LANGUAGE FEATURES THAT SUPPORT PROGRAM VERIFICATION (illustrated in PL/C)

Robert L. Constable

TR 76-276

April 1976

Department of Computer Science
Cornell University
Ithaca, New York 14853

Language Features that Support Program Verification (illustrated in PL/C)

Robert L. Constable
Cornell University

Abstract:

The author of a program can convince it's readers (including himself) that it computes as he intended by writing into the program text a precise description of what it should do. This description can include a clear proof that the program behaves as advertised. In the future, programming languages may offer far more mechanical assistance in expressing, checking and processing these descriptions than they do now. However, some existing languages, such as PL/C, provide features which help in these tasks considerably. Here we examine such features applied to simple programs of the type traditionally taught in beginning programming classes. We discuss various proposals for expanding programming languages to provide more assistance of this type.

Keywords and phrases:

program correctness, automatic program verification, Hoare axioms, Scott induction, recursive programs, least number operator, bounded quantifiers, inductive assertion method, programming language design, very high level programming languages, PL/I, PL/C, macro facilities

§1 Introduction

(1.1) Some interesting computer programs never have been executed by any real computer and may never be executed.† They are interesting because people learn from them. These programs serve the same purpose as a mathematical article or scientific report.

There are, on the other hand, many executing programs which were written exclusively for the computer and a small group of its close associates. When people outside this group need to understand such programs, they are in trouble. These programs do not serve to illusidate and educate, but instead become objects of mystery and misplaced awe.

Experience has taught the computing community that all important programs should be written with both the machine and the human reader in mind. In the past, the executing machine has been more demanding than the understanding reader, requiring unfailing precision. Since human understanding does not require precise exactness, program segments written purely to aid understanding have usually appeared as natural language comments.

(1.2) Suppose we do require *precise* understanding of a program, how is this accomplished? Since programs are exact, we ask more generally, how do humans understand exact relationships? People understand informal relationships by reasoning about them. They understand exact relationships by *exact reasoning*;

---

and exact reasoning is mathematical argument. Indeed, logic and mathematics have evolved as subsets of natural language devoted to exact reasoning.

If we are interested in effectively communicating exact understanding of a program, then we need a way to include exact reasoning in the program text. That is, we need facilities for expressing logical relationships and proofs in the program text. Moreover, just as high level programming languages provide their users with automatic assistance in specifying precise <u>actions</u> (by allowing high level mathematical expressions), so too should future high level languages provide mechanical assistance in expressing <u>assertions</u> and <u>proofs</u>, perhaps to the point of checking the correctness of the proof.

(1.3) In this note we illustrate various ways of encorporating exact reasoning into PL/C program text and speculate on ways that future programming languages could make this encorporation easier and even assist in verifying the correctness of the reasoning.

The presentation is organized into sections as follows:

§2 Comparing proofs and programs

Programs for the Euclidean algorithm have become popular as specimens for logical analysis; probably because influential authors [5,8,9] have featured them in written exhibitions of proving the correctness of programs. Let me maintain this tradition.

(2.1) Consider the following algorithm which I claim on inputs $a \geq 0$, $b > 0$ finds q and r such that $a = b \cdot q + r$ & $0 \leq r < b$.

```
integer (a,b)
read (a,b)
if (b ≤ 0 V a < 0) then (print(input illegal); stop) fi
q:=0
while (a-b*(q+1) ≥ 0) do q:=q+1 od
r:= a-b*q
print (a,b,q,r)
stop
```

The logically interesting part is represented by

```
integers a ≥ 0, b > 0
q:=0
while (a-b*(q+1) ≥ 0) do q:=q+1 od
r:= a-b*q
```

In order to prove that this algorithm is "correct" in the usual manner (say following the writings of Floyd [6], Hoare [8]), we first must know our correctness criterion, namely

$$a = b*q + r \ \& \ 0 \leq r < b$$

and we search for assertions about a,b,q,r at various points in the algorithm which will help us conclude the correctness criterion at the exits of the algorithm. For instance, we can attach the following assertions at lines (1,2,4,5,6,7,9,10) (distinguished from actions because they are boolean expressions standing alone)†:

(2.2)

1. $a \geq 0$ & $b > 0$ & $q = 0$
2. $a - b*q \geq 0$
3. while $(a-b*(q+1) \geq 0)$ do $q:=q+1$ od
4. $a - b*q \geq 0$ & $a - b*(q+1) < 0$.
5. $b*q \leq a < b*(q+1)$
6. $0 \leq a-b*q < b*q+b - b*q = b$
7. $a = b*q + a - b*q$
8. $r:= a-b*q$
9. $0 \leq r < b$
10. $a = b*q + r$

The position of these assertions in the sequence of actions, is, of course, critical because the program statements may change the values of the variables. Thus to know that a program with assertions constitutes a proof of the exit assertions, we must be able to talk about the affect of an action on an assertion as well as talk about the more usual logical relationships between assertions.

---

†Notice that such a simple syntactic characterization of assertions fails if the assignment operation is = not := .

In this particular example, this means we must not only justify such formulas as a = b*q + r (line 10) from lines 7 and 8 (by "substituting equals for equals"), but we must justify asserting lines 4, 5 and 6 whose validity depends on knowing that the while loop did not change the relationship $a-b*q \geq 0$. That is, this relationship is left invariant by the loop.

(2.3) One way to prove such an invariance is to use mathematical induction on the number of times the body of the while loop is executed. Hoare [8,9,10] has conveniently expressed the induction axiom when it appeare in this form. He says, given a predicate $p(x_1,\ldots,x_n)$, such as $a - b*q \geq 0$, and given a while loop

while (b) do S od

where S is the loop body, here S is q:= q+1, one can show that $p(x_1,\ldots,x_n)$ is "invariant over the while statement" (i.e. holds after any number n executions of the loop body) by showing that it holds for 0 executions, i.e. before the loop is executed; and by showing that if b & $p(x_1,\ldots,x_n)$ holds before S is executed (say after n executions) then $p(x_1,\ldots,x_n)$ holds after S is executed (after n+1 executions). We then know that if the loop terminates, then

$$p(x_1,\ldots,x_n) \mathbin{\&} \neg b$$

will be valid immediately after the loop.

Using this technique, to show that line 4 is valid, we must know that $a - b*q \geq 0$ before the loop is executed (this is line 2), and that if

$$a - b*(q+1) \geq 0 \quad \text{(this is the condition b)}$$

and $$a - b*q \geq 0$$

are true before executing q:= q+1 , then

$$a - b*q \geq 0$$

is true after executing q: = q+1. This is obvious (and is an immediate consequence of Hoare's axiom for assignment). Thus we conclude line 4.

In order to know that line 5 is true we need only mathematical reasoning from line 4, e.g. add b*q to both sides of $0 \leq a - b*q$, etc. To know line 6 we do more symbolic mathematics, and to know line 7 we invoke a theorem of arithmetic (valid for a,b,q,r integers, so we need only know that no preceeding action has caused them to become undefined).

After the action in line 8, we obtain the assertion in 9 by substituting for r in line 6. Finally line 10 comes by substituting for r in line 7.

Now we have seen how to add commentary to the annotated program to prove the final assertion of that program. We can say that this has proved the program correct.

(2.4) Let us now see how the algebraist deals with the Euclidean algorithm. For instance, in the classic algebra textbook Survey of Modern Algebra by Birkhoff and MacLane in the section entitled, "7. The Euclidean Algorithm" the authors state the theorem:

Division Algorithm: For given integers a and b, with $b > 0$, there exist integers q and r such that

$$a = b*q + r \ , \ 0 \le r < b.$$

Their proof is essentially this.

Proof:

1. We know that $b > 0$.
2. There is some multiple of b which exceeds a, (e.g. $a - b*q < 0$ for some q).
3. Choose the least non-negative integer q such that $a - b*(q+1) < 0$.
4. By construction, $a - b*q \ge 0$ & $a - b*(q+1) < 0$.
5. So $b*q \le a < b*(q+1)$ and
6. $0 \le a - b*q < b*q+b - b*q = b$
7. Also for all a,b,q, $a = b*q + a - b*q$
8. Let $r := a - b*q$, then
9. $0 \le r < b$ by substitution in line 5. Then
10. $a = b*q + r$ by substitution in line 7.

Q.E.D.

It is interesting to compare this proof and the program line by line. Also notice that in the program we saw the use of induction disguised as an axiom about while loops. In this proof we see induction in the form of the least number principle, line 3.

It is noteworthy that this "purely algebraic proof" contains certain actions, such as choose (line 3) and let (line 8). These can be viewed as statements which convert the proof into a program. Such statements are common in ordinary mathematics but are usually translated out of formal mathematical proofs.

§3 Expressing proof oriented language statements in programs

(3.1) If we used a programming language that implemented the construct, "find the least y such that b" or "find the least y less than x such that b," then we could consider the algebra proof as a program (an algorithm plus assertions). The algorithmic part of the text would be:

```
find the least q such that a - b*(q+1) < 0
let r:= a - b*q
print (a,b,q,r).
```

The purely algorithmic part is not very informative text; it is improved considerably by adding the assertions.

(3.2) One might at this point argue that the program arising from the algebra proof is so inefficient that it is uninteresting (as is our first program). A more realistic program would be

```
integer (i,x,r,j,f)
read (i,x)
if ¬(x≥0 & i>0) then stop
    x≥0 & i>0
r:=x; j:=i; f:=0
do while (j≤r); j = 2*j, od
do while (j≠i);
    begin;
    j:= j/2
    f:= 2*f
    if j ≤ r then (r:= r-j; f:= f+1)
    end
od
```

print (i,x,r,f)

$x = f*i+r \ \& \ 0 \leq r < i \ \& \ j = i$

The run time of this program, based on the usual long division technique, is on the order O(log(x/i)) steps rather than O(x/i) steps for the earlier programs. However, a reasonably fast algorithm can also be found on the basis of the construct "find the least y such that p." Namely, for predicates p such as $a - b*q \geq 0$; $y^2 > x$, etc. involving order, one can conduct an efficient search for y, by using a binary search on the interval 0 to a.

This leads to a high level construct of the form, "find (by binary search) the least y less than x such that p." This leads to the algorithm.

(3.3)

1. $a \geq 0 \ \& \ b > 0$
2. $\exists y(a-b*(y+1) < 0)$
3. find(by binary search)
   the least y less than a
   such that $(a-b*(y+1) < 0)$
4. $a-b*y \geq 0 \ \& \ a-b*(y+1) < 0$ by definition of y
5. $b*y \leq a < b*(y+1)$
6. $0 \leq a-b*y < b*y + b - b*y = b$
7. $a = b*y + a - b*y$
8. $r := a - b*y$
9. $0 \leq r < b$
10. $a = b*y + r.$

(3.4) This program is clearer than program 3.2, and its running time is O(log(a)). We can obtain a clear program with running time O(log(a/b)) if we use another search technique to locate a more compact interval in which to do binary search. For instance, we can first look for the least j less than a in the sequence b, $2^1 \cdot b$, $2^2 \cdot b$, $2^3 \cdot b$, ... such that $a - b*j \leq 0$. Then finding $j = 2^n \cdot b < a$, do a binary search on the range $2^{n-1} \cdot b$ to $2^n \cdot b$ for the least y such that $a - b*y \leq 0$. We might call this method "exponential ranging," and we could have the high level constructs

```
find the least(y)(in the range of f) less than (x)
such that (b)

find the least(y)
in the range (x1,x2)
such that (bexp)
```

(3.5) Another language construct fundamental to mathematical proofs and ordinary discourse is the quantifier in its various forms, both bounded (universal $\forall x$ in s, $\forall x \leq d$, etc. and existential, $\exists x$ in s, $\exists x \leq d$, etc.) and unbounded (universal $\forall x$ and existential, $\exists x$). As an example of its role, consider the usual definition of prime numbers in terms of the relation b|a (i.e., b divides a), that is b|a iff $\exists q$ (a = b*q).

Definition: prime(a) iff for all (b) less than (a)(b>1 => ¬(b|a)) or more symbollically, if $\forall b \leq a$ (b>1 => ¬(b|a)).

Other common illustrations relevant to programming come from specifying preconditions on the input to a program such as the condition that the list a(1),...,a(n) contains a unique maximum element, i.e.

$$\exists i \leq n \;\; \forall j \leq n \;(j \neq i \Rightarrow a(j) < a(i)).$$

(3.6) No one doubts that quantifiers are basic to information processing discourse, and most people concerned probably agree that there should be standard symbols or phrases for them (e.g. "$\forall x$", "for all x", or "$\wedge x$" for the universal and "$\exists x$" "$\vee x$", or "there is x" for the existential). But is there any practical reason to have quantifiers as part of the definition of boolean expressions in a programming language?[†] I can think of two good reasons.

1. If we are to precisely express the assertions required to describe algorithms, then we need bounded quantifiers, at least in the assertion language;

   (a) furthermore, if quantified boolean expression (qbexp) could be tested (executed), then various assertions would be suitable debugging aids (in the spirit of flow and check of PL/C), and

   (b) finally, following sound pedagogy, we hope to teach a programming language which is logically complete so that students learning it see it as a fundamental set of mathematical ideas.

[†]In (3.10) we consider the quantifier in imparative statements as well.

2. There are a number of important predicates, such as prime, whose most perspicuous definition is in terms of quantifiers. If these definitions could be used in the programming language, they would greatly simplify the business of giving correctness proofs.

(3.7) Let us illustrate these two points with simple examples. Example 1. Suppose as part of a termination proof we argue that if there is an integer (or other discrete object) not in the range of two functions, f and g over a set S, then the program halts. We might then claim there is such a point without supplying the proof. We assert

$$\exists y \text{ in } S\ \forall x \text{ in } S\ [f(x) \neq y \ \&\ g(x) \neq y].$$

If some problem arises with this program, say for the set $S := \{0,1,2,\ldots,n\}$, then we can check this assertion as a debugging aid, essentially by executing

```
if ∃y in S ∀x in S [f(x) ≠ y & g(x) ≠ y]

then print ('assertion valid')
else print ('assertion false').
```

In the worst case this test might take n^2 steps. In §4.11 we give a PL/C version of this statement. (Rather than writing the conditional we would have a way to selectively "turn assertions on" as with PL/C pseudo-comments.)

This ability to test assertions would be especially appropriate for checking the validity of data, e.g. we could write such code as

```
read (x,y)
x & y are integers
```

or

```
do k = 0 to n; read(a(k)); end
∀i ≤ n (a(i) > 0).
```

Such checking is especially important in file processing applications where it is needed frequently enough to demand an elegant format.

Example 2. Consider the definition of the predicate prime. We could write it directly in a language with bounded quantifiers, but we would want to take advantage of the more "efficient" mathematical definition and write.[†]

Define prime(x) iff for all (y) less than ($[\sqrt{x}]$) ($y > 1 \Rightarrow \neg(y|x)$).

Such a definition of the predicate prime would not only be clear, computable and easily proven correct, but it would be adequately efficient (see §4. for a PL/C version of this definition

(3.8) If we decide to add bounded quantifiers to a programming language, then there is the question of an appropriate version. One simple but quite general approach is to quantify over indexed

[†]It is well known that if $y|x$ then some $y \le [\sqrt{x}]$ divides x; otherwise if $x = y \cdot y_1$ and $y > \sqrt{x}$, $y_1 > \sqrt{x}$, then $x = y \cdot y_1 > \sqrt{x} \cdot \sqrt{x} = x$.

sets. That is, define an enumeration $S := \{s_0, s_1, \ldots, s_n\}$ of a finite set of elements, then allow the expressions

```
for all (x) in (S) (boolean expression)
there is (x) in (S) (boolean expression).
```

The set name S can be the name of a procedure generating S, i.e.

```
S: procedure (n) returns (S-type)
     dcl n integer
     body of S-enumeration
     end S;
```

(3.9) The class of boolean expressions in the language would then have the syntax:

op ::= & | V | => | <=>

bexp ::= (atomic bexp) | (bexp) op (bexp) |
    |¬(bexp) | <u>if</u> bexp <u>then</u> bexp <u>else</u> bexp <u>fi</u> |
    | quant(bexp)

quant ::= for all (id) in (id) | there is (id) in (id)

(3.10) The quantifier can be used as a prefix for imparative statements (actions) as well as for declarative statements (expressions and sentences). For example, in the data processing language ASAP developed at Cornell one is able to say

```
FOR ALL record name = attribute
     PRINT A LIST OF: list of fields.
```

Once the quantifier is available in the language, we can use it imparatively as in

for all (x) in (S) (if b(x) then call p(x)).

In general the syntax is:

quantifier (statment).

(2.11) Finally, I would like to point out that these constructs are probably quite easy to implement. I wrote macros for them in only a few hours. So we are talking about buying interesting, attractive features for a very reasonable price.

§4 Implementing Verification Support in PL/C

(4.1) Modern high level programming languages should permit the expression of concepts and constructs which assist in proving the correctness of programs. Experience with experimental languages and systems which allow mechanical proofs of program correctness may eventually lead to instructional languages which provide mechanical assistance in the coding and checking of correctness proofs. Meanwhile, it is possible to use existing programming language features to improve the programmer's ability to precisely describe his programs and provide proof-like documentation for them.

(4.2) In this section we will show how PL/C [24] can implement some of the features we discussed in §3, assertions, the search operators and the bounded quantifiers.

It is interesting that for the type of problem taught in basic programming courses, see Conway & Gries [23], and Wirth [19] we are able to present very interesting documentation in PL/C.

The implementations rely on the macro-facility of release 7 of PL/C. This facility is described in [23], and we assume familiarity with it.

(4.3) The most basic feature which we need to implement are the assertions. We need a way to write assertions in the program text. This can trivially be done with PL/C comments and pseudo-comments, in which case they have the form:

/* informal comment */ or

/* n if (assertion) then s_1; else s_2; */

where s_1 and s_2 are appropriate actions such as s_1 = null, s_2 = print ('assertion is false').

The first approach is worthless for expressing precise comments and obtaining some mechanical assistance with them.

The second form is not quite so inane and exploits an interesting feature of PL/C, the pseudo-comment (see [23] p. 245). It not only provides compiler checking of assertion syntax, but allows checking the assertion by "turning on" the pseudo-comment. However, the syntactic form of the assertion is quite unpleasant, and its use in this form would be very misleading to a beginning student.

I think a far more satisfactory use of assertions is possible with the PL/C macro facility. We hide the use of the conditional with a macro call. We want this call to be as unobtrusive as possible so that it appears as if the assertion really stands alone in the program text. Here is one way we can accomplish this illustrated on the well-known gcd algorithm (because we can display a wide variety of useful assertion text).

```
*MACRO
   TRUE = '1'B %;
   FALSE = '0'B %;
   READ = GET LIST %;
   PRINT = PUT SKIP LIST %;
   EXCHANGE(X,Y) = BEGIN; N = X; X = Y; Y = N; END;  %;
   INTEGER = FIXED DECIMAL %;
   INT = FIXED DECIMAL %;
   BOOLEAN = BIT(1) %;
   $(N,EXPRESSION) =
     IF (EXPRESSION) THEN; ELSE PRINT
     ('ASSERTION NUMBER',N,',EXPRESSION,',,'IS FALSE);
     %;
*MEND
```

```
        GCD: PROCEDURE OPTIONS(MAIN);
          DCL (X,Y,A,B,N,M,R) INTEGER;
          IN: PRINT('READY FOR INTEGER INPUT,X,Y');
          READ(X,Y);
/*        X AND Y INTEGER                                          */
          IF¬(X>0 & Y>0) THEN BEGIN;
                                PRINT('INPUT ERROR');
                                GO TO IN;
                                END;
/*        GCD(X,Y) = GCD(X,Y) & X>0 & Y>0                          */
          A = X; B = Y;
/*        GCD(X,Y) = GCD(A,B) & A>0 & B>0                          */
          PRINT('A IS', A,' B IS', B);
          DO WHILE(MOD(A,B))=0;
/*         GCD(X,Y) = GCD(A,B)                                     */
           DO;
/*           GCD(A,B) = GCD(B,MOD(A,B))  A THEOREM                 */
             A = MOD(A,B);
$( 1,         MOD(A,B)<B                                            )
             EXCHANGE(A,B);
$( 2,        B~=A                                                   )
             PRINT('A IS',A,' B IS', B);
           END;
/*          GCD(X,Y) = GCD(A,B) & B<A                              */
          END;
/*        GCD(X,Y) = GCD(A,B) & B<A                                */
$( 3,      MOD(A,B) = 0 & B<A
          R = MOD(A,B);                                             )
$( 4,        A = B*(A/B)+R & 0<=R & R<B & B<A                       )
/*        IF R=0 THEN GCD(X,Y) = B                                 */
          PRINT(B,'IS THE GREATEST COMMON DIVOF',X,'AND',Y);
          GO TO IN;
        END GCD;
```

(4.4) Clearly the operation <u>find</u> <u>the</u> <u>least</u>(y) <u>such</u> <u>that</u>(b) is implemented by the asserted program given below.

```
        y:=0
        ∀z(0<z<y  => ¬ b(z))           since there is no such z
        do while (¬ b);
            ∀z(0<z<y  => ¬ b(z)) & ¬ b(y) assume
            ∀z(0≤z<y+1 =>  ¬ b(y))      by def of  z and while con-
            y:=y+1                                     dition.
            ∀z(0≤z<y  => ¬ b(z))        assignment axiom
        od;
       (∃y b(y)) =>  b(y) & ∀z(0≤z<y  =>  - b(z))
```

Remark. If we adopt Hoare's conventions, e.g. that {P} S {Q} means if P is true before S and if S halts then S is true after we could write the final assertion as $b(y) \,\&\, \forall z(0 \le z < y \Rightarrow \neg\, b(z))$. But in this example I prefer to keep the termination hypothesis explicit to contrast with(and thereby highlight)Hoare's convention.

In order to invoke this code whenever we write find the least (y) such that (b), we might think of defining a PL/C procedure. But not only is the syntax wrong, also the parameter b must be called by name so that the relationship between y and b is preserved in the procedure body. There is no call by name of PL/C procedures, but there is with PL/C macros, and with macros we can get the syntax to work out.

(4.5) A simple (but somewhat embarrassing) working implementation of the more complex construct, find the least (y) less than (x) such that (b), is given below, with an example of a program using it. (The code is convoluted because restrictions on the macro facility make it difficult to preserve the syntax we want in the call.)

```
TOP:
*MACRO
     FIND_THE_LEAST (Y) =
     BEGIN;
     GO TO IN;
     NOY: Y=0;
     GO TO OUT;
     IN: DO Y = 0 %;
     LESS_THAN (X) =
     TO X BY 1; %;
     SUCH_THAT (B) =
     IF (B) THEN GO TO OUT;
     END;
     GO TO NOY;
     OUT: PUT SKIP LIST ('END OF SEARCH');
     END;  %;
*MEND
     ROOTS: PROCEDURE OPTIONS(MAIN);
            DCL (X,Y) FIXED DECIMAL;
             INPUT: PUT SKIP LIST
            ('READY TO READ INTEGER INPUT
            GET LIST (X);
            FIND_THE_LEAST (Y)
            LESS_THAN (X)
            SUCH_THAT (Y*Y>=X);
            PUT SKIP LIST
             (Y,'SQUARED >=',X);
            GO TO INPUT;
     END ROOTS;
EOF:
```

(4.6) If the whole story of implementing a search operator was told in (4.4) then there would be little point in introducing the construct, because although the construct represents a fundamental mode of thought, it is subsumed by a more fundamental mode, the while loop. But this is not the whole story. There are interesting variants of this fundamental idea of searching which require more elaborate implementation. Consider the binary search primitive:

<u>find</u> <u>the</u> <u>least</u> (y) <u>in</u> <u>the</u> <u>range</u> (x_1,x_2) <u>such</u> <u>that</u> (b).

It can be implemented by the following simple recursive algorithm:

```
begin
   y:=0
   if 0≤x1≤x2 then y:= search(x1,x2)
                else print('invalid range')
/* procedure definitions follow */

search: procedure(ℓ,u) recursive returns(integer);
        (ℓ,u,m) integer
        (found) function: integer to boolean
  do;
    if  u>ℓ then(print('no value in range'); stop)
    m;=floor((u-ℓ)/2+ℓ)
    if found (m) then return (m)
    if value less than (m)
    then return (search(ℓ,m-1))
    else return (search(m+1,u))
  end;
end search;
/* found is a procedure to determine when a possible least value */
/* satisfying b is found */
   found: proceduce (n) returns (boolean)
          (n) integer
          if b(n) & ¬ b(n-1) then return(true);
                                  else return(false);
   end found;
/* value less than (m) is a procedure to determine whether */
/* the value sought is less than or greater than m */
   value less than: procedure (n) returns (boolean)
        (n) integer
        if b(n) & b(n-1) then return (true);
                              else return (false);
   end value less than;
end;
```

This procedure locates the least y in the range $x_1 \le y \le x_2$ for which b(y), provided b is <u>monotone</u>, i.e. b(y)=>b(j) for all j in $y \le j \le x_2$.

(4.7) In order to properly describe this program we define the following auxillary predicates.

Define $S(x_1,x_2,y)$ iff y is the unique value in the range $x_1 \le y \le x_2$ satisfying found.

Symbolically, iff $x_1 \le y \le x_2$ & found(y) &

$\forall z(x_1 \le z \le x_2$ & found(z) $\Rightarrow$ z=y).

Define $P(x_1,x_2,f)$ iff $\exists y\ S(x_1,x_2,y) \Rightarrow S(x_1,x_2,f(x_1,x_2))$ &

$\neg\exists y\ S(x_1,x_2,y) \Rightarrow f(x_1,x_2)$ = 'no value in ra

We can now annotate as follows:

```
begin  assume b is monotone
  y:=0
  if 0≤x1≤x2 then A1: ∃y S(x1,x2,y) v ¬ ∃y S(x1,x2,y)
                          y:= search(x1,x2)
                      A2: P(x1,x2,search)
                      S(x1,x2,search(x1,x2)) v search(x1,x2) =
                                                    'no value in rang
                else print('invalid range').
fi
0≤x1≤x2  ⇒  S(x1,x2,search(x1,x2)) v search(x1,x2) = 'no val
                                                       in rang
```

In order to prove assertion A2 we need to examine the procedure declarations, search, found and value less than. Note, at A2 the complete assertion in a conventional proof system would be

$$\exists z(\text{search}(x_1,x_2)=z) \Rightarrow P(x_1,x_2,\text{search}).$$

But using Hoare's conventions, the hypothesis that search halts is not made explicit. This saves us writing out many tedious hypothese but may be logically puzzling at first.

The postcondition for binary search is

$\exists z(\text{search}(\ell,u)=z) \Rightarrow$

$(\exists y(S(\ell,u,y)) \Rightarrow S(\ell,u,\text{search}(\ell,u)))$ &

$(\neg\exists y(S(\ell,u,y)) \Rightarrow \text{search}(\ell,u) = \text{'no value'})$

or more briefly

$\exists z(\text{search}(\ell,u)=z) \Rightarrow P(\ell,u,\text{search})$.

The precondition is that ℓ,u are integers, and b is monotone.

```
search: procedure(ℓ,u) recursive returns(integer);
        dcl(ℓ,u,m) integer
        (found, value_less_than).function: integer to boolean
        assume ∀x,y (x1≤x<y≤x2 b(x) => b(y)) /* b is monotone */

        ∃y S(ℓ,u,y) ∨ ¬∃y S(ℓ,u,y)

        if u<ℓ then (¬∃y S(ℓ,u,y); print('no value'))
        ℓ≤u & (∃y S(ℓ,u,y) ∨ ¬ ∃y S(ℓ,u,y))
        m:= floor((u-ℓ)/2)+ℓ
        ℓ≤m≤u & ∃y S(ℓ,u,y) => ∃y(S(ℓ,m,y) ∨ S(m,u,y))

        if found(m) then (S(ℓ,u,m); return(m))
        value_less_than(m) ∨ ¬ value_less_than(m)
        if value less than(m)
   T1:  ∃y S(ℓ,u,y) <=> ∃y S(ℓ,m-1,y)
   T2:  ∃z(search(ℓ,u)=z) <=> ∃z(search(ℓ,m-1)=z)
```

T3: $\exists z(\mathrm{search}(\ell,m-1)=z) \Rightarrow$

$(\exists y\ S(\ell,m-1,y) \Rightarrow S(\ell,m-1,\mathrm{search}(\ell,m-1)))\ \&$

$(\neg \exists y\ S(\ell,m-1,y) \Rightarrow \mathrm{search}(\ell,m=1) = \text{'no value'})$

by induction hypothesis for the usual recursive procedure verification rule.

$\exists z(\mathrm{search}(\ell,u)=z) \Rightarrow$

$\exists y\ S(\ell,u,y) \Rightarrow S(\ell,u,\mathrm{search}(\ell,u)))\ \&$

$\neg \exists y\ S(\ell,u,y) \Rightarrow \mathrm{search}(\ell,u) = \text{'no value'})$

return(search(ℓ,m-1))

else

E1: $\exists y\ S(\ell,u,y) \Leftrightarrow \exists y\ S(m+1,u,y)$

E2: $\exists z(\mathrm{search}(\ell,u)=z) \Leftrightarrow \exists z(\mathrm{search}(m+1,u)=z)$

E3: $\exists z(\mathrm{search}(m+1,n)=z) \Rightarrow$

$(\exists y(S(\ell,m+1,u,y)) \Rightarrow S(m+1,u,\mathrm{search}(m+1,u)))\ \&$

$(\neg \exists y(S(m+1,u,y)) \Rightarrow \mathrm{search}(m+1,u) = \text{'no value'})$

$\exists z(\mathrm{search}(\ell,u)=z) \Rightarrow$

$(\exists y\ S(\ell,u,y) \Rightarrow S(\ell,u,\mathrm{search}(\ell,u)))\ \&$

$(\neg \exists y\ S(\ell,u,y) \Rightarrow \mathrm{search}(\ell,u) \Rightarrow \text{'no value'})$

return(search(m+1,u))

<u>Remark</u>:

(1) We can see from this example that we need a facility for writing parameterized assertions which get treated as procedures.

(2) We do not use Hoare's conventions on halting here because they are inconvenient. Indeed, there is no notation yet devised which conviently covers procedures and functions.

What we need to know here is that if b( ) is a monotone predicate in the sense that

if x<y and b(x) then b(y)

(e.g. $y>x$, $y^2>x$, etc.), then found will locate the unique point at which b(y) first becomes true. This will clearly be the point at which

- b(y-1) & b(y) .

Also given such a monotone b, the predicate "value_less_than" should tell us when the least true value is less than m or greater than m. If b(m) & b(m-1) are true, then the point y such that - b(y-1) & b(y) must be less than m, e.g. y<m.

In order to have a completely specified program we must show that if b() is a monotone predicate in the range $x_1 \le x_2$, then search(x_1,x_2) will find the least y at which b(y). But we have shown that this is the point y such that found(y), and thus the point y such that S(x_1,x_2,y).

So our theorem for the search operator is : If b() is monotone over $\{z | x_1 \le z \le x_2\}$ then after <u>find</u> (by binary search) <u>the least</u>(y) <u>in the range</u>(x_1,x_2) <u>such that</u> (b) is executed, y satisfies b(y) & $\forall z(z<y \Rightarrow \neg b(z))$ if such a y exists.

(4.8) Below is an implementation (again embarrassing) of this search construct. The implementation is strange because of the desire to keep the syntax nice and yet have the parameter information available at the right place.

```
[illegible] = [illegible] SKIP LIST [illegible]
[illegible]
[illegible]
[illegible]
[illegible] ;
 [illegible](X1,X2) FIXED DECIMAL ;
[illegible] = [illegible]_SEARCH'
[illegible] ;
 [illegible]
[illegible]_[illegible] =
  [illegible] = [illegible] ;
  GO TO [illegible] ;
  [illegible]: [illegible] = [illegible] ;
  [illegible] ;
  [illegible] SEARCH [illegible] FIXED DECIMAL,FIXED DECIMAL)
            [illegible] (FIXED DECIMAL));
  SEARCH: PROCEDURE([illegible]) RECURSIVE RETURNS(FIXED DECIMAL);
    [illegible] FIXED DECIMAL ;
      [illegible] DECIMAL) RETURNS(BIT(1));
      [illegible] (FIXED DECIMAL) RETURNS(BIT(1));
      [illegible]_LESS_THAN ENTRY(FIXED DECIMAL) RETURNS(BIT(1));
    [illegible] RETURNS(BIT(1));
      [illegible] DECIMAL ;
    [illegible] ;
[illegible]_[illegible] =
  [illegible] RETURN(TRUE); ELSE RETURN(FALSE);
   [illegible]
[illegible]_[illegible](X1,X2) =
  [illegible] ;
    [illegible] ;
      [illegible] SKIP LIST('SEARCH CALLED ON L=',L,'U=',U);
      [illegible] = [illegible](U-L)/2) + 1;
      IF [illegible] THEN RETURN(M);
      IF U < L THEN BEGIN;
                   PUT SKIP LIST ('NO SUCH VALUE IN RANGE');
                  RETURN(0);
                  END;
      IF VALUE_LESS_THAN(M)
      [illegible] SEARCH(L,M-1);
      [illegible] SEARCH(M+1,U);
    [illegible] ;
  VALUE_LESS_THAN: PROCEDURE([illegible]) RETURNS(BIT(1));
    [illegible] FIXED DECIMAL ;
    IF [illegible] -1) THEN RETURN(TRUE); ELSE RETURN(FALSE);
  [illegible] VALUE_LESS_THAN ;
  [illegible]: [illegible] RETURNS(BIT(1));
    [illegible] FIXED DECIMAL ;
    IF [illegible] -1) THEN RETURN(TRUE); ELSE RETURN(FALSE);
  [illegible] ;
  [illegible] SEARCH ;
  [illegible]: IF ((0<=X1) & (X1<=X2)) THEN Z = SEARCH(X1,X2);
                ELSE BEGIN PRINT('ARGUMENTS TO SEARCH OUT OF RNG');
                       GO TO QUIT;
                    END;
  IF Z=0 THEN GO TO QUIT ELSE GO TO SETY;
[illegible] SKIP LIST('THE END OF THE FIND PROCEDURE');
  [illegible] ;
[illegible] ;
     PUT SKIP LIST ('NO METHOD OF FINDING SPECIFIED');
     STOP ;
     [illegible]
[illegible] ;
    [illegible]
  [illegible] =
    [illegible] FIXED DECIMAL ;
    [illegible] PUT SKIP LIST('READ',N,'INTEGERS');
    [illegible] = [illegible]
  [illegible] = [illegible]
  [illegible]_[illegible] =
    [illegible] J = 1 TO [illegible]
    GET LIST(X); PUT SKIP LIST('THE',J,'TH INTEGER IS',X);
                 X)
```

(4.9) We now look at implementing the bounded quantifiers. Unfortunately I could find no way to treat these as boolean expressions. Thus a quantified statement is an action which sets a value, z, not an expression. However, we want to nest quantifiers, e.g. $\exists x\leq b(\forall y\leq b(s))$ and apply them to boolean expressions. This means that we need a way to distinquish between applying a quantifier to a quantified statement and applying it to a boolean expression. The idea is to use the phrases "we have" and "such that" before applying the quantifier to a boolean expression.

The syntax of the simple versions of the quantifiers given here is

	universal	existental
1.	for all (id) less than (exp)(quant. stmt)	there is (id) less than (exp)(quant. stmt)
2.	for all (id) less than (exp) we have (boolean exp)	there is (id) less than (exp) such that (boolean exp)

where "id" is an identifier, "exp" is an arithmetic expression and "stmt" abbreviates statement. A quantified statement is a statement starting with a quantifier.

We will examine only one type in detail, the bounded universal. The relevant definition and theorem are:

(4.10) Definition: If ℓ is an arithmetic expression x, is a variable, and s is a statement, then let

for all (x) less than (ℓ) (s)

be the statement

```
begin;
   do x=0 to ℓ;
     s
   if ¬ z then go to out;

   end;
   out:
end;
```

Remark: The only statements s to which the quantifier can be applied will be quantified statements, and these will always contain the boolean variable z.

Theorem: Suppose s(x) is a statement with a free variable x, and suppose z is a boolean variable in s, called the (boolean) value of s, and ℓ is an arithmetic expression not containing x free, and not changed by s, then

for all (x) less than (ℓ) (s)

is a statement (where x is no longer free) and z is the (boolean) value of the statement which is <u>true</u> if for all x from o to m, (m the value of ℓ), the value of s(x) is true, and is false if for all x from o to m s(x) halts (the value is defined), but not all values are true.

Proof: To prove this theorem we attach assertions to the algorithm defining the quantifier.

```
begin;
   y integer;
    y(0≤y≤x   =   z=ture & s(y)_z is true)
   do x=0 to ℓ
    y(0≤y≤x   =   z=true & s(y) is true) & m=ℓ
      s

     if - z then (s(x)_z is false & z=false go to out;)
      y(y x   =   z=true & s(y)_z is true) & m=ℓ
   end
    y(y<m+1   =   s(y)_z is true & z=true)

   out:  ;

    y(y<m+1   =   s(y)_z is true & z=true) v  x(s(x)_z is false) &
                                                          z=false
```

(4.11) Below are the macro definitions which implement these quantifiers along with some examples of their use. To use them in defining predicates we have introduced the marco

define(predicate)(argument) iff quantified statement.

This apparatus and much of the complexity of the quantifier definition arise from the fact that we can not make the quantifiers part of the real boolean expressions of the language as we propose in §3 and §6.

```
FORALL(ID)= BEGIN;
                DO ID=0 TO A$%;
A$LESSTHAN(EXP)= EXP BY 1; A$$%;
A$LESS(EXP)= EXP BY 1; A$$%;
A$$(STMT)= STMT; A$$$%;
A$$$ = IF ¬Z THEN GO TO OUT;
          END;
          OUT: ;
          END; %;
A$$WEHAVE(EXP) = IF (EXP) THEN Z=TRUE;
                         ELSE Z = FALSE;A$$$%;
THEREIS(ID)= BEGIN;
                DO ID = 0 TO E$%;
E$LESSTHAN(EXP)= EXP BY 1;E$$%;
E$LESS(EXP)= EXP BY 1;E$$%;
E$$(STMT)= STMT;E$$$%;
E$$$ = IF Z THEN GO TO OUT;
          Z = FALSE;
          END;
          OUT: ;
          END; %;
E$$SUCHTHAT(EXP)= IF (EXP) THEN Z = TRUE;
                         ELSE Z= FALSE;E$$$%;
E$$SOTHAT(EXP)= IF (EXP) THEN Z=TRUE;
                         ELSE Z=FALSE;E$$$%;
IFF(CODE)=  CODE; RETURN(Z); %;
DEFINE( ID ) = ID: PROCEDURE $BODY$%;
$BODY$(ARG)= (ARG) RETURNS(BOOLEAN);
                DCL Z BOOLEAN;
                DCL (ARG) INTEGER; %;
```

§5 Expressing constructions in formal proofs

A programming language with formal assertions has a role of its own; it is destined to influence even the style of mathematical argument. Speculating on these influences is beyond the scope of this note, but it would be a shame not to point out a specimen, however meager. Anyone who tries teaching with such a language will see numerous pleasing buds of things to come.†

Suppose we compare the conventional mathematical approach to a proof with an "algorithmic approach". To keep things simple, consider proving for positive integers, x,p,q, that

$$\forall x\ \exists p\ ((\mathrm{prime}(p)\ \&\ p|x)\ \text{or}\ (\mathrm{prime}(x)\ \&\ p=x)).$$

The conventional constructive proof is:††

For any x either prime(x) or ¬ prime(x) because for each q between 2 and x either q|x or ¬ q|x. If no such q divides x, then q is prime and we take p=x.

If some q|x, then pick out the least q that divides x, call it p (it exists by the least number principle). This p must be prime because if there is n, $2 \leq n < p$, n|p then n|x and this contradicts our choice of p. Q.E.D.

† My own experience is that the basic concepts of logic become easily digestible morsels to students raised on one of these more expressive programming languages. I think this happens because the concepts become part of the down-to-earth meat & potatoes of programming.

†† For reasons which are probably clear, but will be mentioned in §6, it does not make sense to look at non-constructive proofs.

Now consider an algorithmic proof, given by presenting an asserted program.

DEQ

x integer; $x>0$

$q:=2;\ q \geq 2$

define $A(q,x)$ iff $\forall z(2 \leq z < q \Rightarrow \neg(z|x))$

$A(q,x)$

while $(\neg (q|x)\ \&\ q < x)$ do $A(q+1,x);\ q:=q+1;\ A(q,x);$ od;

$q \geq 2\ \&\ (q = x\ \&\ A(q,x)) \vee (q < x\ \&\ q|x\ \&\ A(q,x))$

$q = x\ \&\ A(q,x) \Rightarrow \text{prime}(x)$

$q|x\ \&\ q < x\ \&\ A(q,x) \Rightarrow \text{prime}(q)$

if $\exists n\ (2 \leq n < q\ \&\ n|q)$ then $n < q\ \&\ n|x$

contradicting $A(q,x)$

QED

This algorithmic proof may look more formidable than the mathematical proof, but that is mainly because it is nearly a completely formal proof, whereas the other proof is quite informal.

The same spirit can be detected in the following application of the above proof. Suppose we consider proving

$$\forall n\ \exists p\ (p > n\ \&\ \text{prime}(p))$$

There must be a prime between n and $n!+1$ because no $q \leq n$ divides $n!+1$ $(n!+1 = b*q+1)$ and if $n!+1$ is not itself prime then it has a prime divisor $p < n!+1$. So there is a prime $n < p \leq n!+1$.

Now a more algorithmic proof uses a function

g(x) := the least(y) less than or equal(x)
such that ($2 \le y \Rightarrow y|x$).

This is the function we defined above. Clearly

$$\text{prime}(x) \text{ iff } g(x) = x$$
$$\neg\text{prime}(x) \text{ iff } g(x) < x$$

and also prime (g(x)).

So our algorithmic proof is

1. prime(g(n!+1)) & g(n!+1) $\le$ n!+1.
2. $\forall m(m \le n \Rightarrow \neg(m|n))$ hence
3. prime(g(n!+1)) & n < g(n!+1) $\le$ n!+1.

Q.E.D.

When we work with algorithmic proofs we build up a library of basic constructions and their descriptions. The constructions are as important as the theorem (description). Sounds a bit like good old Euclidean geometry doesn't it? Who has a neat construction for an ellipse?

§6. Suggestions for Future Languages

We have demonstrated how to use pseudo-comments in PL/C to precisely document a program as a step toward verifying its correctness. These pseudo-comments offer some mechanical assistance to verification because the assertions they make can be checked at runtime for particular values of the variables. (In a real verification, the assertions would be proved at compile time for all values of the free variables.)

Contemporary programming languages offer a limited vocabulary for describing relationships among variables; in PL/C there are only boolean expressions. With the PL/C macro feature we have expanded this vocabulary to include bounded quantifiers. We have likewise used macros to provide commands which facilitate natural and precise descriptions of problems.

Future programming languages should offer more convenient ways to take these very simple steps toward writing programs that are easier to understand and verify. Such languages should at least support a flexible assertion facility with a rich vocabulary for specifying problems and program conditions. (Such languages are often called assertion languages or specification languages.) As a minimum this language of assertions should contain bounded quantifiers. Though I have not argued the point in this report, it should also contain a primitive set theory (perhaps only for descriptive purposes with no option to execute statements about sets).

All future languages should provide a macro-like facility designed for writing high level commands and <u>higher level assertions</u>.

Asking for such features as assertions and quantifiers appears to be calling for added complexity at a time when the prevailing opinion about language design favors simplicity. But these features create a balance between the expressiveness of commands and the expressiveness of assertions. A good programming language of the future will be a logical system as well as a programming system. It will offer mechanical assistance for rudimentary reasoning about programs as well as for writing them.

The elementary logical system that we incorporate in a programming language should be designed with an eye on the potential impact of verifying compilers and high level proof checkers. Hopefully there will be an orderly progression from the notion of an asserted program (program with assertions) to the notion of a proved program. Thus the assertions should be a subset of the formulas in a logic for reasoning about programs.

Instructional languages should clearly provide the means for precise reasoning about programs because the task of programming instruction is teaching how humans understand programs as well as how machines execute them.

Acknowledgements

I would like to acknowledge the assistance and encouragement of Professor Richard Conway, James McGraw and Carl Hauser who have commented on earlier forms of this manuscript.

References

[1] de Bakker, J.W., Recursive Procedures, Mathematical Center, Amsterdam, 1971.

[2] de Bakker, J.W. and L.G.L.T. Meertens, "On the Completeness of the Inductive Assertion Method", JCSS, 11, 3, Dec. 1975, pp. 323-357.

[3] Clarke, E., "Completeness and Incompleteness Theorems for Hoare-like Axiom Systems", Ph.D. Thesis, Cornell University, 1976.

[4] Constable, R.L., "Constructive Mathematics and Automatic Program Writers", Proceedings of IFIP Congress, Ljubljana, 1971, pp. 229-233.

[5] Dahl, O.J., E.W. Dijkstra, and C.A.R. Hoare, Structured Programming, Academic Press, 1972.

[6] Floyd, R.W., "Assigning Meaning to Programs", Proceedings of Symposia in Applied Mathematics, Vol. XIX, American Math. Soc., 1967, pp. 19-32.

[7] Good, D.I., R.L. London, W.W. Bledsoe, "An Interactive Program Verifier", IEEE Transactions on Software Eng., Vol. SE-I, March 1975, pp. 59-67.

[8] Hoare, C.A.R., "An Axiomatic Basis for Computer Programming", Comm. ACM 12, Oct. 1969, pp. 576-581.

[9] Hoare, C.A.R., "Procedures and Parameters, An Axiomatic Approach", Symposium on Semantics of Algorithmic Languages, E. Engeler, ed., Springer Verlag, Lecture Notes in Math. No. 188, 1971, pp. 102-116.

[10] Hoare, C.A.R. and N. Wirth, "An Axiomatic Definition of the Programming Language Pascal", Acta Informatica 2, 1973, pp. 333-355.

[11] London, R.L., "Bibliography of Proving the Correctness of Computer Programs", Machine Intelligence 5, Edinburgh Univ. Press, 1970, pp. 569-580.

[12] Manna, Z. and J. Vuillemin, "Fixedpoint Approach to the Theory of Computation", C.ACM, 15,7, 1972, pp. 528-536.

[13] Manna, Z., S. Ness, and J. Vuillemin, "Inductive Methods for Proving Properties of Programs", C.ACM, 16, 8, 1972, pp. 491-502.

[14] McCarthy, J., "A Basis for a Mathematical Theory of Computation", Computer Programming and Formal Systems, North-Holland, Amsterdam, 1963, (ed. Braffort & Hirschberg), pp. 33-70.

[15] Milner, R., "Implementation and Applications of Scott's Logic for Computable Functions", Proc. ACM Conference on Proving Assertions About Programs, Los Cruces, New Mexico, 1972, pp. 1-6.

[16] Park, D.M.R., "Fixedpoint Induction and Proofs of Program Properties", Machine Intelligence 5, Edinburgh, 1969, pp. 59-78.

[17] Pratt, T.W., Programming Languages: Design and Implementation, Prentice-Hall, Englewood Cliffs, 1975, pp. 530.

[18] Waldinger, R.J., "Constructing Programs Automatically Using Theorem Proving", Ph.D. Thesis, Carnegie-Mellon Univ., 1969.

[19] Wirth, N., Systematic Programming, An Introduction, Prentice-Hall, 1973, pp. 169.

[20] Constable, R.L., "Language Features that Support Program Verification (illustrated in PL/C)", Cornell University Technical Report 76-276, April 1976.

[21] Constable, R.L. and J.E. Donahue, "An Elementary Formal Semantics for the Programming Language PL/CS", Cornell University Technical Report 76-271, March 1976.

[22] Conway, R.W., "PL/CS - A Highly-Disciplined Subset of PL/C", Cornell University Technical Report 76-273, March 1976.

[23] Conway, R.W. and D. Gries, "An Introduction To Programming, A Structured Approach Using PL/1 and PL/C-7. Second Edition, Winthrop, Cambridge, MA, 1975, 509p.

[24] Conway, R.W. and T.R. Wilcox, "Design and implementation of a Diagnostic Compiler for PL/I, Comm. ACM, 16,3, 1973, 169-179

[25] Kernighan, Brain W. and P.J. Plauger, "The Elements of Programming Style, McGraw Hill, N.Y. 1974, 147pp.

[26] Constable, Robert L. and J.E. Donahue, "An Elementary Formal Semantics for the Programming Language PL/CS, Cornell Computer Science Dept. Technical Reprot TR-76-271, March 1976, 37pp.

[27] Ashcroft, E.A. and W.W. Wadge, "Lucid - A Formal System for Writing and Proving Programs, SIAM J. on Computing, 5,3, Sept. 1976.

[28] Manna, Zohar and Richard Waldinger, "Towards Automatic Program Synthesis Symposium on Semantic of Algorithmic Languages, Lecture Notes in Math, 188, Springer-Verlag, Berlin, 1971, 270-310.