A PL/CV PRECIS

Robert L. Constable
Scott D. Johnson

Department of Computer Science
Cornell University
Ithaca, NY 14853

ABSTRACT: PL/CV is a new formal system which mixes commands and assertions. It includes axioms and rules for a theory of programming over integers and characters. Since arguments in the theory can be checked by the PL/CV Proof Checker, the system offers an approach to mechanical program verification. The Proof Checker is efficient enough for classroom use. Early experience with PL/CV indicates that it nicely supports formal verification of elementary arguments. Continued work should enable the formalization of non-elementary reasoning as well.

1. Introduction

1.1 Overview

Here is a brief presentation and discussion of PL/CV, a new formal system which mixes commands and assertions. The system contains a programming language whose commands can be compiled and executed. The rules for writing arguments constitute a formal logic, including a theory of the predicate calculus, a theory of arithmetic, and a "theory of programming". The system offers a style of mechanically assisted reasoning in that arguments in the calculus can be checked by the PL/CV Proof Checker. Since these arguments can be about programs, the system provides a new approach to program verification.

The PL/CV system illustrates several interesting principles of programming languages. The ideas and techniques used to make high level programming languages viable are employed to make this high level logic viable, e.g. block structuring of proofs, highly flexible definition and abbreviation facilities, conformity with ordinary mathematics, etc. Moreover the implementation was designed to make proof checking efficient (but for a few well-contained exceptions in arithmetic and tautology checking). The PL/CV system is the first working example of a new type of programming system in which the logical and command languages are thoroughly integrated. The logic is novel, yet simple, and the implementation permits inexpensive experimental verification, of a type distinct from that possible in other verification systems.

1.2 Guiding Principles

The design of PL/CV was begun in 1975, a pilot system was tested in 1976. The version described here was available as of July 1978. The design was guided by experience with the pilot system and a few simple criteria, which we enumerate.

(1) A desire to investigate concepts from the mainstream of modern computing

This work was supported in part by NSF grants DCR-74-14701, MCS-76-14293, and MCS-78-00953.

Proceedings of POPL-1979, pp 7-20.

suggests that the command language should be the core of an Algol-like procedural language, suitably restricted to commands for which simple rules can be found (as in Pascal, see [12]).

(2) Experience with mathematical reasoning (and nearly a hundred years of study by logicians) suggests that the deductive structure of the system should be based on the familiar and well-known predicate calculus.

(3) To permit compact natural expression of computational reasoning, the steps of an argument should allow commands and programs (hence arguments are "dynamic" in the sense of [24]). Conversely, programs (especially long ones) should permit arguments within them, as in the asserted programs of [10].

(4) Unnecessary notation should be avoided. We do not use Hoare's primitive {P}A{Q}, nor do we introduce special elements to denote undefined objects. The only notation used is that of ordinary mathematics and procedural programming, along with syntax to combine them.

(5) It should be tolerable for people to write and read complete formal proofs in the system. Hence, the formal system should mimic as closely as possible informal methods. We therefore adopt the Natural Deduction style calculus of Gentzen and Prawitz [25]. This system should be to ordinary formal proofs as Algol is to Turing machines, and it should be as understandable to the average programmer as Algol is.

(6) The formal proofs should be good documentation for the reasoning behind the program. Thus, it must be possible to argue directly about the program without translating it entirely into logic (as is done in most other verifiers, e.g. Stanford Pascal, SRI system, USC system, etc.).

(7) It should be possible to efficiently check the correctness of these formal arguments with a computer. The use of the proof checker should be inexpensive enough to encourage widespread experimentation with the system, and to permit its use in the classroom.

(8) Based on our experience dealing with partial correctness, we believe the logic should be confined to describing terminating computations. Commands cannot be introduced into an argument unless they are shown to terminate; partial correctness arguments can be considered only by explicitly assuming termination.

(9) The first version of the system should meet the above goals on the class of elementary arguments encountered in beginning undergraduate courses in programming and discrete mathematics. Modern logic has provided an organization of mathematics which makes it possible to discern, at least roughly, this class of elementary arguments. By beginning with these we can concentrate on the issues of whether a logic of this type is feasible at all. If we succeed on the elementary arguments, we can then examine more complex reasoning, e. g. that involving user defined data types, higher type functions, metamathematical reasoning, etc.

1.3 Comparison with other systems

The command language of PL/CV is PL/CS, a disciplined subset of PL/I which is accepted by the PL/CS, PL/C, and PL/I compilers. PL/CV resembles the Stanford Pascal Verifier [13] in that its command language is a "structured" subset of an Algol-like language. However, unlike the Pascal verifier, PL/CV does not translate asserted programs into the predicate calculus, but instead provides rules for reasoning directly about commands. In this

regard, the system is similar to Dijkstra's calculus, [8].

Unlike the Stanford Pascal Verifier, and all other program verifiers we know, PL/CV does not employ a theorem prover. Instead arguments are checked by a Proof Checker. In this regard, the system is similar to AUTOMATH [7] and FOL [29]. Although we have been particularly encouraged by the success of AUTOMATH's methods, PL/CV uses a number of decision procedures and automatic rules which allow the user to take fairly large proof steps.

Unlike verifiers for Lisp [1] or variants such as Cartwright's Typed Lisp [2] and Milner's LCF [21], PL/CV is concerned with procedural languages and their actions, e. g. assignment. The PL/CV logic does not consider its domains to be lattices or CPOs as in LCF. It deals directly with partial functions over integers and strings. However, in general philosophy, PL/CV is quite close to LCF and was directly inspired by it. Both systems present their logic in great detail and are concerned with the structure of proofs. Both also provide powerful mechanisms for abbreviating proofs. LCF, unlike PL/CV, treats non-elementary forms of arguments, including the ability to formalize the metamathematics. We expect to draw heavily upon LCF when we attempt extend PL/CV.

PL/CV is based on a constructive logic (all automatic rules are constructive), though classical reasoning can be expressed by making explicit classical assumptions. But unlike other systems based on a "constructive method" in a quantifier free setting (e. g. Wegbreit [28]), PL/CV uses the full predicate calculus. A full predicate calculus is fundamental to our intention of providing a very expressive logic. Indeed, soon we will permit full constructive second order predicate calculus.

In the remainder of this paper we consider how these goals led to the PL/CV system, and we briefly discuss what we have learned from using the system (there being much more to learn). The resulting logic is simple enough that it has been used by undergraduates, yet it is sufficiently rich to permit the serious study of the problems of computational reasoning, program verification, and the design of understandable programming languages.

2. The Logic

2.1 The Form of Proofs

The PL/CV logic can be viewed as an extension of a natural deduction presentation of the predicate calculus that includes commands, and that takes the concept of an argument, rather than a formula, as the central object. To understand this viewpoint one must know how natural deduction style logics are organized. For each logical operator there is a rule for introducing the operator into an argument and a rule for eliminating that operator. Proofs are trees whose nodes are formulas; the root is the formula proved and the leaves are assumptions. For example, letting &, |, and => denote and, or, and implies, here is an informal proof of (a|b => c) => ((a=>c) & (b=>c)):

Assume (a|b => c). Then a=>c, since if a is assumed, a|b follows immediately, and c follows from a|b and the assumption that (a|b => c). Likewise, from b deduce a|b and thence c, establishing b=>c. qed

This argument would be formalized in logic books as the natural deduction tree in Figure 1. Lines (1) and (6) are the main assumption. Notice the assumption is replicated each time it is needed for the application of an inference rule. Lines (2) and (7) are also assumptions. From line (2), line (3) arises by introducing

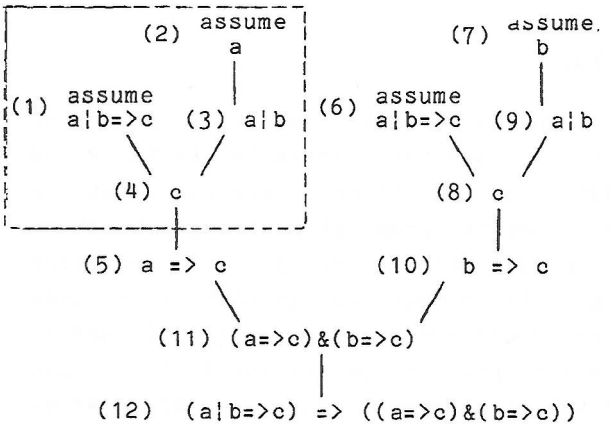


Figure 1

the operator, ¦, into line (2). Line (4) follows by using line (3) to eliminate the implication operator, =>, from line (1). Line (5) introduces =>. The rule justifying line (5) requires the subargument, in dotted lines, which deduces c from the assumptions a and a¦b=>c. At line (5), we say the assumption "assume a" at line (2) has been discharged.

In PL/CV, this two-dimensional tree structure is replaced by a linear representation of text. The linear form is rendered compact by eliminating redundant occurences of lines in favor of rules for accessing a line from a point in the proof. Moreover, the scopes of assumptions and their intended order of elimination is made apparent by block structure. The PL/CV version of this argument is in Figure 2. The proof is complete without the lines in comment delimiters, /#, #/. (The line numbers are also not part of the proof). The tree form has been replaced by a more compact and readable representation. The justification "BY INTRO" indicates that the operator => will be introduced by the argument that follows. Much of the above proof is redundant, and has been written out here for the sake of clarity. By making use of abbreviations and automatic rules, the lemma in Figure 2 can be proved as follows:

```
(1)  LEMMA1:
(2)  (A¦B=>C) => (A=>C)&(B=>C) BY INTRO,
(3)  PROOF;
(4)     ASSUME P1: A¦B => C;
(5)     A => C BY INTRO,
(6)     PROOF;
(7)        ASSUME A;
(8)        P2: A ¦ B;
(9)        C;  /# FROM P1 AND P2 #/
(10)    QED;
(11)    B => C BY INTRO,
(12)    PROOF;
(13)       ASSUME B;
(14)       P3: A ¦ B;
(15)       C;   /# FROM P1 AND P3 #/
(16)    QED;
(17) QED;
```

Figure 2

```
(A¦B => C) => (A=>C) & (B=>C) BY INTRO,
PROOF;
   A=>C BY INTRO;
   B=>C BY INTRO;
QED;
```

Here are two other PL/CV propositional arguments:

```
A¦(B&C) => (A¦B) & (A¦C)
   BY INTRO, CASES, A¦(B&C);

(A=>B) => (~B => ~A)
   BY INTRO, INTRO, INTRO;
```

The justification "BY INTRO" indicates that the main connective of the assertion will be introduced. In the second example, A¦(B&C) is implicitly assumed by "INTRO". From that assumption, (A¦B) & (A¦C) can be proved by a case analysis on A¦(B&C). If A holds, "clearly" (A¦B) & (A¦C); likewise if B&C holds. The automatic reasoning involved in "clearly" is discussed in section 2.3.

Notice that these sample PL/CV proofs are similar to their informal counterparts, yet they are completely formal, and have been checked by a machine.

2.2 Accessibility

In an ordinary linear representation of a proof, every lexically previous line is available as an hypothesis in subsequent deductions. But in a block structured proof, normal scope rules apply, and

it is an error to refer into proof blocks from outside them. For example, in the PL/CV proof of "LEMMA1", the assertion C is made at line (9), but this assertion is clearly not accessible as a conclusion at line (17). Other restrictions on accessibility are necessary because commands are allowed in arguments. Some of these are discussed in section 3.3.

2.3 Automatic Rules and Proofs

Certain rules correspond to steps of reasoning that most people perform automatically "in their heads." In PL/CV, there are three classes of automatic rules: Propositional rules, Equality rules, and Normalizing rules. Except for => introduction, ¦ elimination, contradiction, and ~ introducton, all of the propositional rules are automatic. If all of the propositional rules were automatic, the result would be a tautology checker. Since the tautology problem is coNP-complete, a limited set of automatic rules was settled upon. The algorithm used for checking propositional inferences follows Prawitz's normal form theorem for natural deduction [25]. The automatic equality rules in PL/CV comprise the theory of equality with uninterpreted function symbols. The algorithm used to decide this theory is due to Kozen [15], and Nelson and Oppen [23], and also Downey and Sethi [9]. The Normalizing rules are the associativity and commutativity of certain propositional and arithmetic connectives, and simple arithmetic facts, such as the equivalence of A<B and ~(A>=B) and arithmetic manipulations of constants. Not all of the automatic rules may be used in a single proof step. As discussed in Krafft [16], allowing unrestricted combinations of these rules leads to exponential-space complete problems.

The use of automatic rules and other shorthand notations for proofs are the main methods used to condense proofs. A checker of both classical and intuitionistic tautologies is available with PL/CV, but we have not found its use necessary or even especially convenient. It is our claim that propositional arguments which we understand well correspond to very short PL/CV proofs. Furthermore, any propositional reasoning that we do not understand well should be spelled out as an aid to the reader, and not buried inside a time-consuming decision procedure.

2.4 Rules for Commands

Commands are introduced into arguments and eliminated from them, just as logical operations are. Thus, for each command there is an introduction and elimination rule. For example, the introduction rule for the DO-WHILE loop is the usual termination rule for loops. The elimination rule corresponds to Hoare's WHILE rule. Here is a simplified (no loop-exits) version of the combination of the DO-WHILE introduction and DO-WHILE elimination rule:

```
(1)  {BEXP well-defined}
(2)  {SOME N FIXED . (N)=0 & T(N)}
(3)  {P}
(4)  DO WHILE(BEXP);
(5)  /*/ ARBITRARY N FIXED WHERE T(N); */
(6)  /*/ ASSUME P; */

(7)       <proof statements>

(8)       {~T(0)}

(9)       <loop body>

(10)      {BEXP well-defined}
(11)      {T(N-1)}
(12)      {P}
(13) END;
     ------------------------------------

(14) P & ~BEXP
```

The hypotheses of the introduction rule are lines (2), (5), (8), and (11). The variables of the body of the loop are related by T to a free variable that decreases by one on each iteration of the loop. Since the variables cannot be re-

lated to zero, the loop must halt. The conclusion of the introduction rule is the loop itself: one introduces the loop by proving it terminates.

The hypotheses of the elimination rule are lines (3), (6), and (12). At lines (1) and (10), "BEXP well-defined", is an assertion which guarantees that BEXP will evaluate without error, that is, array subscripts are within bounds, and the arguments to each called function are within its domain. Line (3) asserts that P, the loop invariant, holds before the loop execution begins. Line (6) is an induction hypothesis. The induction step, that P holds after the execution of the loop body, must be proved at (12). The conclusion is line (14). How this rule is checked by the Proof Checker is discussed in section 3.

2.5 A Short Example

As an example of how the DO-WHILE rules are used in a program, Euclid's division algorithm, an obligatory "benchmark" example, is shown in Figure 3. It is shown exactly as it is input into the Proof Checker. So that the same input may be used for both the compiler and the proof checker, all of the PL/CV proof statements are enclosed in special comment brackets. Other syntactic conventions which may not be immediately clear are that FIXED is the attribute used to indicate integral (FIXED DECIMAL) values, and SOME and ALL are used for the existential and universal quantifiers.

2.6 A Longer Example

The procedure FIND_MODE computes the mode of a one dimensional sorted array. The mode of an array is an element occurring most frequently. We define this property in terms of a basic function called COUNT(X,A,L,H) which counts the number of occurrences of X in array A from index L

```
DIVIDE: PROCEDURE(A, B, Q, R);
/* EUCLID'S DIVISION ALGORITHM */

DECLARE (A, B) FIXED /*: READONLY */;
DECLARE (Q, R) FIXED;
/*/ ASSUME A>=0, B>0;                          */
/*/ ATTAIN A = B*Q+R, 0<=R<B;                  */
/*/ A = B*0 + A BY ARITH;                      */

R = A;
Q = 0;

/*/ SOME I FIXED . (I>=0 & R<=I)               */
/*/                      BY INTRO, R;          */

DO WHILE(R-B >= 0);
/*/  ASSUME A = B*Q+R, R>=0;                   */
/*/  ARBITRARY I FIXED WHERE R<=I;             */
/*/  ~(R<=0) BY ARITH, R-B>=0, +, B>0;         */
/*/  A = B*(Q+1) + (R-B)                       */
/*/          BY ARITH, A=B*Q+R;                */
/*/  R-B <= I-1 BY ARITH, R<=I, -, B>0;        */

     R = R-B;
     Q = Q+1;
END;
/*/ R<B BY ARITH, ~(R-B >= 0), +, B=B;         */
/*/ A = B*Q+R; 0<=R<B;                         */

RETURN;
END DIVIDE;
```

Figure 3

to index H. A(I) is a mode of A if COUNT(A(I),A,LOW,HIGH) is greater than or equal to COUNT(X,A,LOW,HIGH) for any X, where LOW and HIGH are the low and high bounds of the array A. In order to prove the correctness of FIND_MODE, we need four lemmas involving COUNT. These lemmas have been listed without proof just before the program. The proofs of these lemmas in turn require several facts about COUNT. All of these facts and lemmas have been proved, but the proofs are not included here.

This example illustrates several features of PL/CV proofs. First, quantifiers and definitions are used extensively. The definition FOR A(*) FIXED DEFINE SEGREGATED(A) = ... , for example, says that SEGREGATED is a property of one dimensional integer arrays defined by the formula or assertion after the equality. A second noteworthy feature of the example is that the asserted program FIND MODE

uses another program, COUNT, in the proof text but not in the program text. This feature is used extensively in the proofs of the lemmas not shown, where various mathematical theorems are proved inductively using DO-WHILE and DO-INDEX loops. In the proof of the basic facts about COUNT, this mixing of program rules in proofs is even more striking; COUNT is an iterative function, essentially:

```
NUMBER = 0;
DO I = LOW TO HIGH;
IF X=A(I) THEN NUMBER = NUMBER+1;
END;
```

but in the proof, COUNT is called recursively and the recursive function rule is used for the inductive proof.

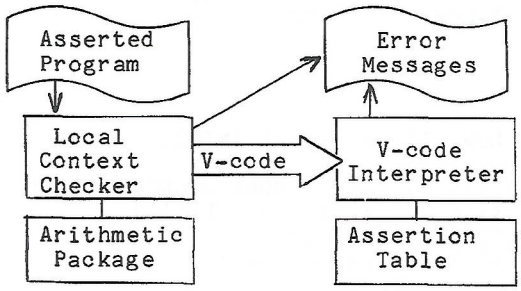
Although readers not familiar with PL/CV may have trouble with some of the notation used in the logic (such as ALLEL for the name of the all elimination rule), we hope this example will be informative.

```
*PROCESS
 COUNT:PROCEDURE(X,A,L,H) RETURNS(FIXED);
 DECLARE (X,A(*),L,H) FIXED;
 /*/
 ASSUME '1'B; /# '1'B MEANS "TRUE" IN PL/I #/
 ATTAIN H < L     => COUNT(X,A,L,H) = 0,
        X = A(H) => COUNT(X,A,L,H) = COUNT(X,A,L,H-1)+1,
        X~= A(H) => COUNT(X,A,L,H) = COUNT(X,A,L,H-1),
        .
        . (rest of COUNT omitted)
        .

 END COUNT;

*THEOREM
 /*/
 /# DOM(A,I) SAYS THAT I BELONGS TO THE DOMAIN OF THE ARRAY A.          #/
 /# THE DOMAIN OF AN ARRAY A IS THE INTERVAL FROM THE LOW BOUND OF A,   #/
 /# LBOUND(A,1), TO THE HIGH BOUND OF A, HBOUND(A,1).                   #/
 /# (LBOUND AND HBOUND ARE BUILTIN PL/I FUNCTIONS).                     #/
 FOR (A(*),I) FIXED DEFINE DOM(A,I) = LBOUND(A,1)<=I<=HBOUND(A,1);

 /# AN ARRAY IS SEGREGATED IF AND ONLY IF IDENTICAL ELEMENTS OCCUR      #/
 /# CONSECUTIVELY.                                                      #/
 FOR A(*) FIXED DEFINE SEGREGATED(A) =
      ALL (I,J,K) FIXED WHERE I<=J<=K & DOM(A,I) & DOM(A,K) & A(I)=A(K) .
      A(J)=A(K);

 /# EXTENDING THE INTERVAL COUNTED OVER BY ONE WILL INCREASE THE COUNT #/
 /# BY ONE OR LEAVE IT THE SAME.                                       #/
 EXTEND_COUNT: ALL (X,A(*),U) FIXED WHERE U<=HBOUND(A,1) .
     COUNT(X, A, LBOUND(A,1), U-1)  <=
     COUNT(X, A, LBOUND(A,1), U)    <=
     COUNT(X, A, LBOUND(A,1), U-1)+1 BY . . . (proof omitted)

 /# IN A SEGREGATED ARRAY, IF A(I-L)~=A(I), THEN THERE ARE NO MORE     #/
 /# THAN L OCCURRENCES OF A(I) UP TO I.                                #/
 BLOCK_UPPER:
 ALL (A(*),I,L) FIXED WHERE SEGREGATED(A) & DOM(A,I) & DOM(A,I-L) & L>=0 .
      ( A(I-L)~=A(I) => COUNT(A(I),A,LBOUND(A,1),I) <= L )
 BY . . . (proof omitted)

 /# IN A SEGREGATED ARRAY, IF A(I-L)=A(I), THEN THERE ARE AT LEAST L+1  #/
 /# OCCURRENCES OF A(I) IN A UP TO I.                                   #/
 BLOCK_LOWER:
 ALL (A(*),I,L) FIXED WHERE SEGREGATED(A) & DOM(A,I-L) & DOM(A,I) & L>=0 .
      ( A(I-L)=A(I) => COUNT(A(I),A,LBOUND(A,1),I) >= L+1)
 BY. . . . (proof omitted)
```

```
*PROCESS
 FIND_MODE:PROCEDURE(MODE,A,L);
 DECLARE A(*) FIXED /*: READONLY */;
 DECLARE (MODE,L) FIXED; /* L IS THE LENGTH OF MODE BLOCK */

 /* ********** DEFINITIONS ***************************  */
 /* THE MODE OF AN ARRAY IS AN ELEMENT OCCURRING AS     */
 /* MANY TIMES AS ANY OTHER ELEMENT.                    */
 /*/ DEFINE LOW = LBOUND(A,1);                         */
 /*/ DEFINE HIGH = HBOUND(A,1);                        */
 /*/ FOR I FIXED DEFINE MODE_BOUND(I)=                 */
 /*/     ALL X FIXED . COUNT(X,A,LOW,I)<=L;            */
 /*/ FOR I FIXED DEFINE MODE_TO(I)=                    */
 /*/       COUNT(MODE,A,LOW,I-1)=L & MODE_BOUND(I-1);  */

 /* ********** PROGRAM SPECIFICATIONS ****************  */
 /*/ ASSUME SEGREGATED(A);                             */
 /*/ ATTAIN MODE_TO(HIGH+1);                           */

 DECLARE I FIXED;

 /* ********** BODY OF THE ARGUMENT ******************  */
 /* IN A SEGREGATED ARRAY, THE LENGTH OF A  BLOCK OF    */
 /* CONSECUTIVE ELEMENTS IS THE NUMBER OF OCCURRENCES   */
 /* OF THAT ELEMENT.  TO FIND THE MODE OF A, WE FIND    */
 /* THE LONGEST BLOCK OF CONSECUTIVE ELEMENTS.          */

 /* ************** BASIS CASE ************************** */
 /* AS THE BASIS STEP OF OUR INDUCTIVE ARGUMENT TO      */
 /* FIND THE MODE,CONSIDER THE VACUOUS CASE OF THE      */
 /* EMPTY ARRAY.  THE LOOP INVARIANT HOLDS VACUOUSLY.   */

 L = 0; /* INITIALLY THERE ARE NO BLOCKS */
 /*/  0 <= 0 <= LOW-LOW;
       LOW-1 < LOW;
       COUNT(MODE,A,LOW,LOW-1)=0 BY FUNCTION,COUNT(MODE,A,LOW,LOW-1);
       ALL X FIXED . COUNT(X, A, LOW, LOW-1)<=0 BY INTRO,
       PROOF;
          COUNT(X, A, LOW, LOW-1) = 0 BY FUNCTION,COUNT(X, A, LOW, LOW-1);
          0<=0;
       QED;

       /# LOOP INVARIANT HOLDS AT LOW #/
       MODE_TO(LOW) & 0<=L<=LOW-LOW;  */

  /* ************** INDUCTIVE CASE ************************/
  /* ASSUME WE HAVE THE MODE UP TO I, THEN FIND THE      */
  /* MODE UP TO I+1 BY LOOKING FOR THE LONGEST BLOCK     */
  /* USING THE LOOP:                                     */
  /*           DO I = LOW TO HIGH                        */
  /*             IF A(I-L)=A(I)                          */
  /*             THEN DO; MODE=A(I); L=L+1; END;         */
  /*           END;                                      */

  /*  PROVE LOOP EXECUTES AT LEAST ONCE                  */
  /*/ ~(HIGH<LOW) BY ARITH,LOW<=HIGH;                    */
  LOOP: DO I = LBOUND(A,1) TO HBOUND(A,1) BY 1;
        /* THE LOOP INVARIANT IS THAT THE MODE OF        */
        /* A UP TO I IS IN MODE.                         */
        /* WE FIND THE MODE UP TO I+1.                   */
        /*/ ASSUME MODE_TO(I) & 0<=L<=I-LOW;             */

        /* DOM(A, I) IS A CONSEQUENCE OF DO-INDEX RANGE  */
        /* PROVE DOM(A, I-L)                             */
        /*/ LOW<=I-L BY ARITH, L<=I-LOW, +, LOW-L=LOW-L; */
        /*/ I-L<=HIGH BY ARITH, I<=HIGH, -, L>=0;        */
```

```
      /*/ ATTAIN MODE_TO(I+1) & 0<=L<=(I+1)-LOW;        */
      IF A(I-L)=A(I)
      THEN
         DO;
         /*/ EXTEND_BOUND:
                ALL X FIXED . COUNT(X, A, LOW, I) <= L+1 BY INTRO,
                PROOF;
                  A1: COUNT(X, A, LOW, I-1)<=L BY ALLEL, MODE_BOUND(I-1), X;
                  A2: COUNT(X, A, LOW, I) <= COUNT(X, A, LOW, I-1)+1
                         BY ALLEL, EXTEND_COUNT, X, A, I;
                  COUNT(X, A, LOW, I) <= L+1 BY ARITH, A1, A2;
                QED;

         B1: COUNT(A(I), A,LOW,I)<=L+1 BY ALLEL, EXTEND_BOUND,A(I);
         B2: COUNT(A(I),A,LOW,I)>=L+1 BY ALLEL,BLOCK_LOWER,A,I,L;
                COUNT(A(I),A,LOW,I)=L+1 BY ARITH,B1,B2;
         */

         /* A(I) IS THE MODE UP TO I+1.    */
         /* L+1 IS THE LENGTH OF THE MODE BLOCK UP TO I+1 */

         /*/ 0 <= L+1 BY ARITH, 0<=L;                     */
         /*/ L+1 <= (I+1)-LOW BY ARITH, L<=I-LOW;         */

         MODE = A(I);
         L = L+1;

         /*  THE INVARIANT HOLDS AT I+1                   */
         /*/ MODE_TO(I+1);                                */
         /*/ 0 <= L <= (I+1)-LOW;                         */
         END;
      ELSE
         DO;
         /* NO MORE THAN L OCCURRENCES OF A(I) UP TO I+1  */
         /* THERE ARE L OCCURRENCES OF MODE, SO A(I) DOES */
         /* NOT REPLACE MODE AND L DOES NOT CHANGE.       */
         /*/ A(I-L)~=A(I); */
         /*/
                EXTEND_BOUND:
                ALL X FIXED . COUNT(X,A,LOW,I)<=L BY INTRO,
                PROOF;
                  DISJ: X~=A(I) | X=A(I) BY ARITH;
                  COUNT(X,A,LOW,I)<=L BY CASES,DISJ,
                PROOF;
                CASE X~=A(I);
                   COUNT(X,A,LOW,I)=COUNT(X,A,LOW,I-1)
                     BY FUNCTION,COUNT(X,A,LOW,I);
                   COUNT(X,A,LOW,I-1)<=L BY ALLEL,MODE_BOUND(I-1),X;
                   COUNT(X,A,LOW,I)<=L;
                  CASE X=A(I);
                   COUNT(A(I),A,LOW,I)<=L BY ALLEL, BLOCK_UPPER, A,I,L;
                  QED;
                QED;

                C1: COUNT(MODE, A, LOW, I-1) = L; /# FROM INVARIANT #/
                C2: COUNT(MODE, A, LOW, I) >= COUNT(MODE, A, LOW, I-1)
                       BY ALLEL, EXTEND_COUNT, MODE, A, I;
                C3: COUNT(MODE, A, LOW, I) <= L BY ALLEL, EXTEND_BOUND, MODE;
                COUNT(MODE, A, LOW, I) = L BY ARITH, C1, C2, C3;

                /# THE INVARIANT HOLDS AT I+1 #/
                MODE_TO(I+1);

                L <= (I+1)-LOW BY ARITH, L<=I-LOW;
                0 <= L <= (I+1)-LOW;
          */
         END;
END LOOP;
END FIND_MODE;
```

3. The Proof Checker



3.1 Local Context Checker

Above is a diagram of the PL/CV Proof Checker. The asserted program written by the user is processed by the Local Context Checker (LCC). The LCC is responsible for checking all explicitly written applications of rules, producing error messages when rules are incorrectly applied. Most of the arithmetic reasoning in PL/CV consists of applications of a powerful rule called ARITH. Hence, a large module of the LCC is a checker of arithmetic inferences. The output of the LCC is known as V-code. V-code is essentially a condensation of the proof, containing only the portion dealing with accessibility and automatic rules. It is checked by the V-code interpreter.

To illustrate the interaction of the LCC and the V-code interpreter, consider the following line from an asserted program:

```
A+B < C+D BY ARITH, A<C, +, B<=D;
```

The LCC checks that A+B < C+D is indeed a consequence of A<C and B<=D. It then generates the following V-code:

```
REQUIRE     A<C
REQUIRE     B<=D
PROCLAIM    A+B < C+D
```

When the V-code interpreter encounters the above instructions, it checks to see that previous instructions have guaranteed that A<C and B<=D, and informs succeeding instructions that A+B < C+D.

3.2 The Assertion Table

The primary data structure of the V-code interpreter is the Assertion Table (AT). At each point in the processing of the V-code, the AT contains a collection S of all assertions which have been proved. Each V-code instruction specifies some kind of manipulation on the AT. The V-code instructions, and the actions they specify are:

Instruction	Action
PROCLAIM P	Add P to S.
REQUIRE P	Test to see that P is in S, or is deducible, using automatic rules, from the assertions in S. If not, print an error message.
NEW V	Create a new variable, V', and replace every occurence of V in S by V'.
OPEN	Make a copy of S, pushing it onto a stack.
CLOSE	Restore S from the top of the stack, and pop the stack.

The instructions are not all implemented as described above, but the implementation behaves as though they were.

3.3 V-code Generation

The LCC is very similar to a compiler, the difference being that V-code, not object code is produced. As an example of the type of processing performed, here is a description of the V-code generated for the DO-WHILE rule given in section 2.3. Lines (1 to 3) translate to REQUIREs. Line (4) generates an OPEN instruction, with the corresponding CLOSE at line (13). These instructions guarantee that assertions proved in the body of the loop will not be used by proof statements after the loop, which would be a violation of accessibility restrictions. Lines (5 to 6) translate into PROCLAIMs of T and P. The proof statements at line (7) are

parsed, and appropriate V-code is generated. These statements may be needed if the proof of line (8), which is translated into a REQUIRE, involves non-automatic rules. At line (9), the body of the loop is parsed, and corresponding V-code is generated. Lines (11 to 12) result in REQUIREs, and line (14) results in a PROCLAIM.

Not mentioned above are the NEW instructions that are also generated. A NEW instruction is generated by any statement that can change the value of a variable. For example, in the following intended proof:

```
(1)  IF (X > 10)
(2)  THEN DO;
(3)  /*/ X>5 BY ARITH, X>10; */
(4)       X = -10;
(5)       /*/ X ~= -10 BY ARITH, X>5; */
```

The assertion on line (5) is incorrect, because X is set to -10 by the assignment statement at line (4). Line (3) will cause X>5 to be PROCLAIMed, but the REQUIRE X>5 generated by line (5) will fail, because the NEW X generated by line (4) will cause the occurence of X>5 in the assertion table to be changed into X'>5.

In the DO-WHILE rule, a NEW instruction for every variable that could possibly be modified by the body of the loop must be inserted just before the OPEN instruction generated by line (4). The NEW instruction is needed to prevent assertions about the state of variables before the execution of the loop from being referenced by proof statements in or after the body of the loop, since those assertions are invalidated by an execution of the body of the loop. Such references would be violations of accessibility restrictions. Were it not for these NEW instructions (and some forward-referencing problems with procedures and functions), the V-code could be interpreted as it is generated. Since the NEWs cannot be inserted until after the entire loop body has been seen, PL/CV has been implemented as a two-pass system.

3.4 Particulars

PL/CV has been implemented on an IBM 370/168 using the CMS operating system. PL/CV contains a large subset of PL/I, including assignment, if, do-while, do-until, do-index, forward goto, return, select, and leave statements, multidimensional arrays, recursive functions and procedures, and external variables. The proof checker was written in PL/I, and contains about thirteen thousand lines. For a typical three-page proof, the program uses about 550K bytes of storage, of which about 100K is data area. Asserted programs are usually checked at a speed of about twenty lines per second, although on some examples it has run as slowly as five lines per second. Some of this sluggishness is due to the high overhead encountered when using PL/I on the IBM 370.

4. Conclusions

4.1 Benefits of a Unified Approach

One advantage of the PL/CV logic is that it presents reasoning about programs as a uniform extension of ordinary mathematical reasoning. For example, the approach reveals that the rules for passing parameters to procedures are simply special cases of substitution for free and bound variables. The rule of substitution for free variables requires that a term t be free for the variable it is replacing. Thus, in the existentially quantified formula $\exists$ Y . (Y > X), the term Z+2 is free for X, so $\exists$ Y . (Y > Z+2) is valid, but the term Y+2 is not free for X because its meaning changes upon substitution. That is, in the formula $\exists$ Y . (Y > Y+2), the quantifier has captured a Y.

Similarly, consider a procedure with an EXTERNAL (global) variable:

```
ASSIGN: PROCEDURE(Y);
DECLARE Y FIXED;
DECLARE X FIXED EXTERNAL;
/*/ ASSUME Y > 0; */
/*/ ATTAIN X = Y - 1; */
X = Y - 1;
RETURN;
END ASSIGN;
```

the variable X is a bound variable, bound by the assignment X = Y - 1. Thus, the substitution ASSIGN(X) causes an illegal capture which explains the absurd conclusion X = X - 1.

4.2 Lessons from Experience

We have been using PL/CV in number theory and the predicate calculus since April 1978. We have verified dozens of small (under 50 lines) examples and several moderate size (100 to 200 lines) examples. Since July 1978, we have been checking programming arguments as well, including large examples (over 300 lines). We have developed a substantial piece of algorithmic number theory including the fundamental theorem of arithmetic.

The formalization of elementary number theory and algorithmic number theory is remarkably direct and appealing. Formal proofs are congruent to their informal counterparts, and only four to six times longer. Although we have discovered ways to significantly condense arguments by providing higher level rules, such proposed improvements are not overwhelmingly necessary or seductive. For example, special cases of the following general principle frequently arise:

```
(ALL X WHERE 1<=X<=I . P(X)) & P(I+1) =>
(ALL X WHERE 1<=X<=I+1 . P(X))
```

A general rule of this form would be convenient, but would save only about six lines of proof per application.

We have also discovered common proof schemes that could be made into seperate new rules, just as the DO I = 1 TO N loop is a seperate form of the DO WHILE loop. But in elementary number theory, these improvements are not necessary, and we expect most of them to become special cases of a systematic extension of the system to non-elementary reasoning, e. g. to higher order predicate calculus and metamathematical reasoning.

In contrast to number theory, the current system is unwieldy when it must deal with programs that extensively modify arrays while maintaining complex quantified invariants about them. The difficulty is that invariants about every part of the array must be reproved whenever any part of the array is modified. In his thesis research, Scott Johnson has discovered a much more useable array rule that overcomes the difficulties so far uncovered. We will incorporate the new array rule in the next version of the Proof Checker because, unlike the other improvements mentioned, this one is essential to manage many ordinary programming problems.

Experience with PL/CV has demonstrated the extent to which even the simplest arguments about programs rely on an extensive inventory of previously proved facts. Early work has required formalization of a large number of "obvious" lemmas. We are learning a great deal about the problem of providing appropriate mechanisms (scope rules, collections of theorems, etc.) for using the information from these theories. By critically examining a large number of arguments, we expect to discover which non-elementary principles of reasoning will be most beneficial. Continuing use of the implemented system and attempts to formalize non-elementary arguments are both essential in learning how to provide a more powerful programming logic. The current system is evidence that this discovery process succeeds; PL/CV2 is a vast improvement over PL/CV1. PL/CV2 is not only a complex research tool, but it is also a valuable instructional aid. If

we succeed to the same extent with non-elementary arguments, we will produce a valuable programming aid. In approaching such arguments, we expect to learn a great deal from AUTOMATH and LCF, which we can combine with our own unique experience.

ACKNOWLEGEMENTS

We would like to acknowledge the excellent and invaluable assistance of Tat-Hung Chan, Dean B. Krafft, and Daniel R. Zlatin in implementing PL/CV2. Michael J. O'Donnell is a principal designer of PL/CV1 and the logic on which PL/CV2 is based [4]. Moreover, he and Tat-Hung Chan are responsible for the elegant arithmetic package. We also appreciate the advice offered by Corky Cartwright, Carl Hauser, and Gary Levin.

REFERENCES

1. Boyer, R. S. and Moore, J. S. Proving theorems about LISP functions, JACM, Vol. 22, 1, January 1975, 129-144.

2. Cartwright, Robert S. User defined data types as an aid to verifying Lisp programs, Proceedings of Third International Conference on Automata, Programming Languages, Edinburgh Press, Edinburgh, 1976, 228-256.

3. Chan, Tat-Hung. An Algorithm for Checking PL/CV Arithmetic Inferences, Technical Report 77-326, Computer Science Dept., Cornell University, 1977.

4. Constable, R. L. and M. J. O'Donnell. A Programming Logic Winthrop, Cambridge, 1978, 389pp.

5. Conway, Richard W. A Primer on Disciplined Programming Using PL/I, PL/CS, and PL/CT, Winthrop, Cambridge, 1978, 419pp.

6. Conway, Richard, D. Gries. An Introduction To Programming, A Structured Approach Using PL/I and PL/C-7, Second Edition, Winthrop, Cambridge, Mass., 1975, 509 pg.

7. deBruijn, N. G. The Mathematical Language AUTOMATH, its usage and, some of its extensions, Symposium on Automatic Demonstration Lecture Notes in Math, Vol. 125, Springer-Verlag, NY, 29-61, 1970.

8. Dijkstra, Edsger W. A Discipline of Programming, Prentice-Hall, Englewood Cliffs, 1976, 217 pg.

9. Downey, Peter J and Ravi Sethi. Variations on the common subexpression problem JACM, to appear.

10. Floyd, Robert W. Assigning meaning to programs, Proceedings of Symposia in Applied Mathematics, Vol. XIX, American Mathematics Society, Providence, 1967, 19-32.

11. Hoare, C.A.R. An Axiomatic Basis for Computer Programming, Comm. ACM 12 October 1969, 576-581.

12. Hoare, C.A.R. and Niklaus Wirth. An Axiomatic Definition of the Programming Language Pascal, Acta Informatica, 2, 1973, 333-355.

13. Igarasshi, Shigeru, R. L. London, D.C. Luckham. Automatic Program Verification I: A Logical Basis and is Implemenation, Acta Informatica, 4, 1975, 145-182.

14. Kleene, S. C. Introduction to Metamathematics, D. Van Nostrand, Princeton, 1952, 550 pg.

15. Kozen, Dexter C. Complexity of Finitely Presented Algebras, Ph.D. Thesis, Cornell University, Computer Science Dept., 1977.

16. Krafft, Dean B. The Assertion Table System for the PL/CV2 Program Verifier, Technical Report 78-337, Computer Science Dept., Cornell University, 1978.

17. Luckham, David C. Program Verification and Verification Oriented Programming, Information Processing 77, B. Gilchrist [Ed.], IFIP, North-Holland, 1977, 783-794.

18. Manna, Zohar. Mathematical Theory of Computation, McGraw Hill, New York, 1974, 448p.

19. Manna, Z. and R. J. Waldinger. Studies in Automatic Programming Logic, North-Holland, Amsterdam, 1977.

20. McCarthy, John. Computer Programs for Checking Mathematical Proofs, Proceedings of Symposia in Pure Mathematics, AMS, Providence, 1962, 219-228.

21. Milner, Robin. Implementation and applications of Scott's Logic for

Computable Functions, Proceedings of ACM Conference on Proving Assertions about Programs, Las Cruces, New Mexico, 1972, 1-6.

22. Milner, R., F. L. Morris, and M. A. Newney. A logic for computable functions with reflexive and polymorphic types, LCF Report No. 1, Department of Computer Science, University of Edinghburgh, January 1975.

23. Nelson, C.G. and D. C. Oppen. A simplifier based on efficient decision algorithms, Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, ACM, NY, 1978, 141-150.

24. Pratt, Vaughan R. Semantical Consideration on Floyd-Hoare Logic, Proceedings of 17th Annual Symposium on Foundations of Computer Science, Houston, October 1976, 109-121.

25. Prawitz, D. Natural Deduction Almquist & Wiksell, Stockholm, 1965.

26. Schwartz, J. T. Correct-Program Technology, Courant Computer Science Report #12, Courant Institute of Mathematical Sciences, New York University, September 1977.

27. Scott, Dana. Data Types as Lattices, SIAM Journal on Computing, 5, 3, September 1976.

28. Wegbreit, Ben. Constructive Methods in Program Verification, IEEE Transactions on Software Engineering, SE-3, e, May 1977, 193-209.

29. Weyhrauch, Richard W. A Users Manual for FOL, Report No. STAN-CS-77-432, Computer Science Dept., Stanford University, July 1977, 68 pg.