Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

135

R.L. Constable
S.D. Johnson
C.D. Eichenlaub

An Introduction to the PL/CV2 Programming Logic

Motivation

One common use for procedures is to take the values of some parameters and return in another parameter a function of these values. For instance, we can write a procedure that returns in Z the sum of the squares of X and Y. It would have the following specification:

```
SUM_OF_SQUARES: PROCEDURE(X,Y,Z);
      DCL (X,Y) FIXED /*/ READONLY */;
      DCL Z FIXED;
  /*/ ASSUME '1'B;
      ATTAIN Z = X*X + Y*Y; */
```

Usually the parameter used for returning the function value (Z in this case) serves no other purpose than to communicate the value to the calling procedure. This is done so frequently that a special form has been provided which eliminates the need for the "storage" variable, thereby removing superfluous details from the calling procedure and its proof. A defined function is a value-returning procedure. That is, it takes all of its parameters readonly, and may be used by name as an operator to build up expressions (either in the logic or in program statements). As an operator, it gives as a value a function of its arguments which is defined by the ATTAIN statement. Here is how we would make SUM_OF_SQUARES a function:

```
SUM_OF_SQUARES: PROCEDURE(X,Y) RETURNS(FIXED);
      DCL (X,Y) FIXED;
  /*/ ASSUME '1'B;
      ATTAIN SUM_OF_SQUARES(X,Y) = X*X + Y*Y; */
```

The two differences from an ordinary procedure specification are that the type of value returned is given, and the procedure name itself is used in the ATTAIN. Note that all parameters and external variables are assumed to be READONLY and need not be explicitly designated as such.

In the procedure version of SUM_OF_SQUARES we would assign the desired value to Z and return. In the function version, we include the value as part of the return statement. The above specification in fact, could have as a program the single statement:

```
RETURN(X*X + Y*Y);
```

A procedure which called the SUM_OF_SQUARES as a function instead of a procedure could replace the two statements:

```
CALL SUM_OF_SQUARES(A,B,C);
FOO = 2*C - 1
```

with just

```
FOO = 2*SUM_OF_SQUARES(A,B) - 1;
```

As with procedures, we have proof rules for both the definition and use of functions.

Rules For Using Functions

Defined functions are used exactly the same way that the special functions like ABS and MOD are. MOD(I,J), for instance, is a function with J~=0 as its ASSUME. Its ATTAIN is the conjunction of the assertions given in figure 2 (sec. III). Reviewing briefly, there are three ways to use functions:

(1) A function name may appear in a program statement as part of an expression to be evaluated. The ASSUME for the function must be proven at that point for the arguments used.

(2) A function application may appear in assertions wherever it could be replaced by an arbitrary expression of the same type. For example, given a FIXED function F(X,Y), the following may be written anywhere:

```
F(1,2) = F(1,2);        ALL (W,Z) FIXED. F(W,Z) > 2
                                  => F(W,Z) > 1 BY INTRO, INTRO,
                           PROOF;
                            F(W,Z) > 1 BY ARITH, F(W,Z) > 2;
                            QED;
```

See section 4.7 for more on the PL/CV semantics of function applications.

(3) The ATTAIN of a function (for given arguments) may be asserted anywhere that its ASSUME (for the same arguments) is true. The justification is BY FUNCTION, followed by the function name and arguments. For instance, we may write

```
0 <= MOD(I,J) < ABS(J)  BY FUNCTION, MOD(I,J);
```

wherever J~=0 is accessible. Similarly, we may assert

```
SUM_OF_SQUARES(W,Z) = W*W + Z*Z BY FUNCTION, SUM_OF_SQUARES(W,Z);
```

anywhere, since the ASSUME for this function ('1'B) is always true.

As the MOD example shows, the ATTAIN of a function may do more than simply state what the value of the function is. Any assertion using the function name and/or parameters may appear in the ATTAIN. Note that, unlike procedures, functions may be used via the FUNCTION rule in strictly logical theorems. This can often be an effective means for providing theorems with access to algorithmic notions of construction.

(Functions that take BIT(*) parameters may be applied to boolean expressions made up of logical variables, but not to assertions. This restriction is to keep PL/CV a constructive logic.)

Verifying Functions

There are three differences between the proof rules for functions and for procedures. The first is a restriction: functions return a value and do nothing else. They may not have any side effects. A side effect is defined as changing the value of any variable other than the functions own local variables. We have alreay mentioned that the parameters and EXTERNAL variables of a function are assumed to be readonly. In addition, the function may not make any procedure calls which will produce side effects. (I.e. only the function's local variables may be read-write arguments to a procedure, and no procedure called may assign to EXTERNAL variables.) The motivation for this restriction comes from our desire to use function names as operators. If they were allowed to change their arguments, or otherwise alter the environment they were evaluated in, the logic for making correct assertions would become vastly more complicated.

The second difference is in the rule for RETURN statements. The basic idea is still the same: the ATTAIN must be proven at any point where the function returns. We simply have to modify what this means to account for the fact that a value is being returned. Let F be a function of one argument, X. In general, its ATTAIN condition will contain occurrences of F(X) to describe the value returned. When a RETURN(exp) statement is encountered during execution, the value of exp will be the value that is returned for the function. Therefore, the ATTAIN must be proven with all occurrences of F(X) replaced by exp. This must be done for each RETURN statement in the function.

Finally, we must recognize a subtle form of recursion that can occur in function definitions. Any time an assertion is justified BY FUNCTION, F(exp), it must be the case that evaluation of F(exp) is possible at that point. What if an assertion in the body of F is justified this way? This will be correct only if the call is a proper one, but clearly it will be a recursive call. The rules for termination of recursive function applications are the same as for procedure calls. In particular, the termination predicate for F(exp) must be satisfied by N-1 before any occurrence of F(exp) in a program statement or any assertion justified BY FUNCTION, F(exp). Note that this means that two functions may be mutually recursive just by mentioning each other in their proofs, without actually executing an expression that contains the other. (Note: it is not permitted for a function and a procedure to be mutually recursive.)

The ability to recursively refer to functions in their own proofs can be very useful. A good example is the function FACT(X), which returns the factorial of X as a value. There is no easy way to express the relation of FACT(X) to X in our assertion language without using recursion. Note that saying that every number less than or equal to X divides FACT(X) is not enough, since other numbers besides the factorial of X have that property. The most elegent ATTAIN for the function FACT(X) is

$$X=0 \Rightarrow FACT(X)=1 \quad \& \quad X > 0 \Rightarrow FACT(X) = X * FACT(X-1)$$

This ATTAIN can actually be proven for a function body which proceeds in the straightforward iterative manner, multiplying together all the numbers less than X and returning the answer. The proof however, contains two uses

of the function rule on FACT itself, both times with arguments which are smaller than X. (See the last example below.) It is probably less complex than anything that could be done using a non-recursive ATTAIN. Moreover, once this description of FACT is shown, other desirable properties of the function may be proven by induction (for example, the fact that FACT(X) is divisible by all numbers less than or equal to X).

Examples

Here is the proof for the built-in function ABS, which returns the absolute value of its argument:

```
ABS: PROCEDURE(X) RETURNS(FIXED);
DECLARE X FIXED /*: READONLY */;
/*/ ASSUME '1'B;                                    */
/*/ ATTAIN X >= 0 => ABS(X) = X,                    */
/*/        X <= 0 => ABS(X) = -X,                   */
/*/        X  = 0 => ABS(X) = 0;                    */

/*/ ATTAIN '0'B;                                    */
SELECT;

WHEN(X > 0)
    DO;
    /*/ X >= 0 BY ARITH, X > 0;                      */
    /*/ ~(X <= 0) BY ARITH, X > 0;                   */
    /*/ ~(X = 0) BY ARITH, X > 0;                    */
    RETURN(X);
    END;

WHEN(X < 0)
    DO;
    /*/ ~(X >= 0) BY ARITH, X < 0;                   */
    /*/ ~(X = 0) BY ARITH, X < 0;                    */
    /*/ X <= 0 BY ARITH, X < 0;                      */
    RETURN(-X);
    END;

OTHERWISE
    DO;
    /*/ X = 0 BY ARITH, ~(X< 0), ~(X > 0);           */
    /*/ X >= 0 BY ARITH, X = 0;                      */
    /*/ X <= 0 BY ARITH, X = 0;                      */
    /*/ -X = 0 BY ARITH, X = 0, *, -1 < 0;           */
    RETURN(X);
    END;
END;
END ABS;
```

The built-in function EXP contains a recursive call to itself in the procedure body. (It returns as a value B raised to the power E.)

```
EXP: PROCEDURE (B, E) RETURNS(FIXED);
DECLARE (B, E) FIXED /*: READONLY */;
```

```
/*/ ASSUME B ~= 0 & E >= 0;                                    */
/*/ ARBITRARY D FIXED WHERE D > E;                              */
/*/ ATTAIN E = 0 => EXP(B, E) = 1,                               */
/*/        E >= 1 => ((EXP(B, E) = B * EXP(B, E - 1)) & */
/*/                   (EXP(B, E) / B = EXP(B, E - 1))); */

/*/ ~(0 > E) BY ARITH, E >= 0;                                  */

/*/ ATTAIN '0'B;                                               */
IF E = 0
THEN DO;
     /*/ ~(E >= 1) BY ARITH, E = 0;                            */
     RETURN(1);
     END;
ELSE DO;
    /*/ E >= 1 BY ARITH, E >= 0, ~(E = 0);                    */
    /*/ D - 1 > E - 1;                                        */
    /*/ E - 1 >= 0 BY ARITH, E >= 1;                          */
    /*/ B <= 0 | B >= 0 BY ARITH;                              */
    /*/ ABS(B) > 0 BY CASES, B <= 0 | B >= 0,                  */
    /*/ PROOF;                                                 */
    /*/ CASE B <= 0;                                          */
    /*/      ABS(B) = -B BY FUNCTION, ABS(B);                  */
    /*/      -B > 0 BY ARITH, B <= 0, B ~= 0;                  */
    /*/ CASE B >= 0;                                          */
    /*/      ABS(B) = B BY FUNCTION, ABS(B);                   */
    /*/      B > 0 BY ARITH, B >= 0, B ~= 0;                   */
    /*/ QED;                                                   */
    /*/ B * EXP(B,E-1) = EXP(B,E-1)*B + 0;                     */
    /*/ MOD_RESULT: ALL (Q, R) FIXED .                         */
    /*/      ((B*EXP(B,E-1) = B*Q+R & 0<=R<ABS(B*EXP(B,E-1))) */
    /*/         => R = MOD(B*EXP(B,E-1), B))                   */
    /*/     BY FUNCTION, MOD(B*EXP(B,E-1), B);                */
     RETURN(B * EXP(B, E-1));
     END;
END EXP;
```

Here is a proof of the factorial function, which was given as an example of a function with recursive reference to itself in an assertion.

```
FACT: PROCEDURE(X) RETURNS(FIXED);
   DECLARE X FIXED;
   /*/ ASSUME X>=0;
       ARBITRARY N FIXED WHERE X < N;
       ATTAIN (X=0 => FACT(X)=1) & (X>0 => FACT(X)=X*FACT(X-1)); */
   DECLARE (PROD,I) FIXED;
       /*/  ~(X < 0) BY ARITH, X>=0; */

         PROD = 1;

      /*/ 1>X => ( (X>0 => PROD=X*FACT(X+1-2)) &
                 (X=0 => PROD = 1)) BY INTRO,
            PROOF;
```

```
        ~(X>0) BY ARITH, 1>X;
        PROD = 1;
         QED;     */
  /*/ (X+1)-1 = X BY ARITH; */
  /*/ ~(0>0) BY ARITH; */

      DO I = 1 TO X BY 1;
     /*/ ASSUME (I-1=0 => PROD=1) & (I-1>0 => PROD=(I-1)*FACT(I-2)); */
     /*/ X>=1 BY ARITH, X>=I>=1; */
     /*/ I-1=0 | I-1>0 BY ARITH, I>=1,-,1=1; */
     /*/ PROD*I = I*FACT(I-1) BY CASES, I-1=0 | I-1>0,
           PROOF;
          CASE I-1=0;
            0>=0 BY ARITH;
            1 < N BY ARITH, X<N, X>=1;
            0 < N-1 BY ARITH, 1<N, -, 1=1;
            FACT(0)=1 BY FUNCTION, FACT(0);
            I=1 BY ARITH, I-1=0, +, 1=1;
          CASE I-1>0;
            I-1 >= 0 BY ARITH, I-1>0;
            I < N BY ARITH, X<N, X>=I;
            I-1 < N-1 BY ARITH, I<N, -, 1=1;
            FACT(I-1)=(I-1)*FACT(I-2) BY FUNCTION, FACT(I-1);
            I>0 BY ARITH, I-1>0;
            PROD*I = I*FACT(I-1) BY ARITH, PROD=FACT(I-1), *, I>0;
          QED;  */

       PROD = PROD*I;

      /*/ ~(I=0) BY ARITH, I>=1; */
      /*/ (I+1)-1 = I BY ARITH;
          (I+1)-2 = I-1 BY ARITH;  */
      /*/ (I=0 => PROD=1) & (I>0 => PROD=I*FACT(I-1)); */

      END;

   /*/ (X=0 => PROD=1) & (X>0 => PROD=X*FACT(X-1)); */

   RETURN(PROD);
END;
```

X FUNCTION RULES

10.1 Introduction

Rules for introducing recursive and nonrecursive function definitions are nearly identical to the corresponding rules for recursive and nonrecursive procedures. However, since functions play an entirely different role in the logic than do assignments and their generalization to procedures, the rules for using defined functions have an entirely different character than the rules for using procedures. The first difference is that functions affect the most basic level of the logic, the predicate calculus. Secondly, all arguments to functions (including external variables) are readonly, so the intricate problem of substitution for bound variables does not arise. Thirdly, defined functions occur in every type of expression, so their use complicates every rule which mentions expressions, that is every rule except the goto and label rules.

10.2 Syntax

```
            f: PROCEDURE(parameter_list) RETURNS(type) [RECURSIVE];
               {declaration, define_statement}*
               ASSUME assertion;
               [ARBITRARY variable FIXED WHERE assertion;]
   heading     ATTAIN assertion;
               proof_stmt*
               declaration*
               argument
               END f;
```

Note 1: f is any label. It is the function name. The attain statement may involve f itself

Note 2: A function in PL/CS may not contain input/output statements, or calls to procedures which may modify either EXTERNAL variables or the function's parameters. This is to guarantee that function evaluations produce no side effects.

Note 3: External variables and all parameters are readonly.

10.3 Proof Rules

In order to introduce a set of mutually recursive function definitions, we must know that each halts on its assumed domain. This is demonstrated almost exactly as in the case of procedures.

RECURSIVE FUNCTION INTRODUCTION

```
f:  PROCEDURE(v̄);
    v̄ READONLY
    x̄ EXTERNAL
    ASSUME IN_f(v̄,x̄);
    ARB d FIXED WHERE T_f(d,v̄,x̄);
    ATTAIN OUT_f(v̄,x̄, f(v̄));
    ~T_f(0,v̄,x̄);
        .
        .
        .
    T_g(d-1, ū,x̄), IN_g(ū,x̄)   OUT_f(v̄,x̄,exp)  before
    before any occurrence       each occurrence of
    of g(ū) for any g in              RETURN(exp)
    the set of mutually
    recursive definitions
        .
        .
        .
    '0'B
END f;
```

Note, not all externals need be mentioned in IN_f, IN_g.

A function definition in PL/CV comprises an algorithm and a lemma about the defined function. This lemma has the form

$$IN_f(\bar{v},\bar{x}) \;\&\; T_f(d,\bar{v},\bar{x}) \Rightarrow OUT_f(\bar{v},\bar{x},f(\bar{v}))$$

where $\bar{v}$ can be considered universally quantified (and $\bar{x}$ are parameters of definition). The lemma is proved by a form of induction, so we call it the recursive function induction rule. The invocation rule is simply called function lemma invocation.

FUNCTION LEMMA INVOCATION

```
f: PROCEDURE(v̄) RETURNS(atype);
   v̄ READONLY;
   x̄ READONLY EXTERNAL;
   ASSUME IN_f(v̄,x̄);
   ARBITRARY d FIXED WHERE T_f(d,v̄,x̄);
   ATTAIN OUT_f(v̄,x̄,f(v̄));
       .                          SOME m FIXED .m>=0&T_f(m̄,ū,x̄);
       .
   END f                          IN_f(ū/v̄,x̄)
   ------------------------------------------------------------
          OUT_f(ū,x̄,f(ū))    BY FUNCTION, f(ū);
```

For completeness we also need the rule

RETURN RULE

$$\frac{\text{RETURN(exp)}}{A}$$