
    (z, s, t) $\leftarrow (z+1, s+t, t+2)$
    repeat
assert $(z-1)^2 \leq c+1 < z^2$, **z**c N^+ .

But rather than computing the integer square-root of c, this program achieves the relation:

assert $(z-1)^2 \leq c+1 < z^2$, **z** c N^+ ,

where N^+ is the set of positive integers. [This follows from the fact that $t=2z+1$ and $s=z^2-1$ throughout.] The cause of the bug was the inadvertent exchange of the initial values of z and s .

Comparing the desired assertion with the actual assertion, we note that the former may be obtained from the latter by replacing z with $z+1$ and c with c-2 . Applying the transformation " c becomes c-7 " to the program statements affects only the exit test c < s , which becomes c-2 < s , or equivalently c ≤ s . The transformation " z becomes $z+1$ " affects two other statements: the initialization z $\leftarrow 1$ becomes $z+1 \leftarrow 1$ and the loop-body assignment z $\leftarrow z+1$ becomes $z+1 \leftarrow z+2$. These resultant assignments, however, are. "illegal", inasmuch as an expression may not appear on the left hand side of an assignment. Instead, the expression $z+1$ is given the initial value 1 by assigning z ← 0 , and the value of the expression $z+1$ is incremented to $z+2$ by the "legal" assignment z • $z+1$.

We have thus obtained the corrected program:

```
(z, s, t) ← (0, 0, 3)
loop until c ≤ s
        (z, s, t) ← (z+1, s+t, t+2)
        repeat
assert z² ≤ c < (z+1)², z ∈ N .
```

Note that though this program is not exactly what the programmer intended — we claimed that he reversed the initial values of z and s — it is nevertheless correct.

3. Correctness Considerations

In the above examples, the transformed programs were correct, i.e., they did in fact satisfy the transformed specifications. This is not necessarily the case with any transformation. Suppose, for example, that we are given the program:

```
(z, y) ← (A[0], 0)
loop until y = n
        y ← y+1
        z ← min(z,A[y])
        repeat
assert z = min(A[0:n])
```

for finding the minimum of the array $A[0:n]$, and we wish to construct a program to find the maximum of the non-empty array $A[1:n]$. The given program achieves $z = min(A[0:n])$, and the output specification of the desired program is $z = max(A[1:n])$. Thus, the transformations "***min becomes max***" and "0 becomes 1" suggest themselves. Though in this case applying these transformations happens to yield

a correct program, such transformations of a function symbol or constant do not necessarily preserve correctness. Were the function ***min*** not explicitly used in the program, ***e.g.***, if the conditionalstatement:

if $A[y] < z$ then $z \leftarrow A[y]$ fi

were substituted for the assignment:

$z \leftarrow min(z, A[y])$,

then the proposed transformation " ***min becomes max*** " would clearly not work.

It can be shown that global transformations where an input variable is systematically replaced by a function of input variables, or an output variable by a function of output variables — as in the previous examples — always yield a program satisfying the transformed specifications. However, transformations of function, predicate or constant symbols — as in this last example — are not guaranteed to result in a program satisfying the specifications.

Hence, for some transformations, correctness must be verified. In order to prove the correctness of a program, ***invariant assertions*** are commonly utilized. ***Assertions*** are comments which express relationships between the different variables manipulated by the program; they relate to specific points in the program, and are meant to hold for the current values of the variables whenever control passes through the corresponding point., When an assertion has been ***proved*** to be consistent with the code — i.e., the assertion holds for the current values of the variables each time control passes through the point to which the assertion is affixed — then it is said to be ***invariant.*** [All assertions annotating our example programs are indeed invariant.] In particular, the

output ***assertfon,*** associated with the point of termination, **is** invariant if the final values of **the** variables satisfy the assertion; a ***loop assertion,*** attached to the beginning of an iterative loop, is **invariant** if it holds when the loop is first entered, and remains true each subsequent time control passes the beginning of the loop-body. The assertion is termed an ***output invariant*** In the former case, and a ***loop fnvarfant*** in the latter. A program, then, may be considered correct if the output invariant implies that the output **specification is true.**

Recently, invariant **generation** techniques have been developed and implemented (see, **e.g.,** German and Wegbreit **[** 1975 **]** and Katz and Manna **[** 1976 **]). They** allow for the automatic discovery of Invariants which may then be used to prove the correctness or incorrectness of the program. Invariant assertions are essential in our approach to debugging too, as it is necessary to **have** an idea of what the program actually does before **It can be corrected.**

Global transformations are applied to all assertions, as well as to the code. Using these **transformed** assertions, verification conditions' for the new program may be obtained; if **they hold,** then **the new** program is correct. Sometimes, a verification condition that **turns** out **not to hold may,** nevertheless, suggest additional transformations which do **succeed.** Alternatively, a program segment can be synthesized that will establish the **verification condition1** for example, the initialization **of** a loop might **be** synthesized if **the condition for the current** initialization is false.

Returning to the above ***min*** example, the program with its loop assertion append&d, is:

```
(z, y) ← (A[0], 0)
'loop  assert z = min(A[0:y])
       until y = n
       y ← y+1
       z ← min(z,A[y])
       repeat
assert z = min(A[0:n]).
```

After application of the transformations "***min*** becomes ***max*** " and "0 ***becomes 1***", ***we*** obtain:

```
(z, y) ← (A[1], 1)
loop  assert z = max(A[1:y])
       until y = n
       y ← y+1
       z ← max(z,A[y])
       repeat
assert z = max(A[1:n])  .
```

Using the new assertions, the correctness of this ***max*** program may straightforwardly be Shown.

4. An Application: Instantiation of Program Schemata

One important application of our program modification techniques ***is*** the ***instantiation*** of program schemata to obtain concrete programs. A ***program schema*** is a generalized version of some programming strategy and contains abstract predicate, function and constant symbols, in terms of which its input-output relation is specified. This abstract specification may then be matched with a given concrete specification and an

instantiation found that, when applied to the schema, yields the 'desired concrete **program.** **Not** all **instantiations** yield correct programs; therefore, a schema is accompanied by *a* set of ***preconditions*** — 'derived from the schema's verification conditions — which must be fulfilled before the schema may be employed. When satisfied, these conditions will guarantee the correctness of the new program.

As an illustration, consider the following program schema:

$(z, y) \leftarrow (k, j)$
loop assert $P([j{:}y],z), y \in I$
until $y = n$.
$y \leftarrow y+1$
if $\neg P(y,z)$ **then** $z \leftarrow f(y,z)$ **fi**
repeat
assert $P([j{:}n],z)$.

Here $P([u{:}v],w)$ means $(\forall i \in I)(u \leq i \leq v)(P(i,w))$ and I is the set of integers. This schema will achieve the relation $P(i,z)$ for each integer i from j to n.

For this schema to be applicable, the following three preconditions must be satisfied by the predicate P, function f and constants j, k and n:

$$P(j,k) \wedge j \in I$$
$$P([j{:}y],z) \wedge y \in I \wedge y \neq n \wedge \neg P(y+1,z) \Rightarrow P([j{:}y+1],f(y+1,z))$$
$$j \leq n \wedge n \in I \ .$$

The first condition ensures that the loop invariant is initialized properly; the second is sufficient to guarantee that if the invariant held before execution of the loop-body, then **it holds after; and the last** condition secures termination.

Programs for finding the position or value of the minimum/maximum of an array (or of other functions with integer domain, for that matter) are valid instantiations of this schema. For example, say we wish to achieve the output specification $A[0{:}n] \leq x$, in order to find the maximum x of the non-empty array $A[0{:}n]$. Applying our modification technique, we compare $A[0{:}n] \leq x$ with the schema's specification $P([j{:}n],z)$. This suggests letting j be 0, z be x and $P(u,v)$ be $A[u] \leq v$. The transformed preconditions, then, are:

$$A[0] \leq k \wedge 0 \in I$$
$$A[0{:}y] \leq x \;\wedge\; y \in I \;\wedge\; y \neq n \;\wedge\; x < A[y+1] \Rightarrow A[0{:}y+1] \leq f(y+1,x)$$
$$0 \leq n \wedge n \in I \;.$$

The first may be achieved by letting k be $A[0]$; the second by letting $f(u,v)$ be $A[u]$, since $A[0{:}y] \leq x < A[y+1]$ and $A[y+1] \leq A[y+1]$; the last is true by virtue of $A[0{:}n]$ being non-empty.

Applying these transformations, viz.

j *becomes* 0,

k *becomes* $A[0]$,

z *becomes* x,

$f(u,v)$ *becomes* $A[u]$

and $P(u,v)$ *becomes* $A[u] \leq v$,

we obtain the guaranteed correct program:

$(x, y) \leftarrow (A[0], 0)$.
loop **assert** $A[0{:}y] \leq x,\ y \in I$
until y = n
$y \leftarrow y+1$
if $x < A[y]$ then $x \leftarrow A[y]$ fi
repeat
assert $A[0{:}n] \leq x$.

The compilation of a handbook of such schemata has recently been advocated by **Gerhart [1975];** their use in the context of program synthesis has been discussed by Dershowitz and Manna **[1975].**

6. Using Extension

Sometimes, transforming a program or instantiating a schema only achieves some of the conjuncts of the output specification. In such a case, it may be possible to ***extend*** the program to achieve all the desired conjuncts by achieving the missing conjuncts at the onset and maintaining them invariant until the end. Alternatively, code that will achieve the additional conjuncts — without "clobbering" what has already been achieved by the program — could be synthesized and appended at the end.

As an example of the need for extension, consider the case where it is desired that the program above also find the position z , in the array, of the maximum x . We can extend the above program to achieve $x = A[z]$ by maintaining that relation as an invariant throughout the execution of the program. Initially we want $x = A[0] = A[z]$,

so we set z ← 0. When the **then** path is executed, we want x = $A[y] = A[z]$ and assign $z \leftarrow y$; when that path is not taken, x is unchanged and the relation remains true. Thus, when the program terminates, the desired relation x = $A[z]$ will hold.

The extended program is:

$(x, y, z) \leftarrow (A[0], 0, 0)$
loop assert $A[0{:}y] \leq x, y \in I, x = A[z]$
until $y = n$
$y \leftarrow y+1$
if $x < A[y]$ then (x, z) $\leftarrow (A[y], y)$ **fi**
repeat
assert $A[0{:}n] \leq x, x = A[z]$.

III. EXAMPLES

In this section we demonstrate various stages in the evolution of one program. We begin with a · program containing a logical error and then find and apply alternative corrections. An abstract version, which represents an important search method embedded in the program, is then applied and adapted to two other problems. Each, in turn, is modified to apply to a new task.

The examples are outlined in Figure 1. They owe their motivation to Wensley [1959] **and** Dijkstra [1976]. Our modification system has successfully performed the modification steps, including debugging and instantiation, in these examples (sometimes resorting to the user's expertise in theorem' proving). An annotated trace of the first example **may** be found in the Appendix.

Example I: Bad Real Division to Good Real Division

Consider the problem **of** computing the quotient z of two real numbers a and b, where $0 \leq a < b$, within a specified tolerance e, $0 <$ e. In other words, the input specification **is:**

$$0 \leq a < b \ \wedge \ 0 < e,$$

and the output, specification is:

$$z \leq a/b \ \wedge \ a/b < z+e,$$

or equivalently:

$$b \cdot z \leq a \ \wedge \ a < b \cdot (z+e).$$

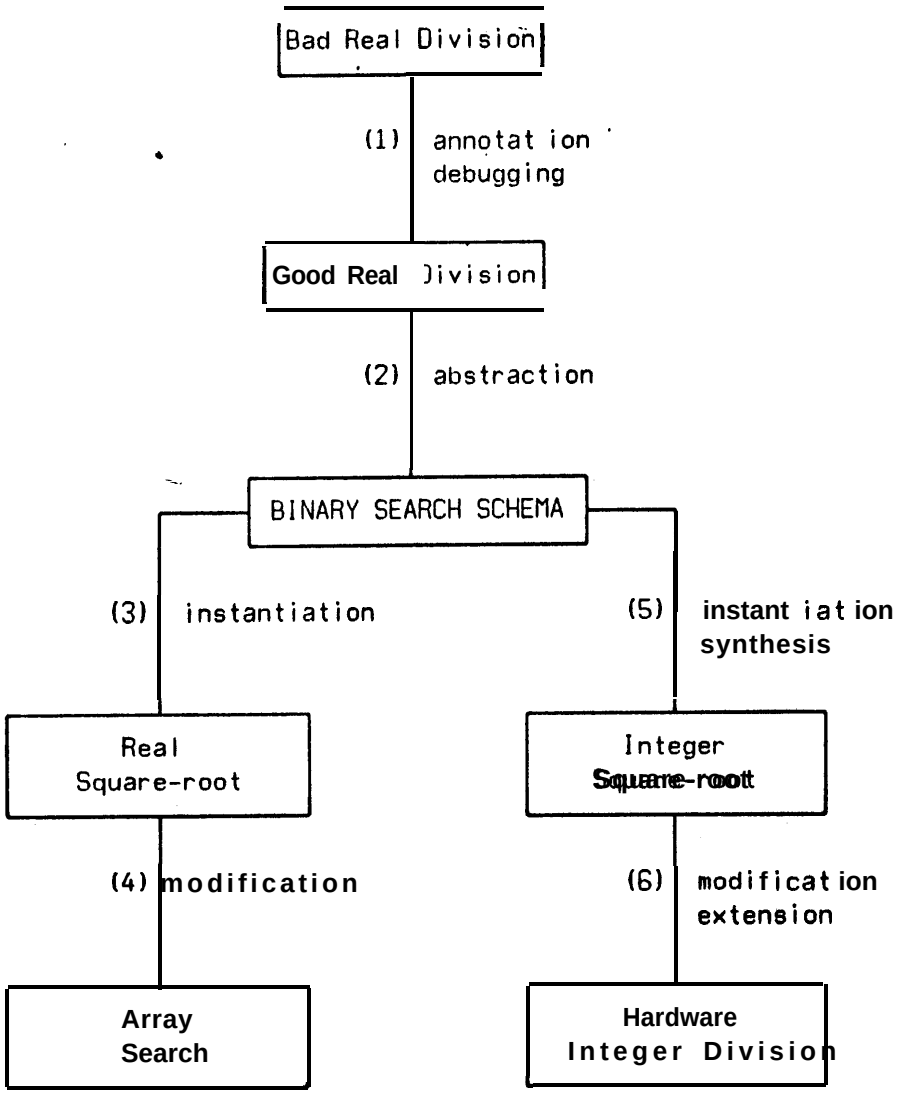

Figure 1. The evolution of a division program.

(Outline of examples 1 through 6.)

In order for the problem to be non-trivial, we must assume that no general real division operator is available (though division by two is permissible). The given program is:

BAD REAL DIVISION PROGRAM

```
assert 0 ≤ a < b, 0 < e
(z, y) ← (0, 1)
loop until y ≤ e
      if b· (zty) ≤ a then z ← z+y fi
      y ← y/2
    repeat
```

The initial assertion contains the input specification which the input variables *a, b* and e are assumed to satisfy. But, for example, *a- 1 , b = 3 ,* and e= *1/3 ,* which satisfy the input specification, yield *z =* 0 which does not satisfy the second conjunct of the output specification. The bug is caused by the interchanging of the two statements within the loop.

Before we can debug this program, we must know more about what it actually does. For this purpose, we annotate the program with loop and output invariants. Recall that for a relation to be a loop invariant, it must be true upon initial entry into the loop, and must **remain** true after each execution of the loop-body.

We begin with the then path of the conditional statement and note that this path is taken when *b· (zty) ≤ a ;* thus, after resetting *z* to *z+y* we have *b· z ≤ a .* Since *b· z ≤* a is true initially, when *z =* 0 *≤* a, and is unaffected when the conditional test is

false (the value of z is not changed), it remains invariant throughout loop execution. We have derived then the loop invariant:

(1) $b \cdot z \leq a$.

The then path is not taken when $a < b \cdot (z+y)$. In that case y is divided in half and z is left unchanged, yielding $a < b \cdot (z+2y)$ at the end of the current iteration. It turns out that the **then** path preserves this relation (the value of $z+2y$ is unchanged), and that it holds upon initiglization (since $a < 2b$ is implied by $0 \leq a < b$). Thus we have the additional invariant:

(2) $a < b \cdot (z+2y)$.

These two loop invariants along with the exit relation y r e imply that upon termination of the program the following output invariants hold:

$$b \cdot z \leq a \ \wedge \ a < b \cdot (z+2e).$$

Note that the desired relation $a < b \cdot$ (zte) is ***not*** implied.

The annotated program — with invariants that correctly express what **the** program ***does*** **do — is:**

ANNOTATED BAD REAL DIVISION PROGRAM

```
assert 0 ≤ a < b, 0 < e
(z, y) ← (0, 1)
loop   assert b·z ≤ a, a < b·(z+2y)
       until y ≤ e
       if b·(z+y) ≤ a then z ← z+y fi
       y ← y/2
       repeat
assert b·z ≤ a, a < b·(z+2e).
```

We **now have** the task of finding a transformation (correction) that transforms the actual, output assertion into the desired output assertion:

assert $b \cdot z \le a, a < b \cdot (z+e)$,

and then applying it to the whole annotated program (statements and invariant assertions). Accordingly, we would like to modify the program in such **a** manner as to transform the insufficiently strong $a < b \cdot (z+2e)$ into the desired $a < b \cdot (z+e)$:

$a < b \cdot (z+2e)$ *becomes* $a < b \cdot (z+e)$.

At the same time, we must preserve the correctness of the other conjunct of the **specification:**

$b \cdot z \le a$ *unchanged.*

The most obvious correction is to replace all occurrences of e in the program (there is only one affected statement - the exit test y ≤ e) with $e/2$:

Correction 1

Replace the exit test $y \le e$ *by* $y \le e/2$.

Additional debugging modifications are possible: we may replace b with $b/2$ and z with $2z$; alternatively, we might replace a with $2a$ and z with $2z$. Doubling z and either halving b or doubling a in the conditional test $b \cdot (zty) \leq a$ yields a test equivalent to $b \cdot (z+y/2) \leq a$. Transforming z into $2z$ affects two additional statements: the initialization z ← 0 becomes the "illegal" assignment $2z \leftarrow 0$, but the equivalent original assignment z ← 0 may be substituted; the assignment z ← $z+y$ of the **then** branch becomes $2z \leftarrow 2z+y$, or z ← $z+y/2$. No other statements are affected by either of the two modifications; thus they both yield:

Correction 2

> ***Replace the conditional statement with***
> **if** $b \cdot (z+y/2) \leq a$ then z ← $z+y/2$ **fi** .

Each of these possible transformations involved one of the input variables e , a and b. One must, however, be careful when transforming input variables, since the transformation should be applied to the input assertion as well, possibly changing the range of legal inputs thereby. In this case, the transformations we have performed are all permissible: The specification 0 < e is equivalent to 0 < e/2 and therefore halving e has no effect on the input range. Since in fact the condition $a < 2b$, rather than $a < b$, is strong' enough to imply the loop invariants, replacing b by $b/2$ (or a by $2a$) still yields a program correct for inputs satisfying $a < b$, as is desired.

Our program after **correction** 2, annotated with appropriately modified invariant

assertions is (all *b* have been replaced by *b/2* and all z **by** *2z* and the resultant expressions have been simplified) :

```
assert 0 ≤ a < b, 0 < e
(z, y) ← (0, 1)
loop  assert b·z ≤ a, a < b·(z+y)
      until y ≤ e
      if b·(z+y/2) ≤ a then z ← z+y/2 fi
      y ← y/2
      repeat
assert b·z ≤ a, a < b· (zte)
```

This program may be slightly optimized, by evaluating the subexpression *y/2* before the conditional statement, to obtain:

GOOD REAL DIVISION PROGRAM

```
assert 0 ≤ a < b, 0 < e
(z, y) ← (0, 1)
loop  assert b·z ≤ a, a < b· (zty)
      until y ≤ e
      y ← y/2
      if b·(z+y) ≤ a then z ← zty fi
      repeat
assert b·z ≤ a, a < b·(z+e).
```

Note that this program is the same **as** the original bad program, with the two loop-body **statements** commuted.

Example 2: Good Real Division to Binary Search Schema

Consider an abstract version of the correct real division program which has just been obtained:

BINARY SEARCH SCHEMA

```
(z, y) ← (j, k)
loop assert P(z), Q(z+y)
       until R(y)
       y ← y/2
       if P(z+y) then z ← z+y fi
       repeat
assert P(z), Q(z+e).
```

This schema is an attempt to capture the technique of binary search underlying the real division program. It is obtained from that program by abstracting predicates that appear in the program text and/or assertions:

$b \cdot u \leq a$ *becomes* $P(u)$,

$a < b \cdot u$ *becomes* $Q(u)$.

and $u \leq e$ *becomes* $R(u)$.

The initial values of the variables are also abstracted:

0 *becomes* j

and 1 *becomes* k.

The following four preconditions on the predicates P, Q and R and constants j and k

are sufficient to guarantee correctness (they correspond to the verification conditions of (1) the initialization path, (2) the loop-body path and (3) the loop-exit path, and (4) termination):

PRECONDITIONS for BINARY SEARCH **SCHEMA**

$$(1) \quad P(j) \wedge Q(j+k)$$
$$(2) \quad \neg P(z+y/2) \Rightarrow Q(z+y/2)$$
$$(3) \quad Q(z+y) \wedge R(y) \Rightarrow Q(z+e)$$
$$(4) \quad (\exists m)(R(k/2^m)) \ .$$

What we have, then, is a general program schema for a binary search within a tolerance . with an output specification:

$P(z) \wedge Q(z+e)$.

Clearly, the predicates ***P*** and ***R*** which appear in the schema must be primitive (that is, **available** in the target language), otherwise they must be replaced by equivalent **predicates for the** schema to yield an executable program. Similarly, the constants j and ***k*** must **be** given, or their values determined, prior to their assignment to the variables **z and y.**

Example 3: Binary Search Schema to Real Square-root

As indicated earlier, one of the applications of our modification system is the

instantiation and adaptation of program schemata to specific problems. To illustrate how the binary search schema that we have just seen may be used, we consider the computation of square-roots.

Suppose that we are given the task of constructing a program that finds the square-root z of the real number c , $1 < c$, within the tolerance d , $0 < d$. The input specification is:

$$0 < d \ \wedge \ 1 < c,$$

and the output specification is:

$$\sqrt{c} \leq z \ \wedge \ z-d < \sqrt{c},$$

that is, the result z may only be greater than the square-root of c by less than the given d .

In order to match this output specification with that of our schema:

$$P(z) \ \wedge \ Q(z+e),$$

we let the constant e be the constant expression $-d$ (viewing $z-d$ as $z+(-d)$) and obtain the transformations:

$P(u)$ *becomes* $\sqrt{c} \leq u,$

$Q(u)$ *becomes* $u < \sqrt{c}$

and e *becomes* $-d.$

Condition (2) is satisfied:

$$(2) \quad \neg(\sqrt{c} \leq z+y/2) \ \Rightarrow \ z+y/2 < \sqrt{c},$$

but we must still satisfy conditions (1), (3) and (4). To satisfy condition (1), we need j and k such that:

(1) $\sqrt{c} \le j \wedge j+k < \sqrt{c}$.

We note **that** since $1 < c$, we have $\sqrt{c} < c$ and $c+(Z-c) = 1 < \sqrt{c}$. Thus both **conjuncts hold when we let:**

j *be* *c*

and *k* *be* *l - c*.

[An alternative would have been to take $-c$ for k, since $c+(-c) = 0 < \sqrt{c}$.]

For condition (3) to be satisfied, we need **a** predicate R such that:

(3) $z+y < \sqrt{c} \wedge R(y) \Rightarrow z-d < \sqrt{c}$.

By **transitivity** it **follows** that R should imply $z-d \le z+y$ and we let:

$R(y)$ *be* $-d \le y$.

. This also satisfies:

(4) $(\exists m)(-d \le (1-c)/2^m)$,

since $-d$ is negative.

The instantiated schema is:

assert $0 < d, 1 < c$
$(z, y) \leftarrow (c, 1-c)$
loop **assert** $\sqrt{c} \le z, z+y < \sqrt{c}$
until $-d \le y$
$y \leftarrow y/2$
if $\sqrt{c} \le zty$ **then** $z \leftarrow z+y$ fi
repeat
assert $\sqrt{c} \le a, z-d < \sqrt{c}$

However, since P involves the square-root function itself, the conditional test is not

primitive and must be replaced. It can be replaced by $c \le (z+y)^2$ provided that c and $z+y$ are non-negative, The relation $0 \le c$ follows from the input specification. And the relation $0 \le z+y$ is in fact an invariant: initially $z+y = c+(Z-c) = 1$; **for the then** path, y is first halved and then added to z , so the value of $z+y$ is unchanged; and if the **then** path is not taken, y is increased by halving it, since y is always negative' (by virtue of the loop assertion $z+y < \sqrt{c} \le z$). Thus we have:

REAL SQUARE-ROOT PROGRAM

```
assert 0 < d, 1 < c
(z, y) ← (c, 1-c)
loop   assert √c ≤ z, z+y < √c, 0 ≤ z+y
       until -d ≤ y
       y ← y/2
       if c ≤ (z+y)^2 then z ← z+y fi
       repeat
assert √c ≤ z, z-d < √c .
```

[We remark that the negative y makes this program appear somewhat unduly complicated; replacing y with $-y$ throughout the program would alleviate this.]

Example 4: Real Square-root to Array Search

In this example, we demonstrate how the above square-root program may be modified to obtain a program that searches for the position z of an element b in an array $A[1:n]$ **that is** sorted in ascending order.

We begin by comparing the output specifications of the desired program with those of the **given** program. We want $z = pos(A,b)$, where $pos(A,b)$ is the position of the element b in the array A. The function pos is the inverse of the array indexing function, ***i.e.,*** **if** $z = pos(A,b)$, then $b = A[z]$. We shall allow indexing the array by real numbers, in which case the array element intended by $A[u]$ is found by truncating u to **an** integer. It is therefore sufficient if:

$$pos(A,b) \leq z \wedge z < pos(A,b)+1.$$

[For simplicity we assume that b appears exactly once in $A[1:n]$. Nevertheless, the program we derive is correct in the more general case where the number of occurrences of b is unspecified>- in that case, pos is extended to yield the (possibly empty) set of positions of b and $z \in pos(A,b)$ is desired.]

For the square-root program we had:

$$\sqrt{c} \leq z \wedge z-d < \sqrt{c}.$$

Comparing the first con juncts suggests the transformation:

$\sqrt{c}$ ***becomes*** $pos(A,b)$;

to obtain this, we can use:

c ***becomes*** $pos(A,b)^2$.

Applying this transformation to the second conjunct of the square-root specification yields $z-d < pos(A,b)$, while we desire $z < pos(A,b)+1$, suggesting the additional transformation:

d ***becomes*** 1.

Applying these transformations to the square-root program, the exit test $-d \le y$ becomes $-1 \le y$. The conditional test $c \le (z+y)^2$ becomes $pos(A,b)^2 \le (z+y)^2$, which is equivalent to $pos(A,b) \le z+y$ (since both $pos(A,b)$ and $z+y$ are non-negative). This contains the non-primitive function pos, but we can test $b \le A[z+y]$ instead (since A **is sorted).** Thus, we have the transformed program:

$(z, y) \leftarrow (pos(A,b)^2, 1-pos(A,b)^2)$
loop **assert** $pos(A,b) \le z,\ z+y < pos(A,b),\ 0 \le z+y$
. **until** $-1 \le y$
$y \leftarrow y/2$
if $b \le A[z+y]$ then $z \leftarrow z+y$ fi
repeat
assert $pos(A,b) \le z,\ z < pos(A,b)+1$.

It is, however, clearly unsatisfactory, since expressions involving pos appear in the initialization. Furthermore, applying the transformation to c in the input assertion $1 < c$ of the square-root program yields $1 < pos(A,b)^2$ which does not hold if $pos(A,b) = 1$. The loop invariant, though, can be initialized in another manner. Since we are given that b appears within the segment $A[1:n]$, we can achieve the relation $pos(A,b) \le z$ by initializing z to n, and we achieve $0 \le z+y < pos(A,b)$ by insisting **that** $z+y = n+y = 0$, for which we initialize y to $-n$. [Replacing the initialization in general **requires** rechecking the termination condition; in this case, for termination $(\exists m)(-1 \le -n/2^m)$ must hold, as indeed it does.]

We **have** obtained the program:

ARRAY' SEARCH PROGRAM

```
assert sorted(A), b ∈ A[1:n]
(z, y) ← (n, -n)
loop  assert pos(A,b) ≤ z, z+y < pos(A,b), 0 ≤ z+y
      until -1 ≤ y
      y ← y/2
      if b ≤ A[z+y] then z ← z+y fi
      repeat
assert pos(A,b) ≤ z, z < pos(A,b)+1
```

Example 5: Binary Search Schema to Integer Square-root

For this example we return to our binary search 'schema:

Preconditions:

(1) $P(j) \wedge Q(j+k)$

(2) $\neg P(z+y/2) \Rightarrow Q(z+y/2)$

(3) $Q(z+y)$ A $R(y) \Rightarrow Q(z+e)$

(4) $(\exists m)(R(k/2^m))$

Schema:

```
(z, y) ← (j, k)
loop assert P(z), Q(z+y)
      until R(y)
      y ← y/2
      if P(z+y) then z ← z+y fi
      repeat
assert P(z), Q(z+e),
```

and illustrate how it may be applied to the computation of integer square-roots. This will necessitate extension and the synthesis of an initialization loop (which have not been completely implemented in our system). "Consequently, this example is more complex than the previous one.

We would like to construct a program that finds the integer square-root z of a non-negative integer c . In other words, z should be the largest integer whose square is not greater than c. Thus, the input specification is:

$$c \in N,$$

and the output specific&ion is:

$$z^2 \leq c \ n \ c < (z+1)^2 \ A \ z \in N \ .$$

Comparison of this output specification with that of our schema:

$$P(z) \wedge Q(z+e),$$

suggests letting:

$$P(u) \quad be \quad u^2 \leq c,$$

$$Q(u) \quad be \quad c < u^2$$

and e *be* *1.*

In addition, we will have to ensure that the final value of z is a non-negative integer.

Clearly, condition (2) is satisfied:

$$(2) \qquad \neg((z+y/2)^2 \leq c) \Rightarrow c < (z+y/2)^2.$$

To satisfy:

$$(3) \qquad c < (z+y)^2 \wedge R(y) \Rightarrow c < (z+1)^2,$$

we let:

$R(y)$ *be* $(z+y)^2 \leq (z+1)^2$.

We are left with the initialization and termination conditions:

(1) $j^2 \leq c$ A $c < (j+k)^2$

(4) $(\exists m)((z+k/2^m)^2 \leq (z+1)^2)$.

In order to satisfy the initialization condition we form the goal:

achieve $j^2 \leq c$, $c < (j+k)^2$.

This conjunctive goal may **be** split into two consecutive ones:

achieve $j^2 \leq c$
achieve $c < (j+k)^2$.

Since c is specified to be non-negative, we can solve the first by letting:

j *be* 0.

i.e., z is initialized to 0 . For the second we need now achieve $c < k^2$.

Our partially written program is:

assert $c \in N$
$z \leftarrow 0$
achieve $c < k^2$
$y \leftarrow k$
loop **assert** $z^2 \le c,\ c < (z+y)^2$
until $(z+y)^2 \le (z+1)^2$
$y \leftarrow y/2$
if $(z+y)^2 \le c$ then $z \leftarrow z+y$ fi
repeat
assert $z^2 \le c,\ c < (z+1)^2$
achieve $z \in N$
assert $z^2 \le c,\ c < (z+1)^2,\ z \in N$.

At this point we have a choice: in order to achieve z c N , either we first execute the loop and then adjust z to satisfy the additional goal z ∈ N while preserving the relationships $z^2 \le c$ and $c < (z+1)^2$ achieved by the loop, or we achieve z ∈ N first and then preserve it throughout the loop computation.

The extension technique suggests preserving z ∈ N throughout loop computation. [This is, in fact, the more efficient of the two choices.] Initially z = 0 ∈ N , but since z is sometimes incremented by y , the latter should also be a non-negative integer. Assuming that z and y are non-negative, the exit test $(z+y)^2 \le (z+1)^2$ can be replaced by $y \le 1$. Furthermore, y is non-zero (since at the start of the loop $0 \le \sqrt{c} < k =$ y and the only operator applied to y is halving), so, under the assumption that y is an **integer, we** need only test for y = 7 .

Finally, in order for y to remain in N while it is repeatedly halved until it equals 1, we must have $y \in 2^N$. So initially, when y = k , we insist that $k \in 2^N$, and accordingly

add the conjunct $k \in 2^N$ to the initialization subgoal $c < k^2$. Note that now, with $k \in 2^N$, the termination condition:

(4) $(\exists m)((z+k/2^m)^2 \leq (z+1)^2)$

clearly holds.

Thus far, we have the partially written program:

```
assert c ∈ N
z ← 0
achieve c < k², k ∈ 2^N
y ← k
loop  assert z² ≤ c, c < (z+y)², z ∈ N, y ∈ 2^N
      until y = 1
      y ← y/2
      if (z+y)² ≤ c then z ← z+y fi
      repeat
assert z² ≤ c, c < (z+1)², z ∈ N .
```

The unachieved subgoal:

```
achieve c < k², k ∈ 2^N
```

must now be synthesized. We would first attempt to achieve this goal one conjunct at a **time.** The first conjunct might be achieved by letting $k = c+1$, while the second could easily be achieved by letting $k = 1$. However, though each conjunct is achievable by **itself in this manner, achieving** both together is more difficult, since these two solutions **in general conflict** with **each** other.

So, we transform this conjunctive goal into an **iterative loop,** choosing first to achieve $k \in 2^n$ by letting $k = 2^0 = 1$, and then to keep it true while executing the loop until the remaining conjunct, $c < k^2$, is also satisfied. 'Within the loop, doubling k with each iteration will preserve the invariant $k \in 2^N$ while making progress towards the exit test $c < k^2$. [The reasoning is as follows: We know that k should be increasing, since initially $k = 1$ and ultimately we want $0 \leq \sqrt{c} < k$. Since we wish $k = 2^n$ for some natural number n to remain invariant while k increases, it follows that the exponent n also increases. Doubling k increments the exponent by 1.]

We have obtained the following initialization:

```
assert c ∈ N
(z, k) ← (0, 1)
loop assert k ∈ 2^N
        until c < k^2
        k ← 2k
        repeat
y ← k
```

Note that the last assignment $y \leftarrow k$ is superfluous; it may be eliminated if we replace all occurrences of k in the code with y. With this change, we have the integer square-root program:

INTEGER SQUARE-ROOT PROGRAM

```
assert c ∈ N
(z, y) ← (0, 1)
loop assert y ∈ 2^N
        until c < y^2
        y ← 2y
        repeat
loop    assert z^2 ≤ c, c < (z+y)^2, z ∈ N, y ∈ 2^N
        until y = 1
        y ← y/2
        if (z+y)^2 ≤ c then z ← z+y fi
        repeat
assert z^2 ≤ c, c < (z+1)^2, z ∈ N .
```

Example 6: Integer Square-root to Hardware Integer Division

We wish to construct a program to compute the quotient q and remainder r of two natural numbers a and b. Such a program could be developed from our binary search **schema** in the same manner as we constructed the integer square-root program. But, instead, we will demonstrate how to transform the just constructed integer square-root **program directly** into the desired integer division program.

The program must satisfy the output specification:

$$0 \le r \;\wedge\; r < b \;\wedge\; q \in N \;\wedge\; a = b \cdot q + r,$$

or equivalently:

$$(*) \qquad q \le a/b \;\wedge\; a/b < q+1 \;\wedge\; q \in N \;\wedge\; r = a - b \cdot q,$$

given the input specification:

$$a \in N \ \wedge \ b \in N^+$$

(N^+ is the set of positive integers). [Rephrasing specifications — so that their similarity with the specifications of another program or schema can be brought out — is a non-trivial problem in its own right. Our system only finds some close variations.]

We compare the output specification (*) with that of the square-root program:

$$z^2 \leq c \ \wedge \ c < (z+1)^2 \ \wedge \ z \in \mathbf{N},$$

or:

$$z \leq \sqrt{c} \ \wedge \ \sqrt{c} < z+1 \ \wedge \ z \in N,$$

and obtain the transformations:

$$z \quad \textit{becomes} \quad q$$

and $\sqrt{c}$ *becomes* a/b.

To obtain the latter, we can use:

$$c \quad \textit{becomes} \quad (a/b)^2,$$

(since a/b is non-negative). In addition we will have to achieve $r = a - b \cdot q$.

Applying these transformations, the exit test of the first loop, $c < y^2$, becomes $(a/b)? < y^2$. Since both a/b and y are non-negative, this is the same as $a/b \ c \ y$ or $a < b \cdot y$. Similarly the conditional *test* $(z+y)^2 \leq c$ becomes $(q+y)^2 \leq (a/b)^2$, or equivalently $b \cdot (q+y) \leq a$.

Thus, we have the program:

$(q, y) \leftarrow (0, 1)$
loop assert $y \in 2^N$
until $a < b \cdot y$
$y \leftarrow 2y$
repeat
loop assert $q \leq a/b,\ a/b < q+y,\ q \in N,\ y \in 2^N$
until $y = 1$
, $y \leftarrow y/2$
if $b \cdot (q+y) \leq a$ **then** $q \leftarrow q+y$ fi
repeat
assert $q \leq a/b, a/b < q+1, q \in N$.

Special attention must **be** paid to the input specification: By applying the transformation " *c becomes* $(a/b)^2$ " to the input assertion of the integer square-root program, the input condition for this program is obtained. We note, however, that the only fact **needed for** the construction of the square-root program was $0 \leq c$; its input specification $c \in N$ **was** unnecessarily restrictive. Applying the transformation to $0 \leq c$ yields $0 \leq (a/b)^2$. Now, since this is implied by the input specification $a \in N \wedge b \in N^+$, the above program is correct for any legal values of a and b.

To achieve the additional output specification $r = a - b \cdot q$, we extend the above program to keep that relation invariantly true. So whenever q is updated, it is necessary to update r accordingly: when q is initialized to 0, $r = a - b \cdot 0 = a$; when q is incremented to $q+y$, r becomes $a - b \cdot (q+y) = r - b \cdot y$,

So far we have:

```
assert a ∈ N, b ∈ N+
(q, y, r) ← (0, 1, a)
loop  assert y ∈ 2^N, r = a-b·q
      until a < b·y
      y ← 2y
      repeat
loop  assert q ≤ a/b, a/b < q+y, q ∈ N, y ∈ 2^N, r = a-b·q
      until y = 1
      y ← y/2.
      if b·(q+y) ≤ a then (q, r) ← (q+y, r-b·y) fi
      repeat
assert q ≤ a/b, a/b < q+1, q ∈ N, r = a-b·q .
```

Note that the conditional test $b\cdot(q+y)\leq a$ is equivalent to $b\cdot y\leq a-b\cdot q$ or $b\cdot y\leq r$. The expression $b\cdot y$ involves multiplication and appears three times, so a new variable u is introduced to always equal $b\cdot y$. Substituting u for all occurrences of $b\cdot y$ and updating u whenever the value of y is changed, we obtain:

HARDWARE INTEGER DIVISION

```
assert a ∈ N, b ∈ N+
(q, y, r, u) ← (0, 1, a, b)
loop  assert y ∈ 2^N, r = a-b·q, u = b·y
      until a < u
      (y, u) ← (2y, 2u)
      repeat
loop  assert q ≤ a/b, a/b < qty, q ∈ N, y ∈ 2^N,
             r = a-b·q, u = b·y
      until y = 1
      (y, u) ← (y/2, u/2)
      if u ≤ r then (q, r) • (q+y, r-u) fi
      r e p e a t
assert q ≤ a/b, a/b < q+1, q ∈ N, r = a-b·q .
```

This then is the desired hardware integer division program. Its only operations are **addition,** subtraction, comparison and shifting, all of which are hardware instructions **on binary computers.**

Note the similarity between the extension and optimization steps in this example. In **both cases a relation was added and** kept invariantly true at all points of the program. Most **of the previous examples** would have profited from similar optimizations.

ACKNOWLEDGEMENT

We thank Richard Waldinger for many fruitful discussions and constructive comments. Computer time was provided by the Artificial Intelligence Center of Stanford Research Institute.

REFERENCES

Dershowitz, N. and Z. Manna *[July* 1975 J, ***On automating structured programming,*** **Proc.** Symp. on Proving and Improving Programs, Arc-et-Senans, France, **pp. 167- 193.**

Dijkstra, E.W. [1976], ***A discipline of programming,*** Prentice Hall, Englewood Cliffs, N.J.

Fikes R.E., P.E. Hart and N.J. Nilsson [Winter **1972],** ***Learning and executing . generalized robot plans,*** Artificial Intelligence, V. 3, No. 4, pp. 251-288.

Gerhart, S.L. [Apr. 1975], ***Knowledge about programs: a model and case study,*** **Proc. Intl.** Conf. ***on*** Reliable Software, Los Angeles, Ca., pp. 88-95.

German, S.M. and B. Wegbreit [Mar. 1975], ***A synthesizer of inductive assertions,*** IEEE Trans. on Software Engineering, V. SE- **1,** No. **1,** pp. 68-73.

Katz, S.M. and Z. Manna [Apr. 1976], ***Logical analysis of programs,*** CACM, **V.** 19, No. 4, **pp. 188-206.**

King, R.E. ***[Aug. 197*** **1], *Reasoning by analogy with applications to heuristic problem solving: a case study,*** Ph.D. thesis, Stanford U., Stanford, Ca.

Manna, Z. and R.J. Waldinger [Summer 1975], ***Knowledge and reasoning in , 'program synthesis,*** Artificial Intelligence, V. 6, No. 2, pp. **175-208.**

Sussman, G.J. [Aug. 1973], ***A computational model of skill acquisition,*** Ph.D. thesis, MIT, Cambridge, Mass.; also published as ***A computer model of skill acquisition,*** Anierican Elsevier, New York, **N.Y.** (1975).

Wensley, J.H. [Jan. 1959], ***A class of non-analytical iterative processes,*** Computer J., V. **1,** No. 4, pp. **163-167.**

Wilber, B.M. [Mar. 1976], ***A QLISP reference manual,*** Tech. note 118, Artificial Intelligence Center, Stanford Research Institute, Menlo Park, **Ca.**

APPENDIX

The following is a QLISP trace of Example 1 (the debugging of the real division program), as executed by our modification system. The steps and expressions differ somewhat from the example as presented in the previous section. The trace has been edited and annotated to enhance its understandability. False leads that the system followed are also included.

The procedure MODIFY modifies a program to achieve a new goal. Here it is used to debug a real division program:

```
MODIFY:
```

This is the annotated-given bad program:

```
((ASSERT (AND (LTQ 0 A) (LT A (TIMES 2 B)) (LT 0 E)))
 (SETQ Z 0) (SETQ Y 1)
 {LOOP (ASSERT (AN@ (LTQ (TIMES B 2) A) (LT A (TIMES B (ADD Z (TIMES 2 Y))))))
       (UNTIL (LTQ Y E))
       (IF (LTQ (TIMES B (ADD Z Y)) A) THEN (SETQ Z (ADD Z Y)) F1)
       (SETQ Y (DIV2 Y))
       REPEAT)
 (ASSERT (AND (LTQ Z (DIV A B)) (LT (DIV A (TIMES 2 B)) (ADD (DIV Z 2) E)))))
```

prefaced by an input assertion, containing the conditions under which the invariants hold, and followed by output invariants. We desire that the program achieve the output specification:

```
(ASSERT (AND (LTQ Z (DIV A B)) (LT (DIV A B) (ADD Z E))))
```

with the legal inputs defined by the following input specification:

```
(ASSERT (AND (LTQ 0 A) (LT A B) (LT 0 E)))
```

Note that this specification differs from the input assertion of the program.

The system begins by applying the function MATCH to compare the output invariant with the desired output specification:

```
MATCH: (AND (LTQ Z (DIV A B)) (LT (DIV A (TIMES 2 B)) (ADD (DIV Z 2) E)))
       '(AND (LTQ Z (DIV A B)) (LT (DIV A B)          (ADD Z       E)))
```

The first conjuncts of both are the same, and the system compares the second conjuncts. It notices that if the expression (TIMES 2 B) could be transformed into B and (DIV Z 2) into Z. then the whole conjunct would transform as desired. So it calls the function INVERT, which suggests the transformation "B becomes (DIV B 2)" for (TIMES 2 B):

```
INVERT: (TRANSFORM (TIMES 2 B) B)
result= (TRANSFORM B (DIV B 2))
```

and similarly for (DIV Z 2):

```
INVERT: (TRANSFORM (DIV Z 2) Z)
result= (TRANSFORM Z (TIMES 2 Z))
```

Thus, the system has found transformation 1:

```
((TRANSFORM B (DIV B 2)) (TRANSFORM Z (TIMES 2 Z)))
```

But first, the system must apply this transformation to the first conjunct:

```
TRANSFORM-EXPRS: (LTQ Z (DIV A B))
result= (LTQ (TIMES 2 Z) (DIV A (DIV B 2)))
```

and prove that the conjunct remains true, i.e.,

```
(IMPLIES (LTQ (TIMES 2 Z) (DIV A (DIV B 2)))
         (LTQ Z (DIV A B)))
```

Before proceeding, the system looks for additional possible transformations. Since ADD is commutative, an attempt is also made to match (ADD (DIV Z 2) E) with (ADD E Z). This, together with (TRANSFORM B (DIV B 2)), yields transformation 2:

```
((TRANSFORM B (DIV B 2)) (TRANSFORM E Z) (TRANSFORM Z (TIMES 2 E)))
```

However, this set of transformations is disqualified, since there is no way to transform the variable Z into the constant expression (TIMES 2 E).

Continuing in its search for alternative transformations, the system also finds equivalent formulations of the specifications, e.g.:

```
(AND (LTQ (TIMES B Z) A) (LT A (ADD (TIMES B Z) (TIMES 2 B E))))
(AND (LTQ (TIMES B Z) A) (LT A (ADD (TIMES B Z) (TIMES B E))))
```

Comparing them yields transformation 3:

```
((TRANSFORM E (DIV E 2)))
```

The system now calls the function TRANSFORM-PROGRAM for each of the two eligible transformations (1 and 3) in turn:

```
TRANSFORM-PROGRAM:
  ((ASSERT (AND (LTQ 0 A) (LT A (TIMES 2 B)) (LT 0 E)))
   (SETQ Z 0) (SETQ Y 1)
   (LOOP (ASSERT (AND (LTQ (TIMES B Z) A) (LT A (TIMES B (ADD Z (TIMES 2 Y))))))
         (UNTIL (LTQ Y E))
         (IF (LTQ (TIMES B (ADD Z Y)) A) THEN (SETQ Z (ADD Z Y)) FI)
         (SETQ Y (DIV2 Y))
         REPEAT)
   (ASSERT (AND (LTQ Z (DIV A B)) (LT (DIV A (TIMES 2 B)) (ADD (DIV Z 2) E)))))
  ((TRANSFORM B (DIV B 2)) (TRANSFORM Z (TIMES 2 Z)))
```

TRANSFORM-CONST-EXPR, which transforms constants, is now called, and B is replaced by (DIV

B 2) throughout:

```
TRANSFORM-CONST-EXPR: (TRANSFORM B (DIV B 2))
```

TRANSFORM-VAR-EXPR transforms a variable, in this case the variable Z becomes (TIMES 2 Z):

```
TRANSFORM-VAR-EXPR: (TRANSFORM Z (TIMES 2 Z))
```

This may entail eliminating expressions from the left-hand side of assignments. The function TRANSFORM-SETQ is used to apply (TRANSFORM Z (TIMES 2 Z)) to all assignments to Z:

```
(SETQ z 0)
result= (SETQ z (DIV 0 2))
```

and:

```
(SETQ Z (ADD Z Y))
result= (SETQ Z (DIV (ADD (TIMES 2 Z) Y) 2))
```

The transformed program is:

```
((ASSERT (AND (LTQ 0 A) (LT A (TIMES 2 (DIV B 2))) (LT 0 E)))
 (SETQ Z (DIV 0 2)) (SETQ Y 1)
 (LOOP (ASSERT (AND (LTQ (TIMES (DIV B 2) (TIMES 2 Z)) A)
                    (LT A (TIMES (DIV B 2) (ADD (TIMES 2 Z) (TIMES 2 Y))))))
       (UNTIL (LTQ Y E))
       (IF (LTQ (TIMES (DIV B 2) (ADD (TIMES 2 Z) Y)) A)
           THEN (SETQ Z (DIV (ADD (TIMES 2 Z) Y) 2)) Fi)
       (SETQ Y (DiV2 Y))
       REPEAT)
 (ASSERT (AND (LTQ (TIMES 2 Z) (DIV A (DIV B 2)))
              (LT (DIV A (TIMES 2 (DIV B 2))) (ADD (DIV (TIMES 2 Z) 2) E)))))
```

Non-executable statements (involving DIV) are now replaced by executable ones (DIV2) as part of a simplification step. The simplified expressions have been underscored; they include replacing TIMES by TIMES2, where possible. Thus the system obtains its first corrected program:

```
((ASSERT (AND (LTQ 0 A) (LT A (TIMES 2 (DIV B 2))) (LT 0 E)))
 (SETQ Z 0) (SETQ Y 1)
 (LOOP (ASSERT (AND (LTQ (TIMES (DIV B 2) (TIMES 2 Z)) A)
                    (LT A (TIMES (DIV B 2) (ADD (TIMES 2 Z) (TIMES 2 Y))))))
       (UNTIL (LTQ Y E))
       (IF (LTQ (TIMES (DIV2 B) (ADD (TIMES2 Z) Y)) A)
           THEN (SETQ Z (DIV2 (ADD (TIMES2 Z) Y))) FI)
       (SETQ Y (DiV2 Y))
       R E P E A T )
 (ASSERT (AND (LTQ (TIMES 2 Z) (DIV A (DIV B 2)))
              (LT (DIVA (TIMES 2 (DIV B 2))) (ADD (DIV (TIMES 2 Z) 2) E)))))
```

Lastly, it must be proved that the transformed input assertion is implied by the given input specification, i.e.:

```
(IMPLIES (AND (LTQ 0 A) (LT A B) (LT 0 E))
         (AND (LTQ 0 A) (LT A (TIMES 2 (DIV B 2))) (LT 0 E)))
```

and it does, since (TIMES 2 (DIV B 2)) is equal to B.

The second possible transformation, transformation 3.. is now applied:

```
TRANSFORM-PROGRAM:
  ((ASSERT (AND (LTQ 0 A) (LT A (TIMES 2 B)) (LT 0 E)))
   (SETQ Z 0) (SETQ Y 1)
   (LOOP (ASSERT (AND (LTQ (TIMES B Z) A) (LT A (TIMES B (ADD Z (TIMES 2 Y))))))
         (UNTIL (LTQ Y E))
         (IF (LTQ (TIMES B (ADD Z Y)) A) THEN (SETQ Z (ADD Z Y)) FI)
         (SETQ Y (DIV2 Y))
         REPEAT)
   (ASSERT (AND (LTQ Z (DIV A B)) (LT (DIV A (TIMES 2 B)) (ADD (DIV Z 2) E)))))
  ((TRANSFORM E (DIV E 2)))
```

obtaining (after simplification) a second corrected program:

```
  ((ASSERT (AND (LTQ 0 A) (LT A (TIMES 2 B)) (LT 0 (DIV E 2))))
   (SETQ Z 0) (SETQ Y 1)
   (LOOP (ASSERT (AND (LTQ (TIMES B Z) A) (LT A (TIMES B (ADD Z (TIMES 2 Y))))))
         (UNTIL (LTQ Y (DIV2 E)))
         (IF (LTQ (TIMES B (ADD Z Y)) A) THEN (SETQ Z (ADD Z Y)) FI)
         (SETQ Y (DIV2 Y))
         REPEAT)
   (ASSERT (AND (LTQ Z (DIV A B)) (LT (DIV A (TIMES 2 B)) (ADD (DIV Z 2) (DIV E 2))))))
```

Again it must be shown that the transformed input assertion is implied by the input specification:

```
(IMPLIES (AND (LTQ 0 A) (LT A B) (LT 0 E))
         (AND (LTQ 0 A) (LT A (TIMES 2 B)) (LT 0 (DIV E 2))))
```

which is indeed true, since A<2B is implied by A<B and 0<E/2 is equivalent to 0<E.