Stanford Artificial Intelligence Laboratory
Memo AIM-303

October 1977

Computer Science Department
Report No. STAN-CS-7 7-63 1

INFERENCE RULES FOR PROGRAM ANNOTATION

by

Nachum Dershowitz and Zohar Manna

Research sponsored by

United States Air Force

Nat ional Science Foundat ion

Advanced Research Projects Agency
ARPA Order No. 2494

COMPUTER SCIENCE DEPARTMENT
Stanford University

Stanford Artificial Intelligence Laboratory
Memo AIM-303

October 1977

Computer Science Department
Report No. STAN-CS-77-631

INFERENCE RULES FOR PROGRAM ANNOTATION

by

Nachum Dershowitz and Zohar Manna

ABSTRACT

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

The annotation techniques are formulated as Hoare-like inference rules which 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 two examples which have run on an experimental implementation.

The authors are also affiliated with the Department of Applied Mathematics of the Weizmann Institute of Science, Rehovot, Israel.

This research was supported in part by the United States Air Force Office of Scientific Research under Grant RFOSR-76-2909 (sponsored by the Rome Air Development Center, Griffiss AFB, NY), by the National Science Foundation under Grant MCS 76-83655 and by the Advanced Research Projects Agency of the Department of Defense under Contract MDA 903-76-C-0206.

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 or the U. S. Government.

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

I. INTRODUCTION

A convenient form for expressing many facts about a program is a set of *invariant assertions (invartants,* for short) which detail relationships 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 Algoi-like program along with an *output specification* stating the desired relationship 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 those specifications. Our task is to generate the invariant assertions describing the workings of the program as is, independent of its correctness or incorrectness.

in the following sections, we present a unified approach to program annotation, using *annotation rules* — in the style of Hoare [1969] — to derive invariants, Section ii presents an overview of our approach. it is followed by two detailed examples: the first (Section ii I) illustrates the basic technique8 on a single-loop program; the second (Section IV) applies the techniques to a program with nested loops and arrays, A catalog of annotation rules is included in the Appendix.

We have implemented the strategies described in this paper in QLISP (Wiiber [1976]), which resides in an INTERLISP environment (Teiteiman [1974]). The two examples presented here are among those that have run successfully on our experimental system. Three earlier annotation systems are:

- the system described in Eispas [1974], based mainly upon the solution of difference equations;
- VISTA (German [1974], German and Wegbreit [1975]), based upon the top-down heuristics of Wegbreit [1974]; and
- ADI (Tamir [1976]), an interactive system based upon the method8 of Katz and Manna [1976] and Katz [1976].

Our system, as described here, attempts to incorporate and expand upon those systems.

II. OVERVIEW

in this section, we first define some terminology and then, in an attempt to impart the flavor of the general approach, present samples of each type of annotation rule.

1. 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 prove its correctness; if the program is incorrect, we would like information helpful in determining what is wrong with it. Three types of invariant8 will play a role in our discussion:

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

  $\{ \alpha \}$ in P

  to indicate that the relation α is a global invariant in the program segment 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,

  $\{ \alpha \}$ at L

  means that the relation α holds each time control is at label L.
- *Candidates for invariants,* also associated with specific points, are relations that are believed to be local invariants, but which have not yet been verified. Using question marks to emphasize that these relations are just candidates, we write

  $\{? \alpha ?\}$ at L .

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
    B0: { c ∈ N , d ∈ N+ }
    q := 0
    r := c
    loop L0: { . . . }
            until r < d
            q := q+1
            r := r-d
            repeat
    E0: {? q ∈ N , q ≤ c/d , c/d < q+1 , r = c-q•d ?}
    end ,
```

where N is the set of natural numbers, and N^+ is the set of positive integers. We use the **loop-until-repeat** construct, to Indicate that the two loop-body assignments, $q := q+1$ and $r := r-d$, are repeated 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; examples of full annotation are given in Sections lit and IV.

The invariant

$$\{ c \in N , d \in N^+ \}$$

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 Is a positlve integer. The input specification is assumed to hold, regardless of whether the program is correct or not. Since it is a local invariant at B_0, we refer to it as

$$\{ c \in N , d \in N^+ \} \text{ at } B_0 .$$

The candidate

$$\{? q \in N , q \le c/d , c/d < q+1 , r = c-q•d ?\}$$

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 that is not larger than

c/d and r be the remainder. Since one cannot assume that the programmer has not erred, initially ail programmer-supplied assertions — including the program's output 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

$$\{?\ r = c-q\cdot d\ ?\} \quad \text{at} \quad L_0 .$$

To prove that it is an invariant, one must show that the relation holds at L_0 when the loop is first entered, and 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

$$\{\ r = c-q\cdot d\ \} \text{ at } L_0 .$$

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

$$\{?\ _q \in N\ ,\ _q \le c/d\ ,\ \ c/d\ < q+1\ ?\} \text{ and } \{r = c-q\cdot d\} \quad \text{at} \quad E_0 .$$

Global invariants often express the range of variables. For example, since the variable q is first initialized to zero and is subsequently incremented by ones, it is obvious that the value of q is always a natural number, Thus we have the global invariant

$$\{\ q \in N\ \} \quad \text{in} \quad P_0$$

which relates to the program as a whole, and states that $q \in N$ throughout execution of the program segment 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. We list about forty such rules in the Appendix; they are numbered < 1>,<2>, etc. 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. Not only are these

rules useful for automatlo annotation, but they may also help oiarity the relationships between program text and invariant8 for the human programmer.

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, though promising, are not guaranteed to be invarlants.

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, ref iecting common programming practice.

2. 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 the 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: an *addition* rule, which gives the range of a variable which is updated by adding (or subtracting) a constant; a *set-addition* rule for the case where the variable is added to another variable whose range is already known; and an *addition-relation rule* which relates two variables that are always incremented by similar expressions. Corresponding rules apply to other operators,

in- dealing with sets, we find the following notation convenient: The set of elements $f(s_1, s_2, \ldots, s_m)$ such that $s_1 6 S_1, s_2 \in S_2, , , , , s_m \in S_m$ — where f is any expression and $m \geq 0$ — is denoted by $f(S_1, S_2, \ldots, S_m)$. For example, since N denotes the set of natural numbers, the set $f(N, N) = a_0 + N \cdot a_1^N$ contain8 all elements $a_0 + m \cdot a_1^n$ such that m and n are natural numbers.

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

$$\frac{x := a_0 \mid x+a_1 \mid x+a_2 \mid \ldots \quad \text{in } P}{\{x \in a_0+a_1 \cdot N+a_2 \cdot N+\ldots) \quad \text{in } P},$$

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

$$x := a_0 \mid xta_1 \mid \text{xta}_2 \mid \ldots \quad \text{in } P$$

indicates that the *only* assignments to the variable x in P are $x :=$ a, , $x :=$ xta, , $x :=$ xta_2 , etc. The consequent

$$\{x \in a_0+a_1 \cdot N+a_2 \cdot N+\ldots) \quad \text{in } P$$

is a global invariant indicating that x belongs to the set $a_0+a_1 \cdot N+a_2 \cdot N+\ldots$ throughout execution of P — but only from the point when x first receives a defined value in P. [After any execution of $x :=$ a_0, clearly $x \in a_0+a_1 \cdot N+a_2 \cdot N+\ldots$ with $x = a_0+a_1 \cdot 0+a_2 \cdot 0+\ldots$, and if $x = a_0+a_1 \cdot m+a_2 \cdot n+\ldots$ for some m , n , . . . before executing $x := x+a_1$, then $x = a_0+a_1 \cdot (m+1)+a_2 \cdot n+\ldots$ after executing the assignment. Thus, m represents the number of executions of $x :=$ xta, since $x :=$ a, was executed last, n 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, e.g., Gibb [1961]). 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,, the invariant may not hold.

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

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

So we can apply the *addition rule,* letting $a_0 = 0$ and $a_1 = 1$, and obtain the global invariant $q \in 0+1 \cdot N$, i.e.,

$$\{q \in N) \quad \text{in } P_0 .$$

The assignments to r in P_0 are

$r := c \qquad r := r-d$,

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

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

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

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 of $f(S)$ is assigned to x. Note that an assignment $x := f(r)$, where it is only known that $s \in S$, may be viewed as the nondeterministic assignment $x :\in f(S)$. The ***set-addition rule*** **<5>** is

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

where ΣS denotes the set of sums $s_1+s_2+\ldots+s_m$ for (not necessarily distinct) addends s_i in S . if $m = 0$, the sum is 0; if S contains the single element s, then $\Sigma S = s \cdot N$, (This rule applies analogously to any associative and commutative operator "⊕".) These assignment rules for global invariant8 are related to the weak interpretation method of Sintzoff [I 972] (see also Wegbreit [1975] and Harrison [1977]) which has been implemented by Scheriis [1974].

In our program P_0, the assignments to r were

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

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

$r :\in N \qquad r :\in r-N^+$,

and by applying the *set-addition rule,* we obtain the global invariant $r \in N-\Sigma N^+$. This simplifies to

$\{ r \in I \}$ in P_0 ,

where I is the set of ail integers.

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

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

where u , v , . . . , are arbitrary (not necessarily constant) expressions. The invariant begins to hold when the multiple assignment $(x\,,\,y) := (a_1,\ b_1)$ has been executed for the first time. [The invariant $a_1{\cdot}(y{-}b_0) = b_1{\cdot}(x{-}a_0)$ clearly holds when $x = a_1$ and $y = b_1$. 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 stilt holds.] The multiple assignments in the antecedent of the rule, 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 $a_0 = 0$, $b_1 = c$, $a_1 = u = v = 1$, and $b_1 = -d$, we derive the invariant $1{\cdot}(r{-}c) = -d{\cdot}(q{-}0)$, which simplifies to

$$\{\ r = c{-}q{\cdot}d\ \} \quad \text{in } P_0\ .$$

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

$$\frac{(x\,,\,y) := (a_0\,,\,b_0) \mid (x{\oplus}(u{\otimes}a_1)\,,\,y{\oplus}(u{\otimes}b_1)) \mid (x{\oplus}(v{\otimes}a_1)\,,\,y{\oplus}(v{\otimes}b_1)) \mid \ldots \quad \text{in } P}{\{\ (a_0{\otimes}b_1){\oplus}(y{\otimes}a_1) = (x{\otimes}b_1){\oplus}(b_0{\otimes}a_1)\ \} \quad \text{in } P\ ,}$$

where the operator $\oplus$ is commutative and associative, operator $\otimes$ satisfies $(a \otimes b) \otimes c = (a \otimes c) \otimes b$, and $(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$. (These relation rules are related to the approach in Captain [1975].)

Before turning to the control rules, we mention an additional useful technique: the augmentation of a program with counters. 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. (The variable q serves a loop counter in the example program P_0.) By deriving upper/lower bounds on the counter upon loop exit, the termination of the loop may be proved and time complexity analyzed. Loop counters may also be used to discover relations between variables by solving first-order difference equations (see, *e.g.*, Elspas [1974] and Katz and Manna [1976]).

3. Control Rules

Unlike the previous rules which completely ignore the control structure of the program, there are also *control rules* that derive important invariants from the program structure. (They are related to the verification rules of Hoare [1969].) For example, the *forward loop-exit rule* (3 1),

loop P'
{ α }
until t
L':
P''
repeat
L'':

{ α, $\neg t$ } at L'
{ α, t } at L'',

reflects the fact that if a loop is exited and control is 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. The *forward loop-body rule* (29),

$\{ \alpha \}$
loop L:
P
$\{ \beta \}$
repeat

$\{ \alpha \vee \beta \}$ **at** L ,

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 with an invariant β known to hold at the end of the loop body, must hold at L .

Applying the first 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:

$q := 0$
$r := c$
loop L_1:
until $r < d$
$\{ r \geq d \}$
$q := q+1$
$r := r-d$
repeat
E_0: $\{ r < d \}$.

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

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

$\{ \alpha(f^{-}(x , y) , y) \}$ *at* L ,

where f^{-} is the inverse of the function f in the first argument, i.e., $f^{-}(f(x , y) , y) = x$, In

our example, since the first loop-body assignment $q := q+1$ does not affect any variable appearing in the Invariant $r \geq d$, the invarlant is pushed forward unchanged. To propagate $r \geq d$ past the second assignment, $r := r-d$, we replace r by the inverse of $r-d$, that Is $r+d$, yielding $r+d \geq d$, or

$$\{\ r \geq 0\ \}\ ,$$

at the end of the loop body.

The *assignment axiom* **(18)**,

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

(the expression a may not contain x), gives us the invariant

$$\{\ r = c\ \}$$

prior to entering the loop. Thus, by the *forward loop-body rule* **(29)**, *we* get the loop invariant

$$\{\ r = c \vee r \geq 0\ \} \quad \text{at } L_0\ .$$

Since, by the input specification $0 \leq c$, the first disjunct Is subsumed by the second, *i.e.,* if the first disjunct is true, then the second must also hold, and the Invariant simplifies to

$$\{\ r \geq 0\ \} \quad \text{at } L_0\ .$$

To generate invariants from conditional statements, *we* have a *forward test* ***rule*** **(25)**:

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

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**. And anything that held before the test, holds after.

To Illustrate the control rules, consider the following single-loop, single-condltlonal, program schema:

```
P*: begin
    z := c
    loop   L*: { . . . }
           until t(z)
           z := f(2)
           if s(z) then 2 := g(z) else 2 := 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>, when applied to the initial assignment $z := c$, yields the invariant**

$$\{\ z = c\ \}$$

before the loop. The *forward loop-exit rule* **<31> generates the invariant** $-t(z)$ **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. So far we have the loop body**

```
until t(x)
2 := f(2)
( ¬t(f⁻(z)) )
if s(2) then 2 := g(2) else 2 := h(z) fi .
```

The *forward test rule* **<25> propagates that invariant forward and adds** $s(2)$ **at the head of the then clause of the conditional, and** $\neg s(z)$ **at the head of the else clause:**

```
if s(z) then { ¬t(f⁻(z)) A s(z) }; z := g(z)
        else { ¬t(f⁻(z)) ∧ ¬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(x) , we get**

```
if s(z) then z := g(r); { ¬t(f⁻(g⁻(z))) ∧ s(g⁻(z)) }
        else z := h(z); { ¬t(f⁻(h⁻(z))) ∧ ¬s(h⁻(z)) }
        fi .
```

After a conditional statement, we know that one of the two branches must have been taken. This is expressed by the *forward branch rule* (27)

```
if   t then P' ; { α )
      else   P'' ; ( β )
      fi
L:
─────────────────────────
{ α ∨ β }   at L .
```

Thus, by disjoining the invariants from the two different paths, one gets

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

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

```
{ z = c }
loop   L*: ( . . . }
       until t(z)
       z :-f(z)
       if s(2) then 2 := g(z) else 2 := h(z) fi
       { [¬t(f⁻(g⁻(z))) ∧ s(g⁻(z))] ∨ [¬t(f⁻(h⁻(z))) ∧ ¬s(h⁻(z)) ] }
       repeat ,
```

we derive the loop invariant

$$\{\ z = c \ \vee [\neg t(f^-(g^-(z))) \ A \ s(g^-(z))] \ \vee \ [\neg t(f^-(h^-(z))) \ A \ \neg s(h^-(z))]\ \} \text{ a t } L^* \ .$$

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

- *exit lemma:* **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,

$$\{\ z = c \ \vee\ \neg t(f^-(g^-(z)))\ \vee\ \neg t(f^-(h^-(z)))\ \}\quad \text{at}\ L^*\ .$$

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.

- *test lemma:* 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,**

$$\{\ z = c \ \vee\ s(g^-(z))\ \vee\ \neg s(h^-(z))\ \}\quad \text{at}\ L^*\ .$$

The following *forall rule* (35) is valuable for programs with universally-quantified **output specification. Given a loop Invariant** $a(x)$ **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**

$$\frac{\begin{array}{l}\{\ x = a\ \}\\ \textbf{loop}\ L\text{:}\ \{\ \alpha(x)\ \}\\ \qquad P\\ \qquad \{\ x = x_L + 1\ \}\\ \qquad \textbf{repeat}\end{array}}{\{\ (\forall l \in I)(a \le l \le x)\alpha(l)\ \}\quad \text{at}\ L\ ,}$$

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 [1973].) The rule may be broadened to** apply when x is increasing by an **amount-other than I, or for a decreasing** x. **Note that any loop counter will satisfy the conditions on** x.

As **a** simple example, consider the loop

```
i :- 0
loop L:
     until t(i)
     i := i+1
     repeat
E: .
```

We clearly have i = 0 upon entering the loop, and $i = i_L + 1$ at the end of the loop body. By the *exit lemma, we* have

$$\{\ i = 0 \ \vee\ \neg t(i-1)\ \}\ \text{at}\ L\ ,$$

and generalization of this invariant yields $(\forall l)(0 \le l \le i)(l = 0 \vee \neg t(l-1))$ at L, Simplifying, we get

$$\{\ (\forall l)(0 \le l < i) \neg t(l)\ \}\ \text{at}\ L\ .$$

This invariant may be pushed forward to E, where we also have the Invariant $t(i)$, Together they imply

$$\{\ i = \min_{l \ge 0} t(l)\ \}\ \text{at}\ E\ .$$

4, Heuristic Rules

In contrast with the above rules which are algorithmic in the sense that they derive relations that are guaranteed to be Invariants, there Is another class of rules, *heuristic rules,* that can only suggest candidates for invariants. These candidates must be verified. [Since we have not implemented a theorem prover, our system suggests candidates, but does not verify them.]

As an example, consider the following *disjunction heuristic* <36>

```
- if t  then  P' ; { α }
       else  P'' ; { β }
       fi
  L:
-------------------------
{? α , β ?} at L .
```

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.

Wegbreit [1974] and Katz and Manna [1976] have suggested a more general form of this heuristic ⟨39⟩:

$$\frac{\{\ \alpha \vee \beta\ \}\quad \text{at } L}{\{?\ \alpha\ ,\ \beta\ ?\}\quad \text{at } L}\ .$$

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

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, and the *top-down heuristic* ⟨38⟩,

$$\frac{\begin{array}{l}\{\ \gamma\ \}\\ \textbf{loop}\\ \quad L':\\ \quad \textbf{until } t\\ \quad P''\\ \quad \textbf{repeat}\\ L'':\ \{?\ \gamma\ ?\}\end{array}}{\{?\ \gamma\ ?\}\ \text{at } L'}\ ,$$

may be used to push a candidate backwards into a loop, Though $t \supset \gamma$ would be a sufficiently strong loop invariant at L' to establish γ at L'' 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 antecedent of the rule requires the invariant γ before the loop. If some β, rather than γ, is known at that point, then for the heuristic to be applied, β must imply γ.

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

$$\{?\ q \in N,\ q \le c/d\ ,\ c/d < q+1\ ,\ r = c-q\cdot d\ ?\} \quad \text{at} \quad E_0\ ,$$

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

$$\{?\ q \in N\ ,\ q \le c/d\ ,\ r = c-q\cdot d\ ?\} \quad \text{at} \quad L_0\ .$$

The third 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 et L_0, the first, $q \in N$, and last, $r = c-q\cdot d$, have already been shown to be global invariants. To prove that the second, $q \le c/d$, is a loop invariant at L_0, we first try to show that it is true when the loop is entered, *i.e.*, that

$$0 \le c/d\ .$$

The truth of this condition follows from the input specifications. Then we try to show that if $q \le c/d$ is true at L_0, and assuming that the loop Is not exited, then it holds when control returns to L_0, *i.e.*,

$$q \le c/d \quad A \quad r \ge d \supset \quad q+1 \le c/d\ .$$

This condition, however, does not hold. Nevertheless, we can show that $q \le c/d$ is an invariant by using other invariants: We have seen why $r \ge 0$ and $r = c-q\cdot d$ are loop invariants at L_0. Since substituting $c-q\cdot d$ for r in $r \ge 0$ yields $c-q\cdot d \ge 0$, it follows that $q \le c/d$ is also an invariant at L_0. Thus, while an attempt to directly verify the candidate $q \le c/d$ failed, once we have established that $r \ge 0$ and $r = c-q\cdot d$ are invariants, we can also show that $q \le c/d$ is an invariant.

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

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 for a proof to carry through. 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 $(r \geq d) \vee (r = c - q \cdot d)$ 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.*,

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

does not hold. 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,

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

does hold, Clearly, once we establish that $r = c - q \cdot d$ is an invariant, it follows that $(r \geq d) \vee (r = c - q \cdot d)$ also is.

Various specific methods of strengthening candidates have been discussed in the literature (Wegbreit [1974], Katz and Manna [1976], Moriconi [1974] and others); they are closely related to methods of "top-down" structured programming. Related techniques are used by Greif and Waldinger [1974] and Suzuki and Ishihata [1977]. Also the candidates that Misra [1975] and Morris and Wegbreit [1977] derive, using the *subgoal-induction* method of verification, fall into this class.

In each of the following two sections, we shall demonstrate how a nontrivial program can be annotated using the rules in the Appendix. These examples are deliberately taken from previously published papers on program annotation in order to demonstrate the power of our approach.

III. EXAMPLE: Real-Division Program

Consider the following program P, purporting to approximate the quotient c/d of two real numbers c and d, where $0 \le c < d$. 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. in other words, the input specification is

$$0 \le c < d \quad A \quad 0 < e$$

and the output specification is

$$q \le c/d \quad A \quad c/d < q+e \ .$$

The program is

```
P,: begin comment real division
    B , : { 0 ≤ c < d , 0 < e }
    q := 0; qq := 0; r := 1; rr := d
    loop   L1: ( . . . }
           until r ≤ e
           if qq+rr ≤ c then q := q+r; q q := qq+rr fi
           r := r/2; rr := rr/2
           repeat
    E,:  {? q ≤ c/d , c/d < q+e ?}
    end
```

and our goal is to find loop invariants at L, 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 a heuristic rule.

1. Assignment Rules

As a first step we attempt to derive simple Invariants by ignorlng 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 invariant8 by considering all assignments to aoh variable. For example, since the assignments to r are

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

we can apply the *multiplication rule* (2)

$$\frac{x := a_0 \mid x \cdot a_1 \quad \text{in } P}{\{\ x \in a_0 \cdot a_1^N\ \} \text{ in } P} \ .$$

Taking r for x, 1 for a, and $1/2$ for a, , we derive the global invariant

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

in other words, $r = 1/2^n$ for some natural number n. From this it is possible to derive lower and upper bounds on r, *i.e.*, $0 < r \leq 1$.

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

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

yields

$$\{\ rr \in d/2^N\ \} \quad \text{in } P, \quad . \tag{2}$$

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

The assignments to q are

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

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

$$q :\in 0 \qquad q :\in q + 1/2^N \ .$$

Using the *set-addition rule* (5)

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

we conclude

$$\{\ q \in \Sigma 1/2^N\ \} \quad \text{in} \quad P_1\ . \tag{3}$$

This invariant states that q is a finite sum of elements of the form $1/2^n$, where n is some natural number. 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 $m, n \in N$.

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

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

we get by the same *set-addition rule*

$$\{\ qq \in d \cdot \Sigma 1/2^N\ \} \quad \text{in} \quad P,\quad . \tag{4}$$

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

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

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⟩

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

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

$$\{\ rr = d \cdot r\ \} \quad \text{in} \quad P,\quad . \tag{5}$$

The assignments to q and qq are

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

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

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

which is an instance of the *addition-relation* ***rule*** **<11>**

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

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

$\{ qq = d \cdot q \}$ **in** P, . **(6)**

In all, we have established the following global invariants:

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

2. Control and Heuristic 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>**

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

to the four assignments at the beginning of the program, we get the local invariant

$\{ (q , qq , r , rr) = (0 , 0 , 1 , d) \}$

just prior to the loop. The ***loop*** *axiom* **<20>,**

```
loop P'
     until t
     { ¬t }
     P''
     repeat
```

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

```
{(q , qq , r , rr) = (0 , 0 , 1 , d)}

loop   L1: ( ... }
       until r ≤ e
       { r > e }
       if qq+rr ≤ c then q := q+r; qq := qq+rr fi
       r := r/2; rr := rr/2
       repeat .
```

The conditional statement of the loop,

```
if qq+rr ≤ c then q := q+r; qq := qq+rr fi
```

may be considered as having an empty **else** branch, *i.e.*,

```
if qq+rr ≤ c then q := q+r; qq := qq+rr else fi .
```

So we apply the *forward test rule* **(25)**,

```
{ α }
if t then L';;P'
     else L'';;P''
     fi
-------------------------
{ a, t } at L'
{ a , ¬t } at L'' ,
```

obtaining, thereby,

```
if qq+rr ≤ c  then   { r > e , qq+rr ≤ c }; q := q+r; qq := qq+rr
              else   { r > e , c < qq+rr }
              fi .
```

Using the *forward assignment rule* **(21),**

$\{ \alpha(u, y) \}$
$x := u$
$L:$

$\{ \alpha(x, y) \}$ at L ,

where x **does not appear in** $\alpha(l, y)$ **, the assignments of the then branch transform the invariant** $qq+rr \le c$ **into** $qq \le c$ **and leave** $r > e$ **unchanged. We obtain**

```
if qq+rr ≤ c then q := q+r; qq := qq+rr; { r > e , qq ≤ c }
             else { r > e , c < qq+rr }
             fi .
```

We may now apply the *forward branch rule* **(27)**

```
if t  then  P' ; { a }
      else  P" ; { β }
      fi
L:
```

$\{ \alpha \vee \beta \}$ at L .

This rule disjoins the two possible outcomes of the conditional, and we obtain the invariant

$\{ (r > e \wedge qq \le c) \vee (r > e \wedge c < qq+rr) \}$.

The invariant simplifies to just

$\{ r > e \}$,

since $r > e$ **appears in both disjuncts while** $qq \le c \vee c < qq+rr$ **is a tautology (if the first disjunct is false, then** $qq > c$ **, and since** rr **is positive,** $qq+rr > c$ **is implied).**

However, the *disjunction heuristic*

```
if   t  then  P' ; { α }
      else      P'' ; { β }
      fi
L:
```

{? α , β ?} at *L*

suggests that each of the two Invariants, $qq \le c$ **and** $c < qq+rr$ **, may itself be an Invariant. So we have**

{ $r > e$ } **and** {? $qq \le c$, $c < qq+rr$?}

following the conditional and preceding the assignments

$r := r/2$; $rr := rr/2$.

By further application of the ***forward assignment rule*** *to* the one invariant and the two **candidates, we get**

{ $2 \cdot r > e$ } **and** {? $qq \le c$, $c < qq+2 \cdot rr$?}

at the end of the loop. So far we have the annotated loop:

```
{ (q , qq , r , rr) = (0 , 0 , 1 , d) }
loop   L,: { . . . )
       until r ≤ e
       if qq+rr ≤ c then q := q+r; qq := qq+rr fi
       r := r/2; rr := rr/2
       { 2•r > e } {? qq ≤ c , c < qq+2•rr ?}
       repeat .
```

Finally, by applying the *forward* ***loop-body*** *rule* **<29>,**

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

to the invariant at the end of the **loop** body, we derive the **loop** invariant

$$(\ (q, qq, r, rr) = (0, 0, J, d)\ \vee\ 2 \cdot r > e\ )\quad \textbf{at}\ L_1 .$$

In order to simplify the presentation slightly, we shall use instead the weaker

$$\{\ r = 1\ \vee\ 2 \cdot r > e\ \}\ \textbf{at}\ L, \quad . \tag{7}$$

By a similar application of the *forward loop-body rule* **to the two candidates at the end of the loop body,** we get the candidates

$$\{?\ (q, qq, r, rr) = (0, 0, 1, d) \vee qq \le c\ ?\}\quad \textbf{at}\ L_1$$

and

$$\{?(q, qq, r, rr) = (0, 0, 1, d)\ \vee\ c < qq + 2 \cdot rr\ ?\}\quad \textbf{at}\ L, .$$

Both candidates may be simplified, since their first disjunct is **subsumed by their second, leaving**

$$\{?\ qq \le c,\ c < qq + 2 \cdot rr\ ?\}\ \text{at}\ L_1 .$$

These two candidates can indeed be proved to be invariants: The first candidate, $qq \le c$, **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**

$$\{\ qq \le c,\ c < qq + 2 \cdot rr\ \}\quad \textbf{at}\ L, . \tag{8}$$

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, With the loop exit test $r \le e$, the output invariants include

$$\{\ rr = d \cdot r\ ,\ qq = d \cdot q\ ,\ \ (r=1 \vee 2 \cdot r > e)\ ,\ qq \le c\ ,\ \ c < qq + 2 \cdot rr\ ,\ 7 \le e\ \} \quad \text{at } E_1\ . \tag{9}$$

Note that we did not make any use of the candidates

$$\{?\ q \le c/d\ ,\ \ c/d \ \ < q+e\ ?\} \text{ at } E_1\ ,$$

suggested by the output specification, au no new invariants would be derived.

Though these invariants do imply $q \le c/d$ as specified, they do not imply $c/d < q+e$. In fact our program as given is incorrect. For a discussion of how these invariants may be used to guide the debugging of the program, see Dershowitz and Manna [1977].

3, Loop Counter8

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

The extended program (annotated with some of the invariants we have already found) is:

```
P1: begin comment real division
    B1: { 0 ≤ c < d , 0 < e }
    q := 0; qq := 0; r := 1; rr := d
    n := 0
    loop   L1: { rr = d·r , qq = d·q , (r=1 ∨ 2·r>e) , qq ≤ c , c < qq+2·rr }
           until r ≤ e
           if qq+rr ≤ c then q := q+r; qq := qq+rr fi
           77 := r/2; 77 := rr/2
           n := n+1
           repeat
    E1: { rr = d·r , qq = d·q , (r=1 ∨ 2·r>e) , qq ≤ c , c < qq+2·rr , 7 ≤ e }
    end .
```

Obviously,

$$\{\ n \in N\ \} \quad \text{in } P, \quad . \tag{10}$$

For the variables r and n, we have the assignments

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

and we can apply the *linear-relation rule* ⟨14⟩

$$\frac{(x\ ,\ y) := (a_0\ ,\ b_0) \mid (a_1 \cdot x + a_2\ ,\ y + b_2) \quad \text{in } P}{\{\ [x \cdot (a_1 - 1) + a_2]^{b_2} \cdot a_1^{b_0} = [a_0 \cdot (a_1 - 1) + a_2]^{b_2} \cdot a_1^{y}\ \} \text{ in } P\ .}$$

With this rule we get the global invariant

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

which simplifies to yield

$$\{\ r = 1/2^n\ \} \text{ in } P, \quad . \tag{11}$$

Applying the same rule to

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

we deduce

$$\{\ rr = d/2^n\ \} \text{ in } P, \quad . \tag{12}$$

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 output invariant (Q), $\neg\, \varepsilon \le e \wedge (r = 1 \vee 2 \cdot r > e)$, and get

$$1/2^n \le e \quad \wedge \quad (1/2^n = 1 \vee 2/2^n > e)\ .$$

Taking the logarithm (e is positive), we have the lower bound

$$-\log_2 e \le n$$

and upper bound

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

on the number of loop iterations n. Note that by finding an upper bound on the number of

iterations, we have actually proved that the loop terminates.

Combining both bounds 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 bound

$$n = \lceil -\log_2 e \rceil = -\lfloor \log_2 e \rfloor .$$

Thus we have the output invariant

$$\{ n = 0 \vee n = -\lfloor \log_2 e \rfloor \} \text{ at } E_1 . \qquad (13)$$

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. EXAMPLE: Selection-Sort Program

The previous example contained only one loop and dealt with simple variables. As a more challenging example, we annotate an array-manipulation program containing nested loops. The program is intended to sort the array $A[0:n]$ of $n+1$ elements $A[0], A[1], \ldots, A[n]$ in ascending sequence. The output specification can therefore be expressed as

$$(\forall l)(0 \le l < n)(A[l] \le A[l+1]) \quad A \quad perm(A[0:n], A_0[0:n])$$

where $perm(A[0:n], A_0[0:n])$ indicates that $A[0:n]$ is a permutation of the array $A_0[0:n]$, and A_0 is the value of the array A when the program is first entered. The program is:

```
begin comment selection sort
B2: { n ∈ N }
i := 0
loop    L1: { . . . }
        until i ≥ n
        P1: begin
            j := i+1; m := A[i]; k := i
            loop    L3: { . . . }
                    until j > n
                    if A[j] < m then m := A[j]; k := j fi
                    j := j+1
                    repeat
            A[k] := A[i]; A[i] := m; i := i+1
            end
        repeat
E1: {? (∀l)(0 ≤ l < n)(A[l] ≤ A[l+1]), perm(A[0:n], A0[0:n]) ?}
end .
```

1. Assignment Rules

We first try to determine the range of the program variables. The variables in the

program P_2 are i, j, k, m, and A; the inner loop (the program segment P_3) sets the variables j, k and m, and leaves i and A unchanged.

The assignments to i are

$$i := 0 \qquad i := i+1$$

which by the *addition rule* <1>

$$\frac{x := a_0 \mid x+a_1 \quad \text{in } P}{\{\, x \in a_0+a_1 \cdot N \,\} \quad \text{in } P}$$

give the global invariant

$$\{\, i \in N \,\} \quad \text{in } P_2 . \tag{1}$$

Since the program P_2 contains the labels L_2, L_3 and E_2, this relation holds at ail three points.

The assignments to j are

$$j := i+1 \qquad j := j+1 .$$

Since we know $i \in N$, we may substitute N for i to obtain the nondeterministic assignments

$$j :\in N+1 \qquad j :\in j+1 ,$$

and by the *Jet-addition rule* <5> we get $j \in N+1+\Sigma 1$, which simplifies to

$$\{\, j \in N, \quad 1 \le j \,\} \quad \text{in } P_2 . \tag{2}$$

(Recall that these global invariants only hold after $j := i+1$ is executed for the first time,) Since within P_3 the value of i is unchanged, it may be regarded as a constant. We can therefore apply the *addition rule* to the assignments to j, $j := i+1$ and $j := j+1$, obtaining

$$\{\, j \in i+1+N \,\} \text{ in } P ,$$

and consequently

$$\{\, i < j \,\} \qquad \text{in } P_3 . \tag{3}$$

The assignments to k are

$$k := i \qquad k := j.$$

Using (1) and (2) to substitute N for i and j, we have

$$k :\in N \qquad k :\in N$$

and from the simple *set-union rule* <4>

$$\frac{x :\in S_0 \mid S_1 \quad \text{in } P}{\{x \in S_0 \cup S_1\} \quad \text{in } P}$$

it follows that

$$\{ k \in N \} \quad \text{in } P_2 . \tag{4}$$

in P_2, as we have seen, i is constant and $j \in i+1+N$, so we substitute $i+1+N$ for j in the assignments to k to obtain

$$k :\in i \qquad k :\in i+1+N .$$

By the same *set-union rule, we* have that k belongs to the union of i and $i+1+N$. Therefore $k \in i+N$, and

$$\{ i \le k \} \quad \text{in } P_1 . \tag{5}$$

Finally, for m we have the assignments

$$m := A[i] \qquad m := A[j] .$$

Using (1) $i \in N$ and (2) $j \in N$ to substitute N for i and j, we get

$$m :\in A[N] \qquad m :\in A[N] .$$

Thus, by the *set-union rule, we* obtain

$$\{ m \in A[N] \} \text{ in } P_2 , \tag{6}$$

In the following subsections, we shall apply the control rules and heuristics first to the inner loop and then to the outer loop.

2, Control Rules - Inner Loop

At any point in a program, the disjunction of what is known from the paths leading to that point is an invariant. So we can obtain loop invariants at label L_3, by considering the three paths leading to L_3: the initialization path from L_2 to L_3, the loop-body path from L_3 to L_3 via the then branch of the conditional, and the loop-body path via the **else** branch of the conditional.

From the initialization path, we have upon entering the inner loop

$$i < n \wedge j = i+1 \quad A \quad m = A[i] \quad A \quad k = i \ . \tag{7}$$

The conjunct $i < n$ derives from the negation of the outer-loop exit test (using the *loop* axiom **<20>**); the other three conjuncts are obtained from the three assignments along the initialization path (by the *assignment axiom* **<18>**).

At the head of the inner-loop body, we have the invariant

$$j \leq n \quad A \quad i = i_{L_3} \quad A \quad A = A_{L_3} \quad A \quad j = j_{L_3} \quad A \quad k = k_{L_3} \quad A \quad m = m_{L_3} \ ,$$

where x_L, for some variable x and label L, denotes the value of x when control was last at L. The first conjunct Is the negation of the exit test and the other conjuncts, which are generated at L_3 using the *value axiom* **<33>**,

$$\{ \ x = x_L \ \} \quad \text{at} \quad L \ ,$$

have been pushed passed the exit test unchanged (this is an application of the forward *loop-exit* rule **<31>** to the inner loop). After executing the assignments in the **then** branch of the conditional, we know

$$j \leq n \quad A \quad m = A[j] \quad \text{A} \quad k = j \quad \text{A} \quad i = i_{L_3} \quad A \quad A = A_{L_3} \quad \wedge \quad j = j_{L_3} \ .$$

The second and third conjuncts derive from the assignments (by **<18>**); all the other conjuncts **have** been propagated **forward** (by the *forward test rule* **<25>** and forward *assignment rule* **<21>**).

After the (empty) **else** branch of the conditional, we have

$$j \le n \ \text{A} \ m \le A[j] \ \text{A} \ i = i_{L_3} \ \text{A} \ A = A_{L_3} \ \text{A} \ j = j_{L_3} \ \text{A} \ k = k_{L_3} \ \text{A} \ m = m_{L_3} .$$

The second conjunct is the negation the conditional test (by the *conditional axiom* **<19>). Since we must have traversed either the then or else branch, we know that after the conditional**

$$\begin{aligned} &(j \le n \ \text{A} \ m = A[j] \ \text{A} \ k = j \ \text{A} \ i = i_{L_3} \ \text{A} \ A = A_{L_3} \ \text{A} \ j = j_{L_3}) \\ \text{V} \ &(j \le n \ \text{A} \ m \le A[j] \ A \ i = i_{L_3} \ A \ A = A_{L_3} \\ &\qquad \text{A} \ j = j_{L_3} \ \text{A} \ k = k_{L_3} \ \text{A} \ m = m_{L_3}) \end{aligned}$$

(this is the *forward branch rule* **<27>). Thus, at the end of the loop body, after incrementing j by** *1* **, we have (by <21>)**

$$\begin{aligned} &(j\text{-}1 \le n \ \text{A} \ m = A[j\text{-}1] \ \text{A} \ k = j - 1 \ \text{A} \ i = i_{L_3} \ \text{A} \ A = A_{L_3} \ \text{A} \ j\text{-}1 = j_{L_3}) \\ \text{V} \ &(\text{j-l} \le n \ \text{A} \ m \le A[j\text{-}1] \ \text{A} \ i = i_{L_3} \ \text{A} \ A = A_{,_3} \\ &\qquad \text{A} \ j\text{-}1 = j_{L_3} \ \text{A} \ k = k_{L_3} \ \text{A} \ m = m_{L_3}) . \end{aligned} \tag{8}$$

Furthermore, if a relation α holds **upon entering** a **loop,** and we **know** that the **loop body either does** not change the values of the variables in α , or **reachieves** *a* for the **new values of the variables, then a is a loop invariant. This** *is* **the** *protected-invariant rule* **<34>**

$$\frac{\begin{array}{l} \{ \ \alpha(x) \ \} \\ \textbf{loop} \ L: \\ \qquad P \\ \qquad \{ \ \alpha(x) \ \text{V} \ x = x_L \ \} \\ \qquad \textbf{repeat} \end{array}}{\{ \ \alpha(x) \ \} \ \text{at} \ L \ .}$$

By substituting k **for j-1 in the first disjunct of (8), we may derive** $k \le n$ **and** $m = A[k]$**. Thus, at the end of the loop body we know** $(k \le n \ \text{A} \ m = A[k]) \ \text{V}$ $(\text{A} = \text{A},_3 \ \text{A} \ \text{k} = \text{k}_{L_3} \ \text{A} \ \text{m} = \text{m}_{L_3})$**. This invariant is of the form** $\text{a}(\text{x}) \ \text{V} \ x = x_L$ **, taking** $\text{a}(\text{x})$ **to b e** $k \le n \wedge m = A[k]$ **and** x **to** be **the variables** A **,** k **and** m **. The first disjunct indicates that the then path achieves** $\alpha(x)$ **; the second disjunct states that the else path leaves** A **,** k **and** m **unchanged, From invariant (7) preceding the loop, we can derive that initially** $k \le n$ **and** $m = A[k]$ **.** *So we* **have**

$$\{\ k \le n,\ m = A[k]\ \} \text{ at } L_3\ . \tag{9}$$

Similarly, by (8) we have $i = i_{L_3}$ for both loop-body paths, and by (7) we have $i < n$ upon entering the loop. Taklng $a(f)$ to be $i < n$, we get

$$\{\ i < n\ \} \text{ at } L_3\ . \tag{10}$$

Disjoining invariant (7) of the initialization path and (8) from the loop-body path, we get the following inner-loop invariant (by the *forward loop-body rule* <29>):

$$\begin{aligned}\{\ &(\ i < n \text{ A } j = i+1 \text{ A } m = A[i] \wedge k = i\ )\\ &\text{V } (\ j\text{-}1 \le n \text{ A } m = A[j-1] \text{ A } k = j-1\ )\\ &\text{V } (\ j\text{-}J \le n \text{ A } m \le A[j-1]\ )\qquad\qquad \}\ \text{at } L_3\ .\end{aligned} \tag{11}$$

(The conjuncts refering to the previous value of a variable at L_3 have been removed.)

Now we extract the "common denominator" of the disjuncts in (11) arising from the different paths. The relation $j\text{-}J \le n$ appears in the second two disjuncts and is implied by the two conjuncts $i < n$ and $j = i+1$ of the first disjunct, so we get the invariant

$$\{\ j\text{-}l \le n\ \} \text{ at } L_3\ . \tag{12}$$

in the first disjunct of (11) we have $j = i+1$ A $m = A[i]$, in the second we have $m = A[j-1]$, while in the third we have $m \le A[j-1]$, thus for ail paths

$$\{\ m \le A[j-1]\ \} \text{ at } L,\ . \tag{13}$$

3. Generalization Heuristic - Inner Loop

The following *generalization heuristic* <37> is particularly valuable for loops involving arrays:

$\{\ x = a\ \}$
loop L: $\{\ \alpha(x, y)\ \}$
P
$\{\ x = x_L + 1\ \}$
repeat

$\{?\ (\forall l)(a \le l \le x)\alpha(l, y)\ ?\}$ at L .

This heuristic is similar to the *forall rule* (35), but only suggests a candidate, since the variable y may change value in P. in our case, reconsider the inner-loop invariant (13) $\alpha(j, m)$: $m \le A[j-1]$ at L_3. initially j is $i+1$, and at the end of the loop body j = $j_{L_3}+1$, so, as an invariant candidate, we try

$\{?\ (\forall l)(i+1 \le l \le j)(m \le A[l-1])\ ?\}$ at L_3 ,

which we shall abbreviate as $m \le A[i:j-1]$. Checking the candidate for the then and else paths, determines that it is in fact an invariant, and we have for the inner loop

$\{\ m \le A[i:j-1]\ \}$ at L_3 . (14)

- So far we have derived the following inner-loop invariant8

$\{\ k \le n,\ m = A[k],\ i < n,\ j - 1 \le n,\ m \le A[i:j-1]\ \}$ at L_3 .

We turn now to consider the outer loop.

4. Control Rules - Outer Loop

Using the *forward loop-exit rule* (31), the invariants at L_3 may be propagated past the exit test $j > n$, obtaining

$\{\ k \le n,\ m = A[k],\ i < n,\ j-1 \le n,\ m \le A[i:j-1],\ j > n\ \}$

just prior to the assignments

$A[k] := A[i]$; $A[i] := m$; i := $i+1$.

Propagating these invariants past the assignments, we get the following invariants at the end of the outer-loop body:

$$\{\ k \le n\ ,\ i \le n\ ,\ m \le A[i:j-1]\ ,\ m = A[i-1]\ ,\ j-1 = n\ \}\ . \qquad (15)$$

The invariant $k \le n$ is propagated unchanged. The invariant $i < n$ becomes $i-1 < n$ after executing $i := i+1$ (by the *forward assignment rule* ⟨21⟩), which is equivalent to $i \le n$ (since both i and n are integers). The invariant $m \le A[i:j-1]$ still holds after assigning to $A[k]$, since it also held for $A[i]$; after the assignment to $A[i]$, it becomes $m \le A[i+1:j-1]$ (by the *forward array-assignment rule* ⟨23⟩); after incrementing i, it becomes $m \le A[i:j-1]$. The assignment $A[i] := m$ generates the invariant $m = A[i]$ (by the *assignment axiom* ⟨18⟩), which becomes $m = A[i-1]$ after incrementing i. Finally, the invariants $j-1 \le n$ and $j > n$ simplify to $j-1 = n$ (since (2) $j \in N$).

Clearly upon entering the outer loop (by ⟨18⟩)

$$i = 0\ .$$

Thus, by the *forward loop-body rule* ⟨29⟩, we have the outer-loop invariant

$$\{\ i = 0 \ \vee\ (k \le n \wedge i \le n \wedge m \le A[i:j-1] \wedge m = A[i-1] \wedge j-1 = n)\ \}\quad \text{at } L_2$$

with the following two corollaries:

$$\{\ i = 0 \vee A[i-1] \le A[i:n]\ \}\quad \text{at } L_2 \qquad (16)$$

(the second disjunct follows from $m \le A[i:j-1]$, $m = A[i-1]$ and $j-1 = n$), and

$$\{\ i \le n\ \}\quad \text{at } L_2 \qquad (17)$$

(since $i = 0$ is subsumed by $i \le n$ for $n \in N$). If we use the *forward loop-exit rule* ⟨31⟩ to push $i \le n$ past the exit test $i \ge n$ and out of the loop, we get the output invariant $i \le n \wedge i \ge n$ at E_2, or,

$$\{\ i = n\ \}\quad \text{at } E_2\ . \qquad (18)$$

5. Heuristics - Outer Loop

We use the *generalization heuristic* ⟨37⟩ to generalize (16) for the counter i, where $\alpha(i, A)$ is $i = 0 \vee A[i-1] \le A[i:n]$. Since i is initially 0, this yields the candidate

$$\{?\ (\forall l)(0 \le l \le i)(l = 0 \vee A[l-1] \le A[l{:}n])\ ?\} \quad \text{at } L_2 .$$

This is equivalent to

$$\{?\ (\forall l)(0 \le l < i)(A[l] \le A[l+1{:}n])\ ?\} \quad \text{at } L_2$$

and states, in effect, that the array elements $A[0{:}i-1]$ are sorted and that they are all smaller than the array elements $A[i{:}n]$. It can be shown that it does indeed remain invariant, so we have the outer-loop invariant

$$(\ (\forall l)(0 \le l < i)(A[l] \le A[l+1{:}n])\ ) \quad \text{at } L_2 . \tag{19}$$

This may be pushed out of the loop to E_2, and with (18), *i.e.*, $i = n$ at E_2, implies the first conjunct of the output specification,

$$(\forall l)(0 \le l < n)(A[l] \le A[l+1]) .$$

The *top-down heuristic* ⟨38⟩ suggests that the output specification $perm(A[0{:}n], A_0[0{:}n])$, which is obviously true initially, is itself a candidate at L_2. Since it can be shown that the only two assignments to A have the effect of exchanging the values of $A[k]$ and $A[i]$, we have the invariant

$$(\ perm(A[0{:}n], A_0[0{:}n])\ \} \text{ at } L_2 . \tag{20}$$

The program, annotated with some of the more important loop and output assertions, is:

P_3: **begin comment** *selection* sort
B_2: { $n \in N$ }
$i := 0$
loop L_2: { $i \in N$, $i \le n$, $(\forall l)(0 \le l < i)(A[l] \le A[l{:}n])$, $perm(A[0{:}n], A_0[0{:}n])$ }
until $i \ge n$
P_4: **begin**
$j := i+1$; $m := A[i]$; $k := i$
loop L_3: { $i, j, k \in N$, $i < n$, $i < j \le n+1$, $i \le k \le n$, $m = A[k]$, $m \le A[i{:}j-1]$ }
until $j > n$
if $A[j] < m$ **then** $m := A[j]$; $k := j$ **fi**
$j := j+1$
repeat
$A[k] := A[i]$; $A[i] := m$; $i := i+1$
end
repeat
E_2: { $i = n$, $(\forall l)(0 \le l < i)(A[l] \le A[l+1{:}n])$, $perm(A[0{:}n], A_0[0{:}n])$ }
end .

To determine the time complexity of this program, we add three counters: one for the outer loop, one for the inner loop, and a third to sum the total number of inner-loop executions. Using **the annotation** rules, one can **easily** show that the outer loop **is iterated** n **times, that the inner loop is executed** n-i **times for each outer-loop iteration, and that the total number of inner-loop executions is** $n \cdot (n+1)/2$ **.**

ACKNOWLEDGEMENT

We thank David Harel, Shmuei Katz, Jim King, Larry Paulson, Wolf Poiak and Richard Waldinger for their critical readings of the manuscript.

REFERENCES

Caplain, M. [Apr. 1975], *Finding invariant assertions for proving programs,* Proc. inti. Conf. on Reliable Software, Los Angeles, CA, pp. 166-l 71.

Dershowitz, N. and Z. Manna *[Nov.* 1977], *The evolution of programs: Automatic program modification,* IEEE Software Engineering, vol. SE-3, no. 6.

Elspas, B. [July 1974], *The semiautomatic generation of inductive assertions for proving program correctness,* interim report, SRI international, Menlo Park, CA.

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

German, S.M., and B. Wegbreit [Mar, 1975], *A synthesizer of inductive assertions,* IEEE Software Engineering, vol. SE-l, no. 1, pp. 68-76.

Gibb, A. [July 196 1), *Algorithm 61: Procedures for range arithmetic,* CACM, vol. 4, no. 7, pp. 319-320.

Greif, I. and R.J. Waldinger [Apr. 1974], *A more mechanical heuristic approach to program verification,* Proc. inti. Symp. on Programming, Paris, France, pp. 83-90.

Harrison, W.H. *[May* 1977], *Compiler analysis of the value ranges for variables,* IEEE Software Engineering, vol. SE-3, no. 3, pp. 243-250.

Hoare, C.A.R. [Oct. 1969], *An axiomatic basis of computer programming,* CACM, vol. 12, no, 10, pp. 576-580, 683.

Katz, S.M. [Sept. 1976], *Invariants and the logical analysis of programs,* Ph.D. thesis, Weizmann institute of Science, Rehovot, Israel.

Katz, S.M. and Z. Manna [Aug. 1973], *A heuristic approach to program verification,* Adv, Papers Third inti. Conf. on Artificial intelligence, Stanford, CA, pp. 600-512.

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

Misra, J. [July 1975], *Relations uniformly conserved by a loop,* Coiloques IRIA on Proving and improving Programs, Arc-et-Senans, France, pp. 71-80.

Moriconi, M.S. [Oct. 19743, *Towards the interactive synthesis of assertions,* Memo ATP-20, Automatic Theorem Proving Project, Univ. of Texas, Austin, TX.

Morris, J.H. and B. Wegbreit [Apr. 1977], *Subgoal induction,* **CACM, vol. 20, no. 4, pp. 209-222.**

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

Sintzoff, *M.* **[Jan. 1972],** *Calculating properties of programs by valuations on specific models,* **Proc. Conf. on Proving Assertions About Programs, Las Cruces, NM, SIGPLAN Notices,** *vol. 7,* **no. 1,** pp. 203-207.

Suzuki N. and K. Ishihata [Jan, 1977], *Implementation of an array bound checker,* **Proc. Fourth Symp. on** Principles of **Programming** Languages, Los Angeles, CA, pp. 132-l **43.**

Tamir, M. [Aug. 1976], *ADI - Automatic derivation of invariants,* **Master's thesis, Weizmann Institute of Science,** Rehovot, Israel,

Teitelman, W, [1974], *INTERLISP reference manual,* **Xerox Research Center, Palo Alto, CA.**

Wegbreit, B. [Feb. 1974], *The synthesis of loop predicates,* **CACM, vol. 17, no. 2, pp. 102-l 12.**

Wegbreit, B. [Sept. 1975], *Property extraction in well-founded property sets,* **IEEE Software Engineering, vol.** SE-l, **no. 3,** pp. **270-285.**

Wilber, B.M. [Mar. 1976], *A QLISP reference manual,* **Tech. note 118, Artificial Intelligence Center, SRI International, Menlo Park, CA.**

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, γ and δ denote predicates;
x, y and z are variables;
a, a_i and b_i are expressions **which are constant in the given program segment;**
u and v are arbitrary expressions;
N denotes the set of natural numbers and I the set of all integers.

1. Assignment Rules

● Range rules

<1> *addition rule*

$$\frac{x := a, \ | x+a_1 | x+a_2 | \ . \ . \ . \quad \text{in } P}{\{ x \in a_0+a_1 \cdot N+a_2 \cdot N+ \ . \ . \ . \} \quad \text{in P}}$$

<2> *multiplication rule*

$$\frac{x := a, \ | x \cdot a_1 | x \cdot a_2 | \ . \ . \ . \quad In \ P}{\{ x \in a_0 \cdot a_1^N \cdot a_2^N \cdot . \ . , \ ) \quad \textbf{in } P}$$

<3> *exponentiation rule*

$$\frac{a := a, \ | a | a | \ . \ . \ . \quad \text{in } P}{[illegible]}$$

- **Set assignment rules**

$x :\in S$ refers to an assignment $x := u$ where it is known 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*

$$\frac{x :\in S_0 \mid S_1 \mid S_2 \mid \ldots \quad \text{in } P}{\{ x \in S_0 \cup S_1 \cup S_2 \cup \ldots \} \quad \text{in } P}$$

⟨5⟩ *set-addition rule*

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

⟨6⟩ *set-multiplication rule*

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

⟨7⟩ *set-exponentiation rule*

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

- **Counter relation rules**

n is an integer variable;
n_0 is an integer;
$v(n)$ is an expression containing the one variable n.

⟨8⟩ *addition-counter rule*

$$\frac{(x, n) := (a_0, n_0) \mid (x+v(n), n+1) \quad \text{in } P}{\{ x = a_0+\Sigma_{l=n_0}^{n-1} v(l) \} \text{ in } P}$$

⟨9⟩ *multiplication-counter rule*

$$\frac{(x, n) := (a_0, n_0) \mid (x \cdot v(n), n+1) \quad \text{in } P}{\{ x = a_0 \cdot \Pi_{l=n_0}^{n-1} v(l) \} \text{ in } P}$$

<10> *exponentiation-counter rule*

$$\frac{(x\ ,\ n) := \ (a_0\ ,\ n_0) \mid (x^{v(n)}\ ,\ \ n+1) \text{ in } P}{\{\ x = a_0 \prod_{l=n_0}^{v-1} v(l)\ \} \text{ in } P}$$

- **Basic relation rules**

<11> *addition-relation rule*

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

<12> *multiplication-relation rule*

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

<13> *exponentiation-relation rule*

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

- **Assorted relation rules**

<14> *linear-relation rule*

$$\frac{(x\ ,\ y) := \ (a,\ ,\ b_0) \mid (a_1 \cdot x+a_2\ ,\ b_1 \cdot y+b_2) \quad \textit{in } P}{\begin{array}{ll} \{\ [x \cdot (a_1-1)+a_2]^{b_2} \cdot a_1^{b_0} = [a_0 \cdot (a_1-1)+a_2]^{b_2} \cdot a_1^{y}\ \} \text{ in } P & \textit{when } b_1 = I \\ \{\ [x \cdot (a_1-1)+a_2]^{log(b_1)} \cdot [b_0 \cdot (b_1-1)+b_2]^{log(a_1)} = & \\ \quad [a_0 \cdot (a_1-1)+a_2]^{log(b_1)} \cdot [y \cdot (b_1-1)+b_2]^{log(a_1)}\ \} \text{ in } P & \textit{otherwise} \end{array}}$$

<15> *quadratic rule*

$$\frac{(x\ ,\ y) := \ (a_0\ ,\ b_0) \mid (x+a_2\ ,\ y+b_1 \cdot x+b_2) \quad \text{in } P}{\{\ (y-b_0) \cdot 2 \cdot a_2^{\ 2}\ =\ (x-a_0) \cdot [b_1 \cdot (x-a_0-a_2)+2 \cdot a_2 \cdot b_2]\ \} \text{ i n } P}$$

<16> *factorial rule*

$$\frac{(x, y) := (a_0 \cdot a_1, b_0) \mid (x + a_1, y \cdot x \cdot b_1) \quad \textbf{in } P \qquad a_1, a_1 \in iv}{\{\, y \cdot a_0! = b_0 \cdot (a_1 \cdot b_1)^{x/a_1 - a_0} \cdot (x/a_1)! \,\} \; in \; P}$$

<17> *multiplication-exponentiation rule*

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

2. *Control* Rules

- **Control axioms**

<18> *assignment axiom*

$x := a$
$\{\, x = a \,\}$

<19> *conditional axiom*

if t **then** $\{\, t \,\}; P'$
else $\{\, \neg t \,\}; P''$
fi

<20> *loop axiom*

loop P'
until t
$\{\, \neg t \,\}$
P''
repeat
$\{\, t \,\}$

- **Assignment control rules**

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

<21> *forward assignment rule*

$$\frac{\{\ \alpha(x, y)\ \}\quad x := f(x, y)\quad L:}{\{\ \alpha(f^{-}(x, y), y)\ \}\ \text{at}\ L} \qquad \frac{\{?\ \gamma(x, y)\ ?\}\quad x := f(x, y)\quad L:}{\{?\ \gamma(f^{-}(x, y), y)\ ?\}\ \text{at}\ L}$$

where f^{-} is the inverse of the function f in the first argument, *i.e.*, $f^{-}(f(x, y), y) = x$.

$$\frac{\{\ \alpha(u, y)\ \}\quad x := u\quad L:}{\{\ \alpha(x, y)\ \}\ \text{at}\ L} \qquad \frac{\{?\ \gamma(u, y)\ ?\}\quad x := u\quad L:}{\{?\ \gamma(x, y)\ ?\}\ \text{at}\ L}$$

where x does not appear In $\alpha(l, y)$ or $\gamma(l, y)$.

<22> *backward assignment rule*

$$\frac{L:\quad x := u\quad \{\ \beta(x, y)\ \}}{\{\ \beta(u, y)\ \}\ \text{at}\ L} \qquad \frac{L:\quad x := u\quad \{?\ \delta(x, y)\ ?\}}{\{?\ \delta(u, y)\ ?\}\ \text{at}\ L}$$

<23> *forward array-assignment rule*

$$\frac{\{\ \alpha(A, z)\ \}\quad A[y] := f(A[y], z)\quad L:}{\{\ \alpha(assign(A, y, f^{-}(A[y], z)), z)\ \}\ \text{at}\ L} \qquad \frac{\{?\ \gamma(A, z)\ ?\}\quad A[y] := f(A[y], z)\quad L:}{\{?\ \gamma(assign(A, y, f(A[y], z)), z)\ ?\}\ \text{at}\ L}$$

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

<24> *backward array-assignment rule*

$$\frac{L:\quad A[y] := v\quad \{\ \beta(A, z)\ \}}{\{\ \beta(assign(A, y, v), z)\ \}\ \text{at}\ L} \qquad \frac{L:\quad A[y] := v\quad \{?\ \delta(A, z)\ ?\}}{\{?\ \delta(assign(A, y, v), z)\ ?\}\ \text{at}\ L}$$

- **Conditional control rules**

<25> *forward test rule*

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

$$\frac{\begin{array}{l}\{?\ \gamma\ ?\}\\ \textbf{if}\ t\ \textbf{then}\ L'\text{:};\ P'\\ \quad \textbf{else}\ \ L''\text{:};\ P''\\ \quad \textbf{fi}\end{array}}{\{?\ \gamma\ ?\}\ \text{at}\ L'\ \text{and}\ L''}$$

<26> *backward test rule*

$$\frac{\begin{array}{l}L\text{:}\\ \textbf{if}\ t\ \textbf{then}\ \{\ a\ \};\ P'\\ \quad \textit{else}\ \ \{\ \beta\ \};\ P''\\ \quad \textbf{fi}\end{array}}{\{\ t \supset \alpha\ ,\ \neg t \supset \beta\ \}\ \text{at}\ L}$$

$$\frac{\begin{array}{l}L\text{:}\\ \textbf{if}\ t\ \textbf{then}\ \{?\ \gamma\ ?\};\ P'\\ \quad \textbf{else}\ \ \{?\ \delta\ ?\};\ P''\\ \quad \textbf{fi}\end{array}}{\{?\ t\ 3\ \gamma\ ,\ \neg t\ 3\ \delta\ ?\}\ \text{at}\ L}$$

<27> *forward branch rule*

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

$$\frac{\begin{array}{l}\textbf{if}\ t\ \textbf{then}\ P';\{?\ \gamma\ ?\}\\ \quad \textbf{else}\ \ P'';\{?\ \delta\ ?\}\\ \quad \textbf{fi}\\ L\text{:}\end{array}}{\{?\ \gamma\ \vee\ \delta\ ?\}\ \text{at}\ L}$$

<28> *backward branch rule*

$$\frac{\begin{array}{l}\textbf{if}\ t\ \textbf{then}\ P';\ L'\text{:}\\ \quad \textbf{else}\ \ P'';\ L''\text{:}\\ \quad \textbf{fi}\\ \{\ \beta\ \}\end{array}}{\{\ \beta\ \}\ \text{at}\ L'\ \text{and}\ L''}$$

$$\frac{\begin{array}{l}\textbf{if}\ t\ \textbf{then}\ P';\ L'\text{:}\\ \quad \textbf{else}\ \ P'';\ L''\text{:}\\ \quad \textbf{fi}\\ \{?\ \delta\ ?\}\end{array}}{\{?\ \delta\ ?\}\ \text{at}\ L'\ \text{and}\ L''}$$

- **Loop control rules**

<29> *forward loop-body rule*

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

$$\frac{\begin{array}{l}\{?\ \gamma\ ?\}\\ \textbf{loop}\ L\text{:}\\ \quad P\\ \quad \{?\ \delta\ \ 3)\\ \quad \textbf{repeat}\end{array}}{\{?\ \gamma \vee \delta\ ?\}\ \text{at}\ L}$$

<30> *backward loop-body rule*

L':
loop { β }
P
L'':
repeat

{ β } **at** L' **and** L''

L':
loop {? δ 3)
P
L'':
repeat

{? δ ?} **at** L' **and** L''

<31> *forward loop-exit rule*

loop P'
{ α }
until t
L':
P''
repeat
L'':

{ α , $\neg t$ } **at** L'
{ a , t } **at** L''

loop P'
{? γ ?}
until t
L':
P''
repeat
L'':

{? γ ?} **at** L' **and** L''

<32> *backward loop-exit rule*

loop P'
L:
until t
{ α }
P''
repeat
{ β }

{ $\neg t \supset \alpha$, $t \supset \beta$ } **at** L

loop P'
L:
until t
{? γ ?}
P''
repeat
{? δ ?}

{? $\neg t \supset \gamma$, $t \supset \delta$?} **at** L

● Value rules

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

<33> *value axiom*

$\neg$ { $x = x_L$) **at** L

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

(34) *protected-invariant rule*

$\{\ \alpha(x)\ \}$
loop *L:*
P
$\{\ a(x)\ \vee\ x = x_L\ \}$
repeat

$\{\ \alpha(x)\ \}$ **at** *L*

where x is the only variable in α,

(35) *forall rule*

$\{\ x = a,\ x \in I\ \}$
loop *L:* $\{\ a(x)\ \}$
P
$\{\ x = x_L + 1\ \}$
repeat.

$\{\ (\forall l \in I)(a \le l \le x)\alpha(l)\ \}$ at *L*

where x is the only variable in α.

3. Heuristic Rules

(36) disjunction heuristic

if *t* **then** P' ; $\{\ a\ \}$
else P'' ; $\{\ \beta\ \}$
fi
L:

$\{?\ \alpha,\ \beta\ ?\}$ **at** *L*

(37) *generalization heuristic*

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

$\{?\ (\forall l \in I)(a \le l \le x)\alpha(l, y)\ ?\}$ at *L*

⟨38⟩ *top-down* ***heuristic***

$\{\ \gamma\ \}$
loop
L:
until *t*
P''
repeat
$\{?\ \gamma\ ?\}$

$\{?\ \gamma\ 3)$ **at** *L*

- **Dangerous heuristics - To be applied with caution**

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

$\{\ \alpha \vee \beta\ \}$ **at** *L*

$\{?\ a,\ \beta\ ?)$ **at** *L*

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

$\{$ a(x) $)$ **and** $\{?\ \gamma(x)\ 3)$ **at** *L*

$\{?\ (\forall x)(\alpha(x) \supset \gamma(x))\ ?\ )$ **a t** *L*

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

$\{\ uRv\ )$ **and** $\{?\ uRw\ ?)$ **at** *L*

$\{?\ vRw$ V v = w ?$\}$ **at** *L*

where *R* **is a transitive relation.**