Inference Rules for Program Annotation

NACHUM DERSHOWITZ AND ZOHAR MANNA

Abstract—Methods are presented whereby an Algol-like program given together with its specifications can be documented automatically. The program is incrementally annotated with invariant relations that hold between program variables at intermediate points in the program text and explain the actual workings of the program regardless of whether it is correct. Thus, this documentation can be used for proving correctness of programs or may serve as an aid in debugging incorrect programs.

The annotation techniques are formulated as Hoare-like inference rules that derive invariants from the assignment statements, from the control structure of the program, or, heuristically, from suggested invariants. The application of these rules is demonstrated by examples that have run on an experimental implementation.

Index Terms—Inference rules, invariant assertions, program annotation, program correctness, verification.

I. Introduction

A CONVENIENT form for expressing many facts about a program is a set of *invariant assertions* (*invariants*, for short) that describe relations between the different variables manipulated by the program. Invariant assertions play an important role in many aspects of programming, including: proving correctness and termination, proving incorrectness, guiding debugging, analyzing efficiency, and aiding in optimization.

Program annotation is the process of discovering these invariants. We are given an Algol-like program along with an *output specification*, stating the desired relation among the program variables upon termination, and an *input specification*, defining the set of inputs on which the program is intended to operate. It is, however, not known whether or not the program is correct and satisfies the given output specification. Our task is to generate the invariant assertions describing the workings of the program as is, independent of its correctness or incorrectness. The process is iterative, since finding some invariants suggests others. Assertions supplied by the programmer cannot be assumed true, although they may be used to guide the search for correct invariants.

Manuscript received August 28, 1978; revised December 21, 1979. This work was supported in part by the U.S. Air Force Office of Scientific Research under Grant AFOSR-81-0014, in part by the National Science Foundation under Grants MCS-79-09495 and MCS-79-04897, and in part by the Office of Naval Research under Contract N00014-76-C-0687. A previous version of this paper appeared in the *Proceedings of the Third International Conference on Software Engineering*, Atlanta, GA, May 1978.

N. Dershowitz was with the Department of Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel. He is now with the Department of Computer Science, University of Illinois, Urbana-Champaign, IL 61801.

Z. Manna is with the Department of Computer Science, Stanford University, Stanford, CA 94305 and with the Department of Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel.

In the following sections, we present a unified approach to program annotation, using *annotation rules*—in the style of Hoare [13]—to derive invariants. Section II presents an overview of our approach. It is followed by a detailed example in Section III. This example is among those that have run successfully on our experimental system. A detailed example involving arrays and nested loops, as well as a description of our QLISP implementation, may be found in Dershowitz [5]. In Section IV we briefly discuss the importance of annotation. A catalog of approximately 40 annotation rules is included in the Appendix.

Some earlier annotation systems are

- the system described in Elspas [6], based mainly upon the solution of difference equations;
- VISTA (German [7], German and Wegbreit [9]), based upon the top-down heuristics of Wegbreit [31]; and
- ADI (Tamir [28], [29]), an interactive system based upon the methods of Katz and Manna [15], [16] and Katz [14].

Our system as described here, attempts to incorporate and expand upon those systems. Suzuki and Ishihata [27] and German [8] have implemented systems that generate invariants useful in checking for various runtime errors.

II. Overview

In this section we first define some terminology and then present samples of each type of annotation rule.

A. Notation and Terminology

Given a program with its specifications, our goal is to document the program automatically with invariants. If the program is correct with respect to the specifications, we would like the invariants to provide sufficient information to demonstrate its correctness; if the program is incorrect, we would like information helpful in determining what is wrong with it.

We shall be dealing with three types of assertions.

- *Global invariants* are relations that hold at all places (i.e., labels) and at all times during the execution of some program segment. We write

  assert α **in** P

to indicate that the relation α is a global invariant in program segment P. (Actually, α is considered a global invariant even if it only begins to hold once the variables in α have been assigned an initial value within P.)

- *Local invariants* are associated with specific points in the program and hold for the current values of the variables whenever control passes through the corresponding point. Thus,

0098-5589/81/0300-0207$00.75

L: **assert** α

means that the relation α holds each time control is at label L.

- *Candidate assertions*, also associated with specified points, are relations hypothesized to be local invariants, but that have not yet been verified. We write

L: **suggest** α.

Consider the following simple program, meant to compute the quotient q and remainder r of the integer input values c and d:

```
P0: begin comment integer-division program
    B0: assert c ∈ N, d ∈ N + 1
    (q, r) := (0, c)
    loop L0: assert ···
        until r < d
        (q, r) := (q + 1, r - d)
        repeat
    E0: suggest q ≤ c/d, c/d < q + 1, q ∈ Z, r = c - q · d
    end,
```

where $\mathbf{N}$ is the set of nonnegative integers, $\mathbf{N}+1$ is the set of positive integers, and $\mathbf{Z}$ is the set of all integers. The *loop-until-repeat* construct[1] indicates that the two loop-body assignments $q := q + 1$ and $r := r - d$ are repeated zero or more times until the exit test $r < d$ is true for the first time. This program will be used only to illustrate various aspects of program annotation; a complete example of annotation is given in the next section.

The invariant

assert $c \in \mathbf{N},\ d \in \mathbf{N}+1$

attached to the **begin**-label B_0 is the input specification of the program defining the class of "legal" inputs. It indicates that whenever computation starts at B_0, the variable c is a natural number and d a positive integer.

The candidate

suggest $q \leqslant c/d,\ c/d < q+1,\ q \in \mathbf{Z},\ r = c - q \cdot d$

attached to the **end**-label E_0 is the output specification of the program. It states that the desired outcome of the program is that q be the largest integer not larger than c/d and r be the remainder. Since one cannot assume that the programmer has not erred, initially the programmer-supplied assertions—including the program's output specification but excluding the input specification—are only candidates for invariants.

In order to verify that a candidate is indeed a local invariant, we must show that whenever control reaches the corresponding point, the candidate holds. Suppose that we are given a candidate for a loop invariant

L_0: **suggest** $r = c - q \cdot d$.

To prove that it is an invariant, one must show 1) that the relation holds at L_0 when the loop is first entered and 2) that once it holds at L_0, it remains true each subsequent time control returns to L_0. If we succeed, then we would write

L_0: **assert** $r = c - q \cdot d$.

Furthermore, if $r = c - q \cdot d$ holds whenever control is at L_0, then it will also hold whenever control leaves the loop and reaches E_0. In other words, $r = c - q \cdot d$ would also be an invariant at E_0 and may be removed from the list of candidates at E_0. In that case we would write

E_0: **assert** $r = c - q \cdot d$; **suggest** $q \leqslant c/d,\ c/d < q+1,\ q \in \mathbf{Z}$.

Global invariants often express the range of variables. For example, since the variable q is first initialized to 0 and then repeatedly incremented by 1, it is obvious that the value of q is always a nonnegative integer. Thus we have the global invariant

assert $q \in \mathbf{N}$ **in** P_0

that relates to the program as a whole and states that $q \in \mathbf{N}$ throughout execution of the program P_0.

In this paper we describe various annotation techniques. These techniques are expressed as rules: the antecedents of each rule are usually annotated program segments containing invariants or candidate invariants and the consequent is either an invariant or a candidate. The rules are numbered ⟨1⟩, ⟨2⟩, etc. and are listed in the Appendix. This list is representative of the kinds of rules that may be used for annotation; it is not, however, meant to be a complete list.

We differentiate between three types of rules: assignment rules, control rules, and heuristic rules.

- *Assignment rules* yield *global* invariants based only upon the assignment statements of the program.
- *Control rules* yield *local* invariants based upon the control structure of the program.
- *Heuristic rules* have *candidates* as their consequents. These candidates, although promising, are not guaranteed to be invariants.

The assignment and control rules are *algorithmic* in the sense that they derive relations in such a manner as to *guarantee* that they are invariants. The heuristics are rules of *plausible* inference, reflecting common programming practice.

B. Assignment Rules

Many of the algorithmic rules depend only upon the assignment statements of the program and not upon its control structure. In other words, whether the assignments appear within an iterative or recursive loop or on some branch of a conditional statement is irrelevant. Since the location and order in which assignments are executed does not affect the validity of the rules, these rules yield global invariants.

The various assignment rules relate to particular operators occurring in the assignment statements of the program. Some of the rules for addition, for example, are as follows: an *addition rule* that gives the range of a variable that is updated by adding (or subtracting) a constant; a *set-addition rule* for the case where a variable is updated by adding another variable whose range is already known; and an *addition-relation rule* that relates two variables always incremented by similar expressions. Corresponding rules apply to other operators.

In dealing with sets, we find the following notation convenient. Let $f(s_1, s_2, \cdots, s_m)$ be any expression containing

[1] Based on the suggestion of J. Ole-Dahl in Knuth [18].

occurrences of m distinct subexpressions $s_1, s_2, \cdots, s_m$. The set of elements

$$\{f(s_1, s_2, \cdots, s_m):\ s_1 \in S_1, s_2 \in S_2, \cdots, s_m \in S_m\}$$

is denoted by

$$f(S_1, S_2, \cdots, S_m).$$

Using this notation, we have the *addition rule* ⟨1⟩

$$\frac{x := a_0 \quad x := x + a_1 \quad x := x + a_2 \cdots \textbf{in}\ P}{\textbf{assert}\ x \in a_0 + a_1 \cdot \mathbb{N} + a_2 \cdot \mathbb{N} + \cdots \textbf{in}\ P,}$$

where P is a program segment and the expressions a_i are of constant value within P. The antecedent

$$x := a_0 \quad x := x + a_1 \quad x := x + a_2 \cdots \textbf{in}\ P$$

indicates that the *only* assignments to the variable x in P are $x := a_0, x := x + a_1, x := x + a_2$, etc. The consequent

$$\textbf{assert}\ x \in a_0 + a_1 \cdot \mathbb{N} + a_2 \cdot \mathbb{N} + \cdots \textbf{in}\ P$$

is a global invariant indicating that x belongs to the set $a_0 + a_1 \cdot \mathbb{N} + a_2 \cdot \mathbb{N} + \cdots$, i.e., $x = a_0 + a_1 \cdot n_1 + a_2 \cdot n_2 + \cdots$ for some $n_1, n_2, \cdots \in \mathbb{N}$. This relation holds throughout execution of P–but *only* from the point when x first receives a defined value in P via the assignment $x := a_0$. (After any execution of $x := a_0$, clearly $x \in a_0 + a_1 \cdot \mathbb{N} + a_2 \cdot \mathbb{N} + \cdots$, with $x = a_0 + a_1 \cdot 0 + a_2 \cdot 0 + \cdots$, and if $x = a_0 + a_1 \cdot n_1 + a_2 \cdot n_2 + \cdots$ for some $n_1, n_2, \cdots$ before executing $x := x + a_1$, then $x = a_0 + a_1 \cdot (n_1 + 1) + a_2 \cdot n_2 + \cdots$ after executing the assignment. Thus, n_1 represents the number of executions of $x := x + a_1$ since $x := a_0$ was executed last, n_2 is the number of executions of $x := x + a_2$, etc.) From such an invariant, more specific properties may be derived. For example, a bound on x may be derived using methods of *interval arithmetic* (see, for example, Gibb [10]). Note that no restrictions are placed on the order in which the assignments to x are executed, except that prior to the first execution of $x := a_0$ the invariant may not hold.

In our simple program P_0, the assignments to the variable q are

$$q := 0 \quad q := q + 1.$$

So we can apply the *addition rule*, instantiating a_0 with 0 and a_1 with 1, and obtain the global invariant $q \in 0 + 1 \cdot \mathbb{N}$, i.e.,

$$\textbf{assert}\ q \in \mathbb{N}\ \textbf{in}\ P_0.$$

The assignments to r in P_0 are

$$r := c \quad r := r - d.$$

Applying the same rule to them, letting $a_0 = c$ and $a_1 = -d$, yields the invariant

$$\textbf{assert}\ r \in c - d \cdot \mathbb{N}\ \textbf{in}\ P_0.$$

Given that d is positive, we may conclude that $r \leqslant c$.

The *set-addition rule* is a more general form of the above *addition rule*, applicable to nondeterministic assignments of the form $x :\in f(S)$, where an arbitrary element in the set $f(S) = \{f(s):\ s \in S\}$ is assigned to x. Note that an assignment $x := f(s)$, where it is already known about s that $s \in S$, may be viewed as the nondeterministic assignment $x :\in f(S)$. The *set-addition rule* ⟨5⟩ is

$$\frac{x :\in S_0 \quad x :\in x + S_1 \quad x :\in x + S_2 \cdots \textbf{in}\ P}{\textbf{assert}\ x \in S_0 + \Sigma S_1 + \Sigma S_2 + \cdots \textbf{in}\ P,}$$

where ΣS denotes the set of finite sums $s_1 + s_2 + \cdots + s_m$ for (not necessarily distinct) addends s_i in S. (If $S = \phi$, the sum is 0; if S contains the single element s, then $\Sigma S = s \cdot \mathbb{N}$. This rule applies analogously to any associative and commutative operator "$\oplus$".) These assignment rules for global invariants are related to the weak interpretation method of Sintzoff [26] (see also Wegbreit [32], Wegbreit and Spitzen [33], and Harrison [12]) that has been implemented by Scherlis [25] and German and Wegbreit [9].

In program P_0 the assignments to r were

$$r := c \quad r := r - d.$$

Since we are given that $c \in \mathbb{N}$ and $d \in \mathbb{N} + 1$, we may view these as the nondeterministic assignments

$$r :\in \mathbb{N} \quad r :\in r - (\mathbb{N} + 1),$$

and by applying the *set-addition rule* we obtain the global invariant $r \in \mathbb{N} - \Sigma(\mathbb{N} + 1)$. This simplifies to

$$\textbf{assert}\ r \in \mathbb{Z}\ \textbf{in}\ P_0,$$

where $\mathbb{Z}$ is the set of all integers.

To relate different variables appearing in a program, we have an *addition-relation rule* ⟨11⟩:

$$\frac{\begin{array}{l}(x, y) := (a_0, b_0) \\ \qquad (x, y) := (x + a_1 \cdot u, y + b_1 \cdot u) \\ \qquad\qquad (x, y) := (x + a_1 \cdot v, y + b_1 \cdot v) \cdots \textbf{in}\ P\end{array}}{\textbf{assert}\ a_1 \cdot (y - b_0) = b_1 \cdot (x - a_0)\ \textbf{in}\ P,}$$

where $u, v, \cdots$ are arbitrary (not necessarily constant) expressions. The invariant begins to hold only when the multiple assignment $(x, y) := (a_0, b_0)$ has been executed for the first time. (The invariant $a_1 \cdot (y - b_0) = b_1 \cdot (x - a_0)$ clearly holds when $x = a_0$ and $y = b_0$. Assuming it holds before executing $(x, y) := (x + a_1 \cdot u, y + b_1 \cdot u)$, then after executing the assignment both sides of the equality are increased by $a_1 \cdot b_1 \cdot u$ and the invariant still holds.) The multiple assignments in the antecedent of such rules, e.g., $(x, y) := (x + a_1 \cdot u, y + b_1 \cdot u)$, may represent the cumulative effect of individual assignments lying on a path between two labels, with the understanding that whenever $x := x + a_1 \cdot u$ is executed, so is $y := y + b_1 \cdot u$ for the same value of the expression u. In that case, the invariant will not, in general, hold between the individual assignments.

In our example the assignments in the initialization path give us

$$(q, r) := (0, c)$$

and for the loop-body path we have

$$(q, r) := (q + 1, r - d).$$

By a simple application of the *addition-relation rule* with $a_0 = 0$, $b_0 = c$, $a_1 = u = v = 1$, and $b_1 = -d$, we derive the invariant $1 \cdot (r - c) = -d \cdot (q - 0)$ which simplifies to

assert $r = c - q \cdot d$ **in** P_0.

Note that this *addition-relation rule* (as well as several other relation rules) may be derived from the following general relation-rule schema:

$$\frac{\begin{array}{l}(x,y) := (a_0, b_0)\\ \quad (x,y) := (x \oplus (u \otimes a_1), y \oplus (u \otimes b_1))\\ \quad\quad (x,y) := (x \oplus (v \otimes a_1), y \oplus (v \otimes b_1)) \cdots \textbf{in } P\end{array}}{\textbf{assert } (a_0 \otimes b_1) \oplus (y \otimes a_1) = (b_0 \otimes a_1) \oplus (x \otimes b_1) \textbf{ in } P,}$$

where the operator $\oplus$ is commutative and associative, operator $\otimes$ satisfies $(a \otimes b) \otimes c = (a \otimes c) \otimes b$, and $(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$. The various relation rules are related to optimization techniques, to the metatheorems in Manna and Pnueli [21], and to the approach of Caplain [3].

C. Control Rules

Unlike the previous rules that completely ignore the control structure of the program, *control rules* derive important invariants from the program structure; they are akin to the verification rules of Hoare [13]. There are several rules for each program construct; two rules, for example, push invariants forward in a loop.

The *forward loop-exit rule* ⟨31⟩

$$\frac{\begin{array}{l}\textbf{loop } P'\\ \quad \textbf{assert } \alpha\\ \quad \textbf{until } t\\ \quad L':\\ \quad P''\\ \quad \textbf{repeat}\\ L'':\end{array}}{\begin{array}{ll}L': & \textbf{assert } \alpha, \neg t\\ L'': & \textbf{assert } \alpha, t\end{array}}$$

reflects the fact that if execution of a loop terminates at L'', then the exit test t must have just held, while if the loop is continued at L', the exit test was false. Furthermore, any relation α that held just prior to the test, also holds immediately after. Applying this rule to the loop in the integer-division program P_0 yields the invariant $r < d$ at E_0 and $r \geq d$ at the head of the loop body:

$(q, r) := (0, c)$
loop L_0:
  until $r < d$
  assert $r \geq d$
  $(q, r) := (q + 1, r - d)$
  repeat
E_0: **assert** $r < d$.

To propagate invariants such as $r \geq d$ past assignment statements, we have a *forward assignment rule* ⟨21⟩

$$\frac{\begin{array}{l}\textbf{assert } \alpha(x,y)\\ x := f(x,y)\\ L:\end{array}}{L: \textbf{ assert } \alpha(f^{-}(x,y), y),}$$

where f^{-} is the inverse (assuming that there is an inverse) of the function f in the first argument, i.e., $f^{-}(f(x,y), y) = x$. By using the inverse function f^{-}, the value of x prior to the assignment may be expressed in terms of the current value of x as $f^{-}(x,y)$. Thus, if the relation $\alpha(x,y)$ held before the assignment to x, then after the assignment $\alpha(f^{-}(x,y), y)$ holds, where $f^{-}(x,y)$ has been substituted for all occurrences of x in $\alpha(x,y)$. (Even if there is no inverse function, variants of this rule, e.g., ⟨21b⟩, may often be used to glean some useful information.) In our example, since the loop-body assignment $q := q + 1$ does not affect any variable appearing in the invariant $r \geq d$, the invariant is pushed forward unchanged. To propagate $r \geq d$ past the assignment $r := r - d$, we replace r by the inverse of $r - d$, that is, $r + d$, yielding $r + d \geq d$, or

assert $r \geq 0$,

at the end of the loop body.

We also have an *assignment axiom* ⟨18⟩

$x := a$
assert $x = a$,

where the expression a may not contain x. This axiom gives us, for example, the invariant

assert $r = c$

prior to entering the loop.

The *forward loop-body rule* ⟨29⟩

$$\frac{\begin{array}{l}\textbf{assert } \alpha\\ \textbf{loop } L:\\ \quad P\\ \quad \textbf{assert } \beta\\ \quad \textbf{repeat}\end{array}}{L: \textbf{ assert } \alpha \vee \beta}$$

states that for control to be at the head of a loop, at L, either the loop has just been entered or the loop body has been executed and the loop is being repeated. Therefore, the disjunction $\alpha \vee \beta$ of an invariant α, known to hold just before the loop, and an invariant β, known to hold at the end of the loop body, must hold at L. By this second rule for loops, we get the loop invariant

L_0: **assert** $r = c \vee r \geq 0$.

By the input specification $c \in \mathbb{N}$, the first disjunct $r = c$ implies the second $r \geq 0$; this invariant therefore simplifies to

L_0: **assert** $r \geq 0$.

With the global invariant $r = c - q \cdot d$, we get $c - q \cdot d \geq 0$, i.e., $q \leq c/d$.

To generate invariants from a conditional test, we have a *forward test rule* ⟨25⟩:

$$\frac{\begin{array}{l}\textbf{assert } \alpha\\ \textbf{if } t \textbf{ then } L':\\ \quad\quad P'\\ \quad \textbf{else } L'':\\ \quad\quad P''\\ \quad \textbf{fi}\end{array}}{\begin{array}{ll}L': & \textbf{assert } \alpha, t\\ L'': & \textbf{assert } \alpha, \neg t.\end{array}}$$

That is, for the **then**-branch to be taken t must be true, while for the **else**-branch to be taken it must be false; furthermore, any α that held before the test, also holds after. Once invariants have been generated for the two branches, they are pushed forward by the *forward branch rule* ⟨27⟩:

if t **then** P'
assert α
else P''
assert β
fi
L:

L: **assert** $\alpha \vee \beta$.

It states that for control to be at the point after the conditional statement, one of the two branches must have been traversed.

The following *forall rule* ⟨35⟩ is valuable for programs with a universally quantified output specification. Given a loop invariant $\alpha(x)$ at L containing the integer variable (or expression) x and *no* other variables, check if x is monotonically increasing by one. If it is, then we have as a loop invariant at L that α still holds for all intermediate values lying between the initial and current values. That is,

assert $x = a$, $x \in \mathbf{Z}$
loop L: **assert** $\alpha(x)$
P
assert $x = x_L + 1$
repeat

L: **assert** $(\forall \zeta \in \mathbf{Z})(a \leqslant \zeta \leqslant x)\ \alpha(\zeta)$,

where a is an integer expression with a constant value in P and x_L is the value of x when last at L. (This rule is similar to the universal-quantification technique for arrays in Katz and Manna [15].) The *forall rule* may be broadened to apply when x is increasing by an amount other than 1, or for decreasing x.

D. Schematic Rules

In this subsection, we shall illustrate how the control rules may be applied to derive invariants for program schemata. Once invariants have been generated for a particular schema, they can be used for any instance of that schema.

Consider, for example, the following single-loop, single-conditional, program schema:

P^*: **begin comment** *single-loop schema*
$z := c$
loop L^*: **assert** $\cdots$
until $t(z)$
$z := f(z)$
if $s(z)$ **then** $z := g(z)$ **else** $z := h(z)$ **fi**
repeat
end.

We shall assume that the inverse functions f^-, g^-, and h^- are available whenever required by the rules.

The *assignment axiom* ⟨18⟩ applied to the initial assignment $z := c$ yields the invariant

assert $z = c$

before the loop. The *forward loop-exit rule* ⟨31⟩ generates the invariant $\neg t(z)$ at the head of the loop body, immediately after the **until**-clause, and then the *forward assignment rule* ⟨21⟩ gives $\neg t(f^-(z))$ preceding the conditional:

assert $\neg t(f^-(z))$
if $s(z)$ **then** $z := g(z)$ **else** $z := h(z)$ **fi**.

The *forward test rule* ⟨25⟩ propagates that invariant forward, adding $s(z)$ at the head of the **then**-clause and $\neg s(z)$ at the head of the **else**-clause:

if $s(z)$ **then assert** $\neg t(f^-(z))$, $s(z)$
$z := g(z)$
else assert $\neg t(f^-(z))$, $\neg s(z)$
$z := h(z)$
fi

By pushing $\neg t(f^-(z))$ and $s(z)$ through the **then**-branch assignment $z := g(z)$, and $\neg t(f^-(z))$ and $\neg s(z)$ through the **else**-branch assignment $z := h(z)$, we get

if $s(z)$ **then** $z := g(z)$
assert $\neg t(f^-(g^-(z)))$, $s(g^-(z))$
else $z := h(z)$
assert $\neg t(f^-(h^-(z)))$, $\neg s(h^-(z))$
fi.

Combining the invariants from the two different paths—using the *forward branch rule* ⟨27⟩—one gets

assert $[\neg t(f^-(g^-(z))) \wedge s(g^-(z))] \vee$
$[\neg t(f^-(h^-(z))) \wedge \neg s(h^-(z))]$

after the conditional, at the end of the loop body.

The *forward loop-body rule* ⟨29⟩ expressed the fact that if control is at the head of a loop, either the loop-initialization invariant or the loop-body invariant must hold. Applying this rule to our schema

assert $z = c$
loop L^*: **assert** $\cdots$
until $t(z)$
$z := f(z)$
if $s(z)$ **then** $z := g(z)$ **else** $z := h(z)$ **fi**
assert $[\neg t(f^-(g^-(z))) \wedge s(g^-(z))] \vee$
$[\neg t(f^-(h^-(z))) \wedge \neg s(h^-(z))]$
repeat

we derive the loop invariant

L^*: **assert** $z = c \vee [\neg t(f^-(g^-(z))) \wedge s(g^-(z))]$
$\vee [\neg t(f^-(h^-(z))) \wedge \neg s(h^-(z))]$.

This loop invariant embodies two facts about the control structure of this schema.

- Whenever control is at L^*, either the loop has just been entered or the loop exit-test was false the last time around the loop. That is,

L^*: **assert** $z = c \vee \neg t(f^-(g^-(z))) \vee \neg t(f^-(h^-(z)))$.

The first disjunct is the result of the initialization path; the second states that the exit test was false for the value of z when L^* was last visited, assuming control came via the **then**-path of the conditional; the third disjunct says the same for the case when control came via the **else**-path.

• Whenever control is at L^*, either the loop has just been entered, or the conditional test was true the last time around and the **then**-path was taken, or the test was false and the **else**-path was taken. That is,

L^*: **assert** $z = c \vee s(g^-(z)) \vee \neg s(h^-(z))$.

As another simple example, consider the loop schema

$z := 0$
loop L:
  until $t(z)$
  $z := z + 1$
  repeat
E: .

By the *label axiom* ⟨33⟩

L: **assert** $x = x_L$,

we get

L: **assert** $z = z_L$.

Thus we can easily derive the following invariants:

$z := 0$
assert $z = 0$
loop L: **assert** $z = z_L$
  until $t(z)$
  $z := z + 1$
  assert $z - 1 = z_L$, $\neg t(z - 1)$
  repeat
E: **assert** $t(z)$.

Now, by the *forward loop-body rule* ⟨29⟩ we can derive the invariant

L: **assert** $z = 0 \vee \neg t(z - 1)$

and by the *forall rule* ⟨35⟩, we get

L: **assert** $(\forall \zeta \in \mathbf{Z})(0 \leqslant \zeta \leqslant z)(\zeta = 0 \vee \neg t(\zeta - 1))$.

This simplifies to

L: **assert** $(\forall \zeta \in \mathbf{N})(\zeta < z) \neg t(\zeta)$.

Combined with the invariant $t(z)$ that holds at E, it implies that the final value of z is the minimum nonnegative integer satisfying the predicate t.

E. Heuristic Rules

In contrast with the above rules that derive relations guaranteed to be invariants, there is another class of rules, *heuristic rules*, that can only suggest candidates for invariants. These candidates must be verified.

As an example, consider the following *conditional heuristic* ⟨36⟩:

if t **then** P'
    assert α
  else P''
    assert β
  fi
L:

L: **suggest** α, β.

Since we know that α holds if the **then**-path P' is taken, while β holds if the **else**-path P'' is taken, clearly their disjunction $\alpha \vee \beta$ holds at L in either case (that was expressed in the *forward branch rule* ⟨27⟩). However, since in constructing a program, a conditional statement is often used to achieve the same relation in alternative cases it is plausible that α (or, by the same token, β) may hold true for *both* the **then**- and **else**-paths.

As mentioned earlier, the output specification and user-supplied assertions are the initial set of candidates. Candidates are propagated over assignment and conditional statements using the same control rules as for invariants. The *top-down heuristic* ⟨38⟩,

assert α
loop L:
  until t
  P
  repeat
suggest γ

$\alpha \supset \gamma$

L: **suggest** γ,

may be used to push a candidate (or invariant) γ backwards into a loop. Although $t \supset \gamma$ (i.e., $\neg t \vee \gamma$) would be a sufficiently strong loop invariant at L to establish γ upon loop exit, the heuristic suggests a stronger candidate, γ itself, at L. Since a necessary condition for γ to be an invariant is that it hold upon entrance to the loop, the second antecedent of the rule requires that the invariant α before the loop imply that γ holds. The idea underlying this heuristic is that an iterative loop is constructed in order to achieve a conjunctive goal $(t \wedge \gamma)$ by placing one conjunct of the goal in the exit test and maintaining the other invariantly true.

Wegbreit [31] and Katz and Manna [16] have suggested a more general form of these two heuristics:

L: **assert** $\alpha \vee \beta$

L: **suggest** α, β.

However, as they remark, this heuristic should not be applied indiscriminately to any disjunctive invariant. We would not, for example, want to replace all occurrences of an invariant $x \geqslant 0$ with the candidates $x > 0$ and $x = 0$. Special cases, such as the above *conditional* and *top-down* heuristics are needed to indicate where the strategy is relatively likely to be profitable.

Returning to our integer-division example P_0, the *top-down heuristic* suggests that of the candidates

E_0: **suggest** $q \leqslant c/d,\ c/d < q + 1,\ q \in \mathbf{Z},\ r = c - q \cdot d,$

those that hold upon entering the loop—when $q = 0$ and $r = c$—are also candidates at L_0. They are

L_0: **suggest** $q \leqslant c/d$, $q \in \mathbf{Z}$, $r = c - q \cdot d$.

The remaining candidate at E_0, $c/d < q + 1$, does not necessarily hold for $q = 0$.

Each candidate must be checked for invariance: it must hold for the loop-initialization path and must be maintained true around the loop. Of the three candidates at L_0, the last two, $q \in \mathbf{Z}$ and $r = c - q \cdot d$, have already been shown to be global invariants. To prove that the first, $q \leqslant c/d$, is a loop invariant at L_0, we try to show that if $q \leqslant c/d$ is true at L_0 and the loop is continued, then $q \leqslant c/d$ holds when control returns to L_0, i.e.,

$$q \leqslant c/d \wedge r \geqslant d \supset q + 1 \leqslant c/d.$$

This condition, however, is not provable. Nevertheless, we can show that $q \leqslant c/d$ is an invariant by making use of the global invariant $r = c - q \cdot d$. Substituting $c - q \cdot d$ for r in $r \geqslant d$ yields $c - q \cdot d \geqslant d$; it follows that the above implication holds and $q \leqslant c/d$ is an invariant at L_0. Thus, while an attempt to directly verify the candidate $q \leqslant c/d$ failed, once we have established that $r = c - q \cdot d$ is an invariant, we can also show that $q \leqslant c/d$ is an invariant.

Indeed, in general there may be insufficient information to prove that a candidate is invariant when it is first suggested, and only when other invariants are subsequently discovered might it become possible to verify the candidate. Therefore, candidates should be retained until all invariants and candidates have been generated. Unproved candidates are also used by the heuristics to general additional candidates. For example, the *top-down heuristic* uses the as yet unproved candidate γ to generate the loop candidate γ at L.

Another heuristic, valuable for loops with universally quantified output invariants, is the *generalization heuristic* ⟨37⟩

assert $x = a$
loop L: **suggest** $\alpha(x, y)$
  P
  assert $x = f(x_L)$
  repeat

L: **suggest** $(\forall \zeta \in \{a, f(a), f(f(a)), \cdots, x\})\ \alpha(\zeta, y)$.

Given a loop candidate $\alpha(x, y)$, we determine the set of values that the variable x takes on. Then we have as a new candidate for a loop invariant that α still holds for all the intermediate values between the initial value a and the current value x. For example, if $a \in \mathbf{Z}$ and $f(x) = x + 1$, then we get the candidate

L: **suggest** $(\forall \zeta \in \mathbf{Z})(a \leqslant \zeta \leqslant x)\ \alpha(\zeta, y)$.

This is a candidate and not an invariant since the program segment P may vary the value of y in such a way as to destroy the relation $\alpha(x, y)$ for previous values of x.

Note that a candidate invariant must sometimes be replaced by a stronger candidate in order to prove invariance. This is analogous to other forms of proof by induction, where it is often necessary to strengthen the desired theorem to carry out a proof. The reason is that by strengthening the theorem to be proved, we are at the same time strengthening the hypothesis that is used in the inductive step. We could not, for example, directly prove that the relation $(r \geqslant d) \vee (r = c - q \cdot d)$ is a loop invariant (that is the necessary condition for $r = c - q \cdot d$ to hold after the loop), since this candidate is not preserved by the loop, i.e.,

$$[r \geqslant d \vee r = c - q \cdot d] \wedge r \geqslant d \supset [r - d \geqslant d \vee r - d = c - (q + 1) \cdot d]$$

is not provable. On the other hand, we can prove that the stronger relation $r = c - q \cdot d$ is an invariant, since we have a stronger hypothesis on the left-hand side of the implication; that is,

$$r = c - q \cdot d \wedge r \geqslant d \supset r - d = c - (q + 1) \cdot d$$

can be proved. Clearly, once we establish that $r = c - q \cdot d$ is an invariant, it follows that $(r \geqslant d) \vee (r = c - q \cdot d)$ also is.

Various specific methods of strengthening candidates have been discussed in the literature (Wegbreit [31], Katz and Manna [16], Moriconi [22], and others); they are closely associated with methods of "top-down" structured programming. Related techniques are used by Greif and Waldinger [11], Suzuki and Ishihata [27], and Basu [1]. Also the candidates that Basu and Misra [2] and Morris and Wegbreit [23] derive, using the *subgoal-induction* method of verification, fall into this class.

F. Counters

A useful technique for proving certain properties of programs is the augmentation of a program with counters of various sorts. For example, by initializing a counter to zero upon entering a loop and incrementing it by one with each iteration, the value of the counter will indicate the number of times that the loop has been executed. Then, relations between the program variables and the counter can be found. By deriving upper/lower bounds on the counter, the termination of the loop may be proved and time complexity analyzed.

As a simple example, reconsider our (now annotated) division program

assert $c \in \mathbf{N}$, $d \in \mathbf{N} + 1$, $q \in \mathbf{N}$, $r = c - q \cdot d$ **in**
P_0: **begin comment** *integer-division program*
  $(q, r) := (0, c)$
  loop L_0: **assert** $q \leqslant c/d$
    until $r < d$
    $(q, r) := (q + 1, r - d)$
    repeat
  E_0: **assert** $r < d$, $q \leqslant c/d$
  end.

The variable q is incremented by 1 with each loop iteration and is initialized to 0; thus it serves as a loop counter. Since the loop invariant $q \leqslant c/d$ gives an upper bound on the value of the counter and the counter is incremented with each loop iteration, the loop must terminate. Since the output invariant $r < d$ and global invariant $r = c - q \cdot d$ yield a lower bound on the value of the counter, one can determine the total number of loop iterations.

Examples of the use of counters for proving termination have appeared in Knuth [17], Katz and Manna [16], and Luckham and Suzuki [19]. Loop counters may also be used to discover relations between variables by solving first-order difference equations. (See, for example, Elspas [6] and Katz and Manna [16]; Netzer [24] applies this technique to recursive programs.) Related work, making use of a small collection of "loop-plans" to decompose program loops, may be found in Waters [30]. McCarthy and Talcott [20] distinguish between *extensional* properties of programs that depend only on the function computed by the program and *intentional* properties, such as space and time requirements, that may be made explicit in derived programs containing counters.

In the following section we demonstrate how a real-division program can be annotated using the annotation rules.

III. Example

Consider the following program P_1 purporting to approximate the quotient c/d of two nonnegative real numbers c and d, where $c < d$. Upon termination, the variable q should be no greater than the exact quotient and the difference between q and the quotient must be less than a given positive tolerance e. The program, with its specifications included as assertions, is

P_1: **begin comment** *real-division program*
  B_1: **assert** $0 \leqslant c < d,\ \ 0 < e$
  $(q, qq, r, rr) := (0, 0, 1, d)$
  loop L_1: **assert** $\cdots$
    until $r \leqslant e$
    if $qq + rr \leqslant c$ **then** $(q, qq) := (q + r, qq + rr)$ **fi**
    $(r, rr) := (r/2, rr/2)$
    repeat
  E_1: **suggest** $q \leqslant c/d,\ \ c/d < q + e$
  end.

Our goal is to find loop invariants at L_1 in order to verify the output candidates at E_1. In our presentation of the annotation of this program, we first apply the assignment rules and then the control rules combined with heuristic rules.

A. Assignment Rules

As a first step, we attempt to derive simple invariants by ignoring the control structure of the program and considering only the assignment statements. This will yield global invariants that hold throughout execution.

We first look for range invariants by considering all assignments to each variable. For example, since the assignments to r are

$$r := 1 \quad r := r/2,$$

we can apply the *multiplication rule* ⟨2⟩

$$\frac{x := a_0 \quad x := x \cdot a_1 \quad x := x \cdot a_2 \cdots \text{in } P}{\text{assert } x \in a_0 \cdot a_1^{\mathbb{N}} \cdot a_2^{\mathbb{N}} \cdots \text{in } P.}$$

Taking 1 for a_0 and $1/2$ for a_1, we derive the global invariant

$$\text{assert } r \in 1/2^{\mathbb{N}} \text{ in } P_1. \tag{1}$$

In other words, $r = 1/2^n$ for some nonnegative integer n. From this, it is possible to derive lower and upper bounds on r, i.e., $0 < r \leqslant 1$, since $r = 1$ when $n = 0$, while $r = 1/2^n$ approaches 0 as n grows larger.

Similarly, applying the *multiplication rule* to the assignments to rr,

$$rr := d \quad rr := rr/2,$$

yields

$$\text{assert } rr \in d/2^{\mathbb{N}} \text{ in } P_1. \tag{2}$$

Since we are given that $d > 0$, it follows that $0 < rr \leqslant d$.

The assignments to q are

$$q := 0 \quad q := q + r.$$

Since we know (1) $r \in 1/2^{\mathbb{N}}$, these assignments may be interpreted as the nondeterministic assignments

$$q :\in 0 \quad q :\in q + 1/2^{\mathbb{N}}.$$

Using the *set-addition rule* ⟨5⟩

$$\frac{x :\in S_0 \quad x :\in x + S_1 \quad x :\in x + S_2 \cdots \text{in } P}{\text{assert } x \in S_0 + \Sigma S_1 + \Sigma S_2 + \cdots \text{in } P,}$$

we conclude

$$\text{assert } q \in \Sigma 1/2^{\mathbb{N}} \text{ in } P_1.$$

This invariant states that q is a finite sum of elements of the form $1/2^n$, where n is some nonnegative integer. Since for any two such elements, one is a multiple of the other, it follows that the sum is of the form $m/2^n$, where $m, n \in \mathbb{N}$:

$$\text{assert } q \in \mathbb{N}/2^{\mathbb{N}} \text{ in } P_1 \tag{3}$$

(i.e., q is a dyadic rational number).

From (2) $rr \in d/2^{\mathbb{N}}$ and the assignments

$$qq := 0 \quad qq := qq + rr,$$

we get by the same *set-addition rule*

$$\text{assert } qq \in d \cdot \Sigma 1/2^{\mathbb{N}} \text{ in } P_1,$$

or

$$\text{assert } qq \in d \cdot \mathbb{N}/2^{\mathbb{N}} \text{ in } P_1. \tag{4}$$

The above four invariants give the range of each of the four program variables. Now we take up relations between pairs of variables by considering their respective assignments. Consider, first, the variables r and rr. Their assignments are

$$(r, rr) := (1, d) \quad (r, rr) := (r/2, rr/2).$$

Each time one is halved, so is the other; therefore, the proportion between the initial values of r and rr is maintained throughout loop execution. This is an instance of the *multiplication-relation rule* ⟨12⟩

$$\frac{(x, y) := (a_0, b_0) \quad (x, y) := (x \cdot u^{a_1}, y \cdot u^{b_1}) \quad (x, y) := (x \cdot v^{a_1}, y \cdot v^{b_1}) \cdots \text{in } P}{\text{assert } x^{b_1} \cdot b_0^{a_1} = a_0^{b_1} \cdot y^{a_1} \text{ in } P,}$$

yielding $r^1 \cdot d^1 = 1^1 \cdot rr^1$ which simplifies to

assert $rr = d \cdot r$ **in** P_1. (5)

The assignments to q and qq are

$$(q, qq) := (0, 0) \quad (q, qq) := (q + r, qq + rr).$$

Using (5) $rr = d \cdot r$ to substitute for rr in the assignment $qq := qq + rr$, we have

$$(q, qq) := (0, 0) \quad (q, qq) := (q + r, qq + d \cdot r),$$

which is an instance of the *addition-relation rule* ⟨11⟩

$$\frac{\begin{array}{l}(x, y) := (a_0, b_0)\\ \qquad (x, y) := (x + a_1 \cdot u, y + b_1 \cdot u)\\ \qquad\qquad (x, y) := (x + a_1 \cdot v, y + b_1 \cdot v) \cdots \textbf{in } P\end{array}}{\textbf{assert } a_1 \cdot (y - b_0) = b_1 \cdot (x - a_0) \textbf{ in } P.}$$

Thus we have the global invariant $1 \cdot (qq - 0) = d \cdot (q - 0)$, i.e.,

assert $qq = d \cdot q$ **in** P_1. (6)

In all, we have established the following global invariants:

assert $r \in 1/2^{\mathbb{N}}$, $rr \in d/2^{\mathbb{N}}$, $q \in \mathbb{N}/2^{\mathbb{N}}$, $qq \in d \cdot \mathbb{N}/2^{\mathbb{N}}$, $rr = d \cdot r$, $qq = d \cdot q$ **in** P_1.

B. Control Rules

So far we have derived global invariants from the assignment statements, ignoring the control structure of the program. We turn now to local invariants extracted from the program structure.

By applying the *assignment axiom* ⟨18⟩

$x := a$
assert $x = a$

to the multiple assignment at the beginning of the program we get the local invariants

assert $q = 0$, $qq = 0$, $r = 1$, $rr = d$

just prior to the loop. The *loop axiom* ⟨20⟩

loop P'
  until t
  assert $\neg t$
  P''
  repeat
assert t

yields $r > e$ at the head of the loop body and $r \leqslant e$ at E_1. Thus far, we have the annotated program segment

assert $q = 0$, $qq = 0$, $r = 1$, $rr = d$
loop L_1: **assert** $\cdots$
  until $r \leqslant e$
  assert $r > e$
  if $qq + rr \leqslant c$ **then** $(q, qq) := (q + r, qq + rr)$ **fi**
  $(r, rr) := (r/2, rr/2)$
  repeat
E_1: **assert** $r \leqslant e$.

Applying the *forward test rule* ⟨25⟩

$$\frac{\begin{array}{l}\textbf{assert } \alpha\\ \textbf{if } t \textbf{ then } L':\\ \qquad P'\\ \quad \textbf{else } L'':\\ \qquad P''\\ \quad \textbf{fi}\end{array}}{\begin{array}{l}L'\text{: } \textbf{assert } \alpha, t\\ L''\text{: } \textbf{assert } \alpha, \neg t\end{array}}$$

to the conditional statement of the loop

if $qq + rr \leqslant c$ **then** $(q, qq) := (q + r, qq + rr)$ **fi**

yields

if $qq + rr \leqslant c$ **then assert** $r > e, qq + rr \leqslant c$
  $(q, qq) := (q + r, qq + rr)$
else assert $r > e, c < qq + rr$
fi.

Using a variant of the *forward assignment rule* ⟨21b⟩

$$\frac{\begin{array}{l}\textbf{assert } \alpha(u, y)\\ x := u\\ L:\end{array}}{L\text{: } \textbf{assert } \alpha(x, y)}$$

where x does not appear in $\alpha(\xi, y)$, the assignment of the **then**-branch transforms the invariant $qq + rr \leqslant c$ into $qq \leqslant c$ and leaves $r > e$ unchanged. We obtain

if $qq + rr \leqslant c$ **then** $(q, qq) := (q + r, qq + rr)$
  assert $r > e, qq \leqslant c$
else assert $r > e, c < qq + rr$
fi.

We may now apply the *forward branch rule* ⟨27⟩

$$\frac{\begin{array}{l}\textbf{if } t \textbf{ then } P'\\ \qquad \textbf{assert } \alpha\\ \quad \textbf{else } P''\\ \qquad \textbf{assert } \beta\\ \quad \textbf{fi}\\ L:\end{array}}{L\text{: } \textbf{assert } \alpha \vee \beta}$$

to the two possible outcomes of the conditional. We obtain the invariant

assert $(r > e \wedge qq \leqslant c) \vee (r > e \wedge c < qq + rr)$,

which simplifies to just

assert $r > e$,

since $r > e$ appears in both disjuncts while $(qq \leqslant c) \vee (c < qq + rr)$ is implied by the global invariant (2) $rr > 0$ ($qq \leqslant c \vee c < qq$ is a tautology, and if rr is positive, then $c < qq$ implies $c < qq + rr$).

By application of the *forward assignment rule* ⟨21a⟩

$$\frac{\begin{array}{l}\textbf{assert } \alpha(x, y)\\ x := f(x, y)\\ L:\end{array}}{L\text{: } \textbf{assert } \alpha(f^{-}(x, y), y)}$$

to the invariant $r > e$, we get

assert $2 \cdot r > e$

at the end of the loop. By applying the *forward loop-body rule* ⟨29⟩

assert α
loop L:
  P
  assert β
  repeat
L: **assert** $\alpha \vee \beta$

taking $r = 1$ for α and $2 \cdot r > e$ for β, we derive the loop invariant

$$L_1: \textbf{assert } r = 1 \vee 2 \cdot r > e. \tag{7}$$

C. Heuristic Rules

Recall that the control rules gave us

if $qq + rr \leqslant c$ **then** $(q, qq) := (q + r, qq + rr)$
  assert $r > e, qq \leqslant c$
else **assert** $r > e, c < qq + rr$
fi,

but that the disjunction of $qq \leqslant c$ and $c < qq + rr$ turned out to be a tautology. The *conditional heuristic* ⟨36⟩

if t **then** P'
  assert α
else P''
  assert β
fi
L:
L: **suggest** α, β

suggests that each of the two invariants $qq \leqslant c$ and $c < qq + rr$ that hold at the end of one of the conditional paths may be an invariant for both paths. So we have the candidates

suggest $qq \leqslant c, \quad c < qq + rr$

following the conditional and preceding the assignment

$(r, rr) := (r/2, rr/2)$.

By application of the *forward assignment rule* ⟨21⟩ to the two candidates, we get

suggest $qq \leqslant c, \quad c < qq + 2 \cdot rr$

at the end of the loop.

Finally, by applying the *forward loop-body rule* ⟨29⟩, we get the candidates

L_1: **suggest** $(q = qq = 0 \wedge r = 1 \wedge rr = d) \vee qq \leqslant c,$
  $(q = qq = 0 \wedge r = 1 \wedge rr = d) \vee c < qq + 2 \cdot rr.$

Both candidates may be simplified, since their first disjunct implies their second, leaving

L_1: **suggest** $qq \leqslant c, \quad c < qq + 2 \cdot rr.$

These two candidates can indeed be proved to be invariants. The first candidate, $qq \leqslant c$, derived from the initialization and **then**-paths, is unaffected by the **else**-path which leaves the value of qq unchanged. Similarly, the other candidate, $c < qq + 2 \cdot rr$, derived from the initialization and **else**-paths, is maintained true by the **then**-path. So we have the loop invariants

$$L_1: \textbf{assert } qq \leqslant c, \quad c < qq + 2 \cdot rr. \tag{8}$$

Note that we have not yet made any use of the candidates

E_1: **suggest** $q \leqslant c/d, \quad c/d < q + e,$

suggested by the output specification. For completeness, we shall apply a heuristic to these candidates, although no new invariants will be derived. The *top-down heuristic* ⟨38⟩

assert α
loop L:
  until t
  P
  repeat
suggest γ

$\alpha \supset \gamma$
L: **suggest** γ

suggests that the output candidate $q \leqslant c/d$ may itself be a loop invariant, since it is true upon entering the loop. Indeed, it is an invariant (it is implied by the loop invariant $qq \leqslant c$ and the global invariant $qq = q \cdot d$). On the other hand, the second output candidate, $c/d < q + e$, does not even hold for the initialization path, when $q = 0$.

Since there are no assignments between the loop and the end of the program, all the loop invariants may be pushed forward unchanged, and hold upon termination. The output invariants include

$$E_1: \textbf{assert } (r = 1 \vee 2 \cdot r > e), qq \leqslant c, c < qq + 2 \cdot rr, r \leqslant e. \tag{9}$$

These invariants, along with the global invariants

assert $rr = d \cdot r, \quad qq = d \cdot q$ **in** P_1,

imply $q \leqslant c/d$ as specified. However, they do not imply $c/d < q + e$, only $c/d < q + 2 \cdot e$. In fact, our program as given is incorrect. For a discussion of how such invariants may be used to guide the debugging of the program, see Dershowitz and Manna [4].

D. Loop Counters

By introducing an imaginary loop counter n—initialized to 0 upon entering the loop and incremented by 1 with each iteration—one may derive relations between the program variables and the number of iterations.

The extended program, annotated with some of the invariants we have already found, is

```
assert rr = d · r,  qq = d · q,  r ∈ 1/2^IN,  rr ∈ IN/2^IN in
P1: begin comment extended real-division program
    B1: assert 0 ≤ c < d,  0 < e
    (q, qq, r, rr) := (0, 0, 1, d)
    n := 0
    loop L1: assert (r = 1 ∨ 2 · r > e),  qq ≤ c,  c < qq + 2 · rr
        until r ≤ e
        if qq + rr ≤ c then (q, qq) := (q + r, qq + rr) fi
        (r, rr) := (r/2, rr/2)
        n := n + 1
        repeat
    E1: assert (r = 1 ∨ 2 · r > e),  qq ≤ c,  c < qq + 2 · rr,  r ≤ e
    end.
```

Obviously, we may

$$\textbf{assert } n \in \mathbb{N} \textbf{ in } P_1. \tag{10}$$

For the variables r and n we have the assignments

$$(r, n) := (1, 0) \quad (r, n) := (r/2, n + 1)$$

and can apply the *linear rule* ⟨14⟩

$$\frac{(x, y) := (a_0, b_0) \quad (x, y) := (x + a_2, b_1 \cdot y + b_2) \textbf{ in } P}{\textbf{assert } [y \cdot (b_1 - 1) + b_3 \cdot x + b_2]^{a_2} \cdot b_1^{a_0} = [b_0 \cdot (b_1 - 1) + b_3 \cdot a_0 + b_2]^{a_2} \cdot b_1^{x} \textbf{ in } P.}$$

With this rule we get the global invariant

$$\textbf{assert } [r \cdot (1/2 - 1) + 0]^1 \cdot (1/2)^0 = [1 \cdot (1/2 - 1) + 0]^1 \cdot (1/2)^n \textbf{ in } P_1,$$

which simplifies to

$$\textbf{assert } r = 1/2^n \textbf{ in } P_1. \tag{11}$$

Applying the same rule to the assignments

$$(rr, n) := (d, 0) \quad (rr, n) := (rr/2, n + 1)$$

we deduce

$$\textbf{assert } rr = d/2^n \textbf{ in } P_1. \tag{12}$$

With these loop-counter invariants, the total number of loop iterations as a function of the input values may be determined. Using (11), we can substitute $1/2^n$ for r in the loop invariant (7) $r = 1 \vee 2 \cdot r > e$ and in the output invariant (9) $r \leq e$ and get $1/2^n = 1 \vee 2/2^n > e$ at L_1 and $1/2^n \leq e$ at E_1. Taking the logarithm (e is positive), we have the upper bound

$$n = 0 \vee n < -\log_2 e + 1$$

and lower bound

$$-\log_2 e \leq n$$

on the number of loop iterations n. Note that by finding a loop invariant giving an upper bound on the number of iterations, we have actually proved that the loop terminates.

Combining both bounds at E_1 gives (assuming $n \neq 0$)

$$-\log_2 e \leq n < -\log_2 e + 1,$$

or, since n is an integer (10), it is equal to the one integer lying between its lower and upper bounds: $-\lfloor \log_2 e \rfloor$. Thus, we have the output invariant

$$E_1: \textbf{ assert } n = 0 \vee n = -\lfloor \log_2 e \rfloor. \tag{13}$$

Since n is the number of times the loop was executed before termination, we have derived the desired expression for the time complexity of the loop.

IV. Discussion

In a sense, annotating programs is "putting the cart before the horse" as the whole tenor of "structured programming" stresses developing invariants hand in hand with the code, and not ex post facto. Nevertheless, the development of automatic annotation systems is important for a number of reasons.

- The real world contains many undocumented, underdocumented, and misdocumented programs. Even annotated programs appearing in structured-programming textbooks have fallen prey to error. A system that could help in documenting such programs would clearly be of utility.
- Ultimately, it is the responsibility of the programmer to guarantee the correctness of his product. Even if he uses one of the current automatic verification systems, he is required to supply most, if not all, of the necessary invariant assertions. The goal of automatic program annotation is to relieve the programmer of this burden. Agreed, no present or future system will discover very subtle invariants or those based on deep mathematical theorems, but such invariants are likely to be uppermost in the programmer's mind anyway. It is the "obvious" invariants that he finds annoying to have to formulate, and indeed often forgets, causing the system to fail in its proof. Fortunately, it is just these invariants that an automatic annotation system would find easy to derive. Similarly, invariants needed to demonstrate the absence of runtime errors are usually quite simple, and there has already been some success in providing current verification systems with the capability of generating them.

• Annotation techniques may be used to discover properties of programs other than correctness, the investigation of which is generally outside the scope of the programmer's expertise. For example, one may wish to analyze the complexity of an algorithm or compare the efficiency of two correct programs. Even simple programs are sometimes very difficult to analyze; invariants may be needed to facilitate such an analysis.

• Annotation research attempts to formalize the intuitions that lie behind well-designed programs; thus, it has implications for automatic program synthesis. Some of the same rules used to generate invariants from programs may be inverted to generate programs from invariants.

Appendix

In this Appendix we present a catalog of annotation rules. We use the following conventions:

P, P', and P'' denote program segments;
L, L', and L'' are statement labels;
α, β, and γ denote predicates;
x, y, and z are variables;
a, a_i, and b_i are expressions that are constant in the given program segment;
u and v are arbitrary expressions;
f and g are arbitrary functions;
$\bar{x}, \bar{y}, \bar{z}, \bar{a}$, and $\bar{u}$ are vectors;
$\mathbf{N}$ denotes the set of natural numbers and $\mathbf{Z}$ the set of all integers.

A. Assignment Rules

The assignment rules all yield global invariants. The notation

$$\bar{x} := \bar{u}_1 \quad \bar{x} := \bar{u}_2 \cdots \text{in } P$$

means that the listed assignments are all the assignments to elements of $\bar{x}$ within P.

• *Range Rules:*

⟨1⟩ *addition rule*

$$\frac{x := a_0 \quad x := x + a_1 \quad x := x + a_2 \cdots \text{in } P}{\text{assert } x \in a_0 + a_1 \cdot \mathbf{N} + a_2 \cdot \mathbf{N} + \cdots \text{in } P}$$

⟨2⟩ *multiplication rule*

$$\frac{x := a_0 \quad x := x \cdot a_1 \quad x := x \cdot a_2 \cdots \text{in } P}{\text{assert } x \in a_0 \cdot a_1^{\mathbf{N}} \cdot a_2^{\mathbf{N}} \cdot \cdots \text{in } P}$$

⟨3⟩ *exponentiation rule*

$$\frac{x := a_0 \quad x := x^{a_1} \quad x := x^{a_2} \cdots \text{in } P}{\text{assert } x \in a_0^{a_1^{\mathbf{N}} \cdot a_2^{\mathbf{N}} \cdot \cdots}} \text{ in } P$$

• *Set Assignment Rules:*

$x :\in S$ assigns some u to x such that $u \in S$;
ΣS is the closure of the set S under +;
ΠS is the closure of the set S under ·.

⟨4⟩ *set-union rule*

$$\frac{x :\in S_0 \quad x :\in S_1 \quad x :\in S_2 \cdots \text{in } P}{\text{assert } x \in S_0 \cup S_1 \cup S_2 \cup \cdots \text{in } P}$$

⟨5⟩ *set-addition rule*

$$\frac{x :\in S_0 \quad x :\in x + S_1 \quad x :\in x + S_2 \cdots \text{in } P}{\text{assert } x \in S_0 + \Sigma S_1 + \Sigma S_2 + \cdots \text{in } P}$$

⟨6⟩ *set-multiplication rule*

$$\frac{x :\in S_0 \quad x :\in x \cdot S_1 \quad x :\in x \cdot S_2 \cdots \text{in } P}{\text{assert } x \in S_0 \cdot \Pi S_1 \cdot \Pi S_2 \cdot \cdots \text{ in } P}$$

⟨7⟩ *set-exponentiation rule*

$$\frac{x :\in S_0 \quad x :\in x^{S_1} \quad x :\in x^{S_2} \cdots \text{in } P}{\text{assert } x \in S_0^{\Pi S_1 \cdot \Pi S_2 \cdot \cdots} \text{ in } P}$$

• *Counter Relation Rules:*

$f(y)$ and $g(y)$ are expressions containing the one variable y;
$g^-(y)$ is the inverse of $g(y)$, i.e., $g^-(g(y)) = y$.

⟨8a⟩ *addition-counter rule*

$$\frac{(x, y) := (a_0, b_0) \quad (x, y) := (x + f(y), g(y)) \text{ in } P}{\text{assert } x = a_0 + f(b_0) + f(g(b_0)) + f(g(g(b_0))) + \cdots + f(g^-(y)) \text{ in } P}$$

⟨8b⟩
$$\frac{(x, y) := (a_0, b_0) \quad (x, y) := (x + f(y), y + 1) \text{ in } P \quad \text{assert } y \in Z \text{ in } P}{\text{assert } x = a_0 + \Sigma_{\zeta = b_0}^{y-1} f(\zeta) \text{ in } P}$$

⟨9a⟩ *multiplication-counter rule*

$$\frac{(x, y) := (a_0, b_0) \quad (x, y) := (x \cdot f(y), g(y)) \text{ in } P}{\text{assert } x = a_0 \cdot f(b_0) \cdot f(g(b_0)) \cdot f(g(g(b_0))) \cdot \cdots \cdot f(g^-(y)) \text{ in } P}$$

⟨9b⟩
$$\frac{(x, y) := (a_0, b_0) \quad (x, y) := (x \cdot f(y), y + 1) \text{ in } P \quad \text{assert } y \in Z \text{ in } P}{\text{assert } x = a_0 \cdot \Pi_{\zeta = b_0}^{y-1} f(\zeta) \text{ in } P}$$

⟨10a⟩ *exponentiation-counter rule*

$$\frac{(x,y) := (a_0, b_0) \quad (x,y) := (x^{f(y)}, g(y)) \text{ in } P}{\text{assert } x = a_0^{f(b_0)\cdot f(g(b_0))\cdot f(g(g(b_0)))\cdot \cdots \cdot f(g^-(y))} \text{ in } P}$$

⟨10b⟩ $$\frac{(x,y) := (a_0, b_0) \quad (x,y) := (x^{f(y)}, y+1) \text{ in } P \quad \text{assert } y \in Z \text{ in } P}{\text{assert } x = a_0^{\prod_{\xi=b_0}^{y-1} f(\xi)} \text{ in } P}$$

- *Basic Relation Rules:*

⟨11⟩ *addition-relation rule*

$$\frac{(x,y) := (a_0, b_0) \quad (x,y) := (x + a_1 \cdot u, y + b_1 \cdot u) \quad (x,y) := (x + a_1 \cdot v, y + b_1 \cdot v) \cdots \text{ in } P}{\text{assert } a_1 \cdot (y - b_0) = b_1 \cdot (x - a_0) \text{ in } P}$$

⟨12⟩ *multiplication-relation rule*

$$\frac{(x,y) := (a_0, b_0) \quad (x,y) := (x \cdot u^{a_1}, y \cdot u^{b_1}) \quad (x,y) := (x \cdot v^{a_1}, y \cdot v^{b_1}) \cdots \text{ in } P}{\text{assert } x^{b_1} \cdot b_0^{a_1} = a_0^{b_1} \cdot y^{a_1} \text{ in } P}$$

⟨13⟩ *exponentiation-relation rule*

$$\frac{(x,y) := (a_0, b_0) \quad (x,y) := (x^{a_1^u}, y^{b_1^u}) \quad (x,y) := (x^{a_1^v}, y^{b_1^v}) \cdots \text{ in } P}{\text{assert } \log(x)^{\log(b_1)} \cdot \log(b_0)^{\log(a_1)} = \log(a_0)^{\log(b_1)} \cdot \log(y)^{\log(a_1)} \text{ in } P}$$

- *Assorted Relation Rules:*

⟨14⟩ *linear rule*

$$\frac{(x,y) := (a_0, b_0) \quad (x,y) := (a_1 \cdot x + a_2, b_1 \cdot y + b_2 + b_3 \cdot x) \text{ in } P}{\begin{aligned}
&\text{assert } [y \cdot (b_1 - 1) + b_3 \cdot x + b_2 + b_3 \cdot a_2/(b_1 - 1)]^{a_2} \cdot b_1^{a_0} \\
&\quad = [b_0 \cdot (b_1 - 1) + b_3 \cdot a_0 + b_2 + b_3 \cdot a_2/(b_1 - 1)]^{a_2} \cdot b_1^{x} \text{ in } P \qquad \text{if } a_1 = 1 \neq b_1 \\
&\text{assert } a_1^{(y-b_0)\cdot(a_1-1)} \cdot [a_0 \cdot (a_1 - 1) + a_2]^{b_2 \cdot (a_1-1) - a_2 \cdot b_3} \\
&\quad = a_1^{(x-a_0)\cdot b_3} \cdot [x \cdot (a_1 - 1) + a_2]^{b_2 \cdot (a_1-1) - a_2 \cdot b_3} \text{ in } P \qquad \text{if } b_1 = 1 \\
&\text{assert } (a_1 - 1) \cdot [(a_1 - 1) \cdot (y \cdot a_0 - x \cdot b_0) + a_2 \cdot (y - b_0)] - (x - a_0) \cdot [b_2 \cdot (a_1 - 1) - a_2 \cdot b_3] \text{ in } P \\
&\quad = (b_3/a_1) \cdot [(a_1 - 1) \cdot a_0 + a_2] \cdot [(a_1 - 1) \cdot x + a_1] \cdot [\log((a_1 - 1) \cdot x + a_2) - \log((a_1 - 1) \cdot a_0 + a_2)]/\log(a_1) \\
&\qquad\qquad \text{if } a_1 = b_1 \neq 1 \\
&\text{assert } [(y \cdot (b_1 - 1) + b_2) \cdot (a_1 - b_1) - b_3 \cdot (x \cdot (b_1 - 1) + a_2)]^{\log(a_1)} \cdot [a_0 \cdot (a_1 - 1) + a_2]^{\log(b_1)} \\
&\quad = [(b_0 \cdot (b_1 - 1) + b_2) \cdot (a_1 - b_1) - b_3 \cdot (a_0 \cdot (b_1 - 1) + a_2)]^{\log(a_1)} \cdot [x \cdot (a_1 - 1) + a_2]^{\log(b_1)} \\
&\qquad\qquad \text{in } P \qquad\qquad \text{otherwise}
\end{aligned}}$$

⟨15⟩ *geometric rule*

$$\frac{(x,y) := (a_0, b_0) \quad (x,y) := (x + a_1, y + b_1 + b_2 \cdot x + b_3^x) \text{ in } P}{\text{assert } 2 \cdot a_1 \cdot (b_3^x - b_3^{a_0}) = (b_3^{a_1} - 1) \cdot [2 \cdot a_1 \cdot (y - b_0) - (x - a_0) \cdot (2 \cdot b_1 + b_2 \cdot (x + a_0 - a_1))] \text{ in } P}$$

⟨16⟩ *factorial rule*

$$\frac{(x,y) := (a_0 \cdot a_1, b_0) \quad (x,y) := (x + a_1, y \cdot x \cdot b_1) \text{ in } P}{\text{assert } y \cdot a_1 = b_0 \cdot (a_1 \cdot b_1)^{\overline{x/a_1 - a_0}} \cdot a_0^{\overline{x/a_1 - a_0}} \text{ in } P}$$

where $u^{\overline{n}} = u \cdot (u+1) \cdot \cdots \cdot (u+n-1)$

⟨17⟩ *multiplication-exponentiation rule*

$$\frac{(x,y) := (a_0, b_0) \quad (x,y) := (x \cdot a_1^u, y^{b_1^u}) \quad (x,y) := (x \cdot a_1^v, y^{b_1^v}) \cdots \text{ in } P}{\text{assert } [x/a_0]^{\log(b_1)} = [\log(y)/\log(b_0)]^{\log(a_1)} \text{ in } P}$$

B. Control Rules

The control rules all yield local invariants. Control rules expressed with **assert** statements in the antecedent may be applied to **suggest** statements, in that case, the consequent is only a candidate.

- *Control Axioms:*

⟨18⟩ *assignment axiom*

$\bar{x} := \bar{a}$
assert $\bar{x} = \bar{a}$

Note that $\bar{a}$ *are constant expressions, i.e.,* $\bar{x}$ *do not appear within them*

⟨19⟩ *conditional axiom*

if t **then assert** t
P'
else assert $\neg t$
P''
fi

⟨20⟩ *loop axiom*

loop P'
until t
assert $\neg t$
P''
repeat
assert t

- *Assignment Control Rules:*

A is an array variable;
the array function $assign(A, y, z)$ yields A, with z replacing $A[y]$.

⟨21a⟩ *forward assignment rule*

assert $\alpha(\bar{x}, \bar{y})$
$\bar{x} := f(\bar{x}, \bar{y})$
L:

L: **assert** $\alpha(f^{-}(\bar{x}, \bar{y}), \bar{y})$

where f^{-} *is the inverse of the function* f, *i.e.,* $f^{-}(f(\bar{x}, \bar{y}), \bar{y}) = \bar{x}$, *and* $\bar{x}$ *do not appear in* $\alpha(\bar{\xi}, \bar{y})$

⟨21b⟩ **assert** $\alpha(\bar{u}, \bar{y})$
$\bar{x} := \bar{u}$
L:

L: **assert** $\alpha(\bar{x}, \bar{y})$

where $\bar{x}$ *do not appear in* $\alpha(\bar{\xi}, \bar{y})$

⟨21c⟩ **assert** $\bar{x} = \bar{a}$
$\bar{x} := f(\bar{x}, \bar{y})$
L:

L: **assert** $\bar{x} = f(\bar{a}, \bar{y})$

Note that $\bar{a}$ *are constant expressions, i.e.,* $\bar{x}$ *do not appear within them*

⟨22⟩ *backward assignment rule*

L:
$\bar{x} := \bar{u}$
assert $\beta(\bar{x}, \bar{y})$

L: **assert** $\beta(\bar{u}, \bar{y})$

⟨23⟩ *forward array-assignment rule*

assert $\alpha(A, \bar{z})$
$A[y] := f(A[y], \bar{z})$
L:

L: **assert** $\alpha(assign(A, y, f^{-}(A[y], \bar{z})), \bar{z})$

where $f^{-}(f(A[y], \bar{z}), \bar{z}) = A[y]$

⟨24⟩ *backward array-assignment rule*

L:
$A[y] := u$
assert $\beta(A, \bar{z})$

L: **assert** $\beta(assign(A, y, u), \bar{z})$

- *Conditional Control Rules:*

⟨25⟩ *forward test rule*

assert α
if t **then** L':
P'
else L'':
P''
fi

L': **assert** α, t
L'': **assert** $\alpha, \neg t$

⟨26⟩ *backward test rule*

L:
if t **then assert** α
P'
else assert β
P''
fi

L: **assert** $t \supset \alpha, \neg t \supset \beta$

⟨27⟩ *forward branch rule*

if t **then** P'
assert α
else P''
assert β
fi
L:

L: **assert** $\alpha \vee \beta$

⟨28⟩ *backward branch rule*

if t **then** P'
L':
else P''
L'':
fi
assert β

L', L'': **assert** β

- *Loop Control Rules:*

⟨29⟩ *forward loop-body rule*

assert α
loop L:
P
assert β
repeat

L: **assert** $\alpha \vee \beta$

⟨30⟩ *backward loop-body rule*

$$\frac{\begin{array}{l}L'\text{:}\\ \textbf{loop assert } \beta\\ \quad P\\ \quad L''\text{:}\\ \quad \textbf{repeat}\end{array}}{L', L''\text{: assert } \beta}$$

⟨31⟩ *forward loop-exit rule*

$$\frac{\begin{array}{l}\textbf{loop } P'\\ \quad \textbf{assert } \alpha\\ \quad \textbf{until } t\\ \quad L'\text{:}\\ \quad P''\\ \quad \textbf{repeat}\\ L''\text{:}\end{array}}{\begin{array}{l}L'\text{: assert } \alpha, \neg t\\ L''\text{: assert } \alpha, t\end{array}}$$

⟨32⟩ *backward loop-exit rule*

$$\frac{\begin{array}{l}\textbf{loop } P'\\ \quad L\text{:}\\ \quad \textbf{until } t\\ \quad \textbf{assert } \alpha\\ \quad P''\\ \quad \textbf{repeat}\\ \textbf{assert } \beta\end{array}}{L\text{: assert } \neg t \supset \alpha,\ t \supset \beta}$$

• *Value Rules:*

x_L denotes the value of the variable x when control was last at label L.

⟨33⟩ *label axiom*

$$L\text{: assert } x = x_L$$

N.B.: An invariant containing x_L may not be pushed over the label L.

⟨34⟩ *protected-invariant rule*

$$\frac{\begin{array}{l}\textbf{assert } \alpha(x)\\ \textbf{loop } L\text{:}\\ \quad P\\ \quad \textbf{assert } \alpha(x) \vee x = x_L\\ \quad \textbf{repeat}\end{array}}{L\text{: assert } \alpha(x)}$$

where x is the only variable in α

⟨35⟩ *forall rule*

$$\frac{\begin{array}{l}\textbf{assert } x = a,\ \ x \in \mathbf{Z}\\ \textbf{loop } L\text{: assert } \alpha(x)\\ \quad P\\ \quad \textbf{assert } x = x_L + 1\\ \quad \textbf{repeat}\end{array}}{L\text{: assert } (\forall \zeta \in \mathbf{Z})(a \leqslant \zeta \leqslant x)\ \alpha(\zeta)}$$

where x is the only variable in α

C. Heuristic Rules

The heuristic rules all yield local candidate assertions.

⟨36⟩ *conditional heuristic*

$$\frac{\begin{array}{l}\textbf{if } t \textbf{ then } P'\\ \quad \textbf{assert } \alpha\\ \textbf{else } P''\\ \quad \textbf{assert } \beta\\ \textbf{fi}\\ L\text{:}\end{array}}{L\text{: suggest } \alpha, \beta}$$

⟨37a⟩ *generalization heuristic*

$$\frac{\begin{array}{l}\textbf{assert } \bar{x} = \bar{a}\\ \textbf{loop } L\text{: suggest } \alpha(\bar{x}, \bar{y})\\ \quad P\\ \quad \textbf{assert } \bar{x} = f(\bar{x}_L)\\ \quad \textbf{repeat}\end{array}}{L\text{: suggest } (\forall \zeta \in \{\bar{a}, f(\bar{a}), f(f(\bar{a})), \cdots, \bar{x}\})\ \alpha(\zeta, \bar{y})}$$

⟨37b⟩

$$\frac{\begin{array}{l}\textbf{assert } x = a, x \in \mathbf{Z}\\ \textbf{loop } L\text{: suggest } \alpha(x, \bar{y})\\ \quad P\\ \quad \textbf{assert } x = x_L + 1\\ \quad \textbf{repeat}\end{array}}{L\text{: suggest } (\forall \zeta \in \mathbf{Z})(a \leqslant \zeta \leqslant x)\ \alpha(\zeta, \bar{y})}$$

⟨38⟩ *top-down heuristic*

$$\frac{\begin{array}{l}\textbf{assert } \alpha\\ \textbf{loop } L\text{:}\\ \quad \textbf{until } t\\ \quad P\\ \quad \textbf{repeat}\\ \textbf{suggest } \gamma\end{array}}{\begin{array}{l}\alpha \supset \gamma\\ \hline L\text{: suggest } \gamma\end{array}}$$

• *Dangerous Heursitics:*

These heuristics should be applied with caution.

⟨39⟩ *disjunction heuristic* (applied in conjunction with the *forward branch rule*)

$$\frac{L\text{: assert } \alpha \vee \beta}{L\text{: suggest } \alpha, \beta}$$

⟨40⟩ *strengthening heuristic* (applied in conjunction with the *top-down heuristic*)

$$\frac{\begin{array}{l}L\text{: assert } \alpha(\bar{x})\\ \textbf{suggest } \gamma(\bar{x})\end{array}}{L\text{: suggest } (\forall \bar{x})(\alpha(\bar{x}) \supset \gamma(\bar{x}))}$$

⟨41⟩ *transitivity heuristic* (applied in conjunction with the *top-down heuristic*)

$$\frac{\begin{array}{l}L\text{: assert } uRv\\ \textbf{suggest } uRw\end{array}}{L\text{: suggest } vRw \vee v = w}$$

where R is a transitive relation, i.e., $uRv \wedge vRw \supset uRw$.

ACKNOWLEDGMENT

We thank D. Harel, S. M. Katz, J. C. King, L. Paulson, W. Polak, R. J. Waldinger, and anonymous referees for their critical readings of the manuscript.

REFERENCES

[1] S. K. Basu, "A note on synthesis of inductive assertions," *IEEE Trans. Software Eng.*, vol. SE-6, pp. 32–39, Jan. 1980.

[2] S. K. Basu and J. Misra, "Proving loop programs," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 76–86, Mar. 1975.

[3] M. Caplain, "Finding invariant assertions for proving programs," in *Proc. Int. Conf. Reliable Software*, Los Angeles, CA, Apr. 1975, pp. 165–171.

[4] N. Dershowitz and Z. Manna, "The evolution of programs: Automatic program modification," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 377–385, Nov. 1977.

[5] N. Dershowitz, "The evolution of programs," Ph.D. dissertation, Weizmann Inst. Sci., Rehovot, Israel, Nov. 1978.

[6] B. Elspas, "The semiautomatic generation of inductive assertions for proving program correctness," Interim Rep., Project 2686, SRI Int., Menlo Park, CA, July 1974.

[7] S. M. German, "A program verifier that generates inductive assertions," undergraduate thesis, Memo. TR-19-74, Harvard Univ., Cambridge, MA, May 1974.

[8] —, "Automatic proofs of the absence of common runtime errors," in *Conf. Rec. 5th ACM Symp. Principles of Programming Languages*, Tucson, AZ, Jan. 1978, pp. 105–118.

[9] S. M. German and B. Wegbreit, "A synthesizer of inductive assertions," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 68–75, Mar. 1975.

[10] A. Gibb, "Algorithm 61: Procedures for range arithmetic," *Commun. Ass. Comput. Mach.*, vol. 4, pp. 319–320, July 1961.

[11] I. Greif and R. J. Waldinger, "A more mechanical heuristic approach to program verification," in *Proc. Int. Symp. Programming*, Paris, France, Apr. 1974, pp. 83–90.

[12] W. H. Harrison, "Compiler analysis of the value ranges for variables," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 243–250, May 1977.

[13] C. A. R. Hoare, "An axiomatic basis of computer programming," *Commun. Ass. Comput. Mach.*, vol. 12, pp. 576–580, 583, Oct. 1969.

[14] S. M. Katz, "Invariants and the logical analysis of programs," Ph.D. dissertation, Weizmann Inst. Sci., Rehovot, Israel, Sept. 1976.

[15] S. M. Katz and Z. Manna, "A heuristic approach to program verification," in *Adv. Papers 3rd Int. Conf. Artificial Intell.*, Stanford, CA, Aug. 1973, pp. 500–512.

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

[17] D. E. Knuth, *The Art of Computer Programming, vol. 1: Fundamental Algorithms.* Reading, MA: Addison-Wesley, 1968.

[18] —, "Structured programming with **go to** statements," *Comput. Surveys*, vol. 6, pp. 261–301, Dec. 1974.

[19] D. C. Luckham and N. Suzuki, "Proof of termination within a weak logic of programs," *Acta Informatica*, vol. 8, pp. 21–36, Mar. 1977.

[20] J. McCarthy and C. Talcott, "LISP programming and proving," Dep. Comput. Sci., Stanford Univ., Stanford, CA, manuscript, 1978.

[21] Z. Manna and A. Pnueli, "Axiomatic approach to total correctness of programs," *Acta Informatica*, vol. 3, pp. 243–263, July 1974.

[22] M. S. Moriconi, "Towards the interactive synthesis of assertions," Automat. Theorem Proving Project, Univ. Texas, Austin, Memo. ATP-20, Oct. 1974.

[23] J. H. Morris and B. Wegbreit, "Subgoal induction," *Commun. Ass. Comput. Mach.*, vol. 20, pp. 209–222, Apr. 1977.

[24] I. Netzer, "Logical analysis of recursive programs," M.S. thesis, Weizmann Inst. Sci., Rehovot, Israel, Apr. 1976.

[25] W. Scherlis, "On the weak interpretation method for extracting program properties," undergraduate thesis, Harvard Univ., Cambridge, MA, May 1974.

[26] M. Sintzoff, "Calculating properties of programs by valuations on specific models," in *Proc. ACM Conf. Proving Assertions About Programs*, Las Cruces, NM; also in *SIGPLAN Notices*, vol. 7, pp. 203–207, Jan. 1972.

[27] N. Suzuki and K. Ishihata, "Implementation of an array bound checker," in *Conf. Rec. 4th ACM Symp. Principles of Programming Languages*, Los Angeles, CA, Jan. 1977, pp. 132–143.

[28] M. Tamir, "ADI–Automatic derivation of invariants," M.S. thesis, Weizmann Inst. Sci., Rehovot, Israel, Aug. 1976.

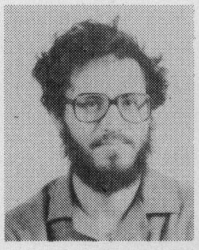
[29] —, "ADI–Automatic derivation of invariants," *IEEE Trans. Software Eng.*, vol. SE-6, pp. 40–48, Jan. 1980.

[30] R. C. Waters, "A method for analyzing loop programs," *IEEE Trans. Software Eng.*, vol. SE-5, pp. 237–247, May 1979.

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

[32] —, "Property extraction in well-founded property sets," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 270–285, Sept. 1975.

[33] B. Wegbreit and J. M. Spitzen, "Proving properties of complex data structures," *J. Ass. Comput. Mach.*, vol. 23, pp. 389–396, Apr. 1976.



Nachum Dershowitz received the B.S. degree from Bar-Ilan University, Ramat-Gan, Israel, in 1974, and the M.S. and Ph.D. degrees in applied mathematics from the Weizmann Institute of Science, Rehovot, Israel, in 1975 and 1979, respectively.

He did his doctoral research in program development while at the Artificial Intelligence Laboratory, Stanford University, Stanford, CA, from 1975 to 1978. Currently, he is an Assistant Professor of Computer Science at the University of Illinois, Urbana, IL. His research interests include program development and verification.

Zohar Manna received the B.S. and M.S. degrees in mathematics from the Technion–Israel Institute of Technology, Haifa, Israel, in 1961 and 1965, respectively, and the Ph.D. degree in computer science from Carnegie-Mellon University, Pittsburgh, PA, in 1968.

From 1968 to 1972 he was an Assistant Professor of Computer Science at Stanford University, Stanford, CA. From 1972 to 1975 he was an Associate Professor, and from 1975 to date a Professor, at the Weizmann Institute of Science, Rehovot, Israel. Since 1978 he has also been a Professor of Computer Science at Stanford University, Stanford, CA. His current research interests include program verification, methodology of programming, program synthesis, and the various areas of the mathematical theory of computation. He is the author of the book *Mathematical Theory of Computation* (New York: McGraw-Hill).