On Subprograms With A Variable Number Of Parameters Of Varying Types

Xu Baowen

Department of Computer Science and Engineering
Nanjing Aeronautical Institute
Nanjing, Jiangsu, P. R. China

Abstract

Subprograms are the fundamental building blocks of programs, and are therefore among the important concepts in programming language design. The effect and function of programs is affected by their parameter mechanisms. This paper describes two facilities: number-variable parameters and type-variable parameters. These two parameter mechanisms do not decrease the efficiency and reliability of a subprogram, and are very useful in many software development, especially in general software package development.

1. Introduction

In programming, we often need to write the subprograms which may have *number-variable parameters* or *type-variable parameters*. For example, we may write a function used to compute the sum of all integer parameters, or a procedure used to swap the values of two variables of any type, etc. In Pascal language, the predefined procedures READ and WRITE and the predefined functions PRED and SUCC may accept certain predefined types of actual parameters. Nearly none of programming languages provides subprograms with the parameter mechanisms. In Pascal, we cannot use the language itself to write the headings of the predefined subprograms with the properties; In C language, functions may have variable number of parameters, but the problem of type compatibility is left to the programmers and sacrifices the reliability (C is not a strongly typed language); in Ada language, although the number of parameters may vary by the default parameter mechanism, but the variation is limited; the types of parameter may also vary by the generic mechanism, but the generic subprogram must be instantiated before calling.

For these reasons, we developed an extended Pascal language, Pascal/N, which extend the parameter mechanism and type mechanism. In Pascal/N, the number and types of subprogram parameters both are variable, and when necessary, some restrictions may be put. The paper will discuss the extension in Pascal/N.

2. Subprograms with variable number of parameters

A formal parameter part plays an important role in a subprogram. On the one hand, the parameter part specifies how to use the parameters in the subprogram body, and on the other hand, it specifies the number, type and association mode of each corresponding actual parameter in the subprogram calls. Therefore, if a subprogram is used with variable number of parameters, the formal parameter part must be redesigned. One of the design goals of Pascal/N is that it may not only handle a subprogram with variable

number of parameters of varying types, but also maintain the properties of static type checking and static consistency checking, that is, the legality checking of association and use of parameters in the subprogram body and calls should be completed at compile-time. Because of this, the following points should be made. First of all, each actual parameter must correspond to one and only one formal parameter, and the correspondence must be determined from the context of the subprogram call. Secondly, there should not be a one-to-one correspondence between actual and formal parameters; because it is required that the number of actual parameters may vary, and that a formal parameter may correspond to zero or more actual parameters, all actual parameters corresponding to the same formal parameter must have the same type and parameter mode with the formal parameter. Lastly, this kind of formal parameters must be distinguished from common formal parameters.

In Pascal/N, A formal parameter may correspond to zero or more actual parameters (or to actual parameters with the number in the specified range) by inserting some appropriate symbols after the formal parameter identifier in the parameter declaration; if a formal parameter is required to correspond to zero or more actual parameters, the sign"(*)" is inserted after the formal parameter identifier. For example, in parameter declaration

TOTAL,INTEGERLIST(*): INTEGER;

TOTAL is a common formal parameter, and must correspond to one and only one actual parameter of INTEGER type in the subprogram invocations; however, INTEGERLIST is a number-variable formal parameter, and in relevant subprogram invocations, it may correspond to zero, one or more actual parameters of the same INTEGER type.

If a formal parameter is required to correspond to one actual parameter at least, the sign"(+)" is inserted after the formal parameter identifier in the formal parameter declaration; if it is required to correspond to m actual parameters at least, the sign"(m. .)"is inserted after the relevant identifier when it is declared; if it is required to correspond to n actual parameters at most, the sign "(. . n)" is inserted after the formal identifier; if a formal parameter is required to correspond to m~n actual parameters, the sign "(m. . n)" is inserted after the formal identifier.

In order to check the consistency between actual and formal parameters, the syntax of a actual parameter part need to be modified in subprogram invocations. In Pascal/N, the actual parameters corresponding to the same formal parameter are separated by a comma, and the two groups of the actual parameters corresponding to the two respective formal parameters are separated by a semicolon.

In subprograms with one-to-one correspondence between the formal parameters and the actual parameters, the addresses or the values of the actual parameters may be accessed by means of the formal parameters. In subprograms in which a formal parameter may correspond to zero or more actual parameters, however, it seems obviously inadequate. In order to access one of the group of actual parameters corresponding to a single number-variable formal parameter, Pascal/N allows an index to be appended after the formal parameter, similar to an array component (subscript variable) in form. The index is the order number of a specified actual parameter in the group of the actual parameters, with the value zero representing a null actual parameter. For example, let FP be a number-variable formal parameter, FP(1) and FP(5) represent the first and the fifth actual parameter of the group of the actual parameters corresponding to the formal parameter respectively in the corresponding subprogram body.

Because the number of the actual parameters corresponding to a number-variable formal parameter is variable, in order to make the access to the actual parameters under effective control in the subprogram body, and to manage to refer each necessary actual parameter and not to refer a nonexistent actual parameter, a new operator, #, is introduced. The operator has only one operand, i. e., the formal parameter, whose value is the number of all the actual parameters corresponding to the formal parameter in

a subprogram invocation. The value is not determined until the relevant invocation has been processed, and may change along with the different invocations.

Loop statements are used to control the access to each actual parameter corresponding to a number-variable formal parameter in a subprogram body. Moreover, if the sign appended to a formal parameter in a formal parameter declaration represents that the number of the corresponding actual parameters is m at most, and if the index in an indexed formal parameter is a constant in the subprogram body, the value of the constant must not be greater than *m*. As a consequence, if the sign appended to a formal parameter in a formal parameter declaration is a asterisk, the formal parameter index may not be constant in the subprogram body, unless it is enclosed in a control statement.

Example 2. 1

Following is a Pascal/N function with its invocation corresponding to the example at page 70 in literature[1]. The function is used to evaluate the maximum of all actual parameters in the related function invocation:

```
function MaxReal(R(+):REAL): REAL;
        var I: INTEGER;
begin
        MaxReal:=R(1);
        for I:=2 to #R do
                if MaxReal<R(I) then MaxReal:=R(I)
end{MaxReal};
... ...
MAX:=MaxReal(1.1, 2.3, 4.0, 100.1, 49.0*45.3, 2100.9, 3214./91.45);
```

Example 2. 2

The following procedure is used to compute the sum of a length-variable integer sequence with the result stored in parameter SUM:

```
procedure SumOfIntegers(I(*):INTEGER; var SUM:INTEGER);
    var Index:INTEGER;
begin
    SUM:=0;
    for Index:=1 to #I do
        SUM:=SUM+I(Index)
end{SumOfInteger};
```

The related procedure invocation statement may be:

```
SumOfIntegers(1,3,5,7,9,11;S1);
SumOfIntegers(I*I,J*J,K*K,L*L;S2);
```

3. Alternative Types and Type-variable Parameters

In order to use subprograms with type-variable parameters, this section introduces a new type concept, *alternative types*. The alternative type declaration is in the form as follows:

```
type AltType=alternative
                    BaseType1
```

```
                BaseType2
                ......
                BaseTypen
            end;
```

where BaseType1, BaseType2, . . . , BaseTypen between the reserved words **alternative** and **end** are called the base types, which are type names (identifiers). When an alternative type is used as the type of formal parameters, the type of corresponding actual parameters must be one of the base types in the alternative type; otherwise, the error about type compatibility will be detected at compile-time. In Subprogram body, when the formal parameters of a alternative type is used, it is necessary to check whether the operations on the parameters are suitable for each base type in the alternative type. As long as there is an unsuitable operation, compiler will show a related compile-time error information.

NUMERICAL, DISCRETE, SCALAR, ANY and FILENAME are five predefined alternative types, which indicate that their base types may be any numerical type, any discrete type, any scalar type, any (non-file) type and any file type respectively. For example, when the type of a formal parameter is a SCALAR, the corresponding actual type may be integer, real, boolean, character or any other enumeration type.

Sometimes the following case may happen; when a subprogram has more than one formal parameters with an alternative type, some corresponding relation may be required between the relevant actual parameters. For example, one may write a procedure to exchange the values of its two parameters with the type SCALAR or ANY;

```
procedure SWAP (var L,R:ANY);
    var TEMP: ANY;
begin
    TEMP:=L; L:=R; R:=TEMP;
end;
```

If analyzing the procedure one would discover that the compiler will show the error of type compatibility when its statement part is compiled. The reason lies in that, because all L, R and TEMP can be of any type, all the three assignment statements are error when the types of L, R and TEMP are not compatible. In addition, is it allowed that the type ANY is used in the variable declaration in the procedure? If it is not allowed, the use scope of an alternative type must be very limited; if it is allowed, however, when the program is executed, what is the base type of the local variable? and what determines the base type by? Obviously, if alternative types can be only used in subprogram parameter declarations, the types are not very useful. However, if alternative types are allowed to be used in subprogram bodies, their base types are beyond determination. For this reason, Pascal/N does not allow this kind of general alternative type to be used in subprogram bodies. In order to solve this problem, we introduce the concepts of corresponding *alternative types*. The corresponding alternative type declaration is as follows;

```
type CorrAltType=corresponding alternative
                    BaseType1,
                    BaseType2,
                    ......
                    BaseTypen
                end
```

Although the only difference in syntax between the correspondig alternative types it and the general alternative types described above is in that there is reserved word **corresponding** (abbreviated to **corr**) in it,

its usage is quite different from the general alternative type; if two formal parameters are declared to be of the same corresponding alternative type in a subprogram parameter part, among the two corresponding actual parameters in the subprogram invocations, when one is of some base type, the other must also be of the base type. Similarly, if a corresponding alternative type, which is used as the type of some formal parameters in a subprogram, is also used in the body, when the actual parameters corresponding to the formal parameters are of a base type (of the corresponding alternative type), the corresponding alternative type is also interpreted as this base type in the body. However, there is one restriction that only the corresponding alternative types used in the formal parameter part can be used in the subprogram body.

The corresponding alternative type can be declared on the basis of some general alternative type. For example,

```
type CAT=corresponding alternative
                BaseType1,
                BaseType2,
                ......
                BaseTypen
            end
```

is equivalent to the following two type declarations;

```
type AT=alternative
                BaseType1,
                BaseType2,
                ......
                BaseTypen
            end;
    CAT=corresponding AT;
```

In order to write the procedure SWAP correctly, one may declare firstly;

```
type CorrAny=corr ANY;
```

and then give the declaration of SWAP procedure;

```
procedure SWAP (var L,R: CorrAny);
    TEMP:=CorrAny;
begin
    TEMP:=L; L:=R; R:=TEMP;
end{SWAP};
```

Because L,R and TEMP have the same (base) type, the assignments are legal in the procedure body.

Pascal/N also provides five predefined corresponding alternative types CNUMERICAL, CDISCRETE, CSCALAR, CANY and CFILENAME, which correspond to the five predefined general alternative types NUMERICAL, DISCRETE, SCALAR, ANY and FILENAME respectively. Their meanings are very clear and not explained here.

A corresponding alternative type just discussed is also called a *homogeneously corresponding alternative type*, because a corresponding alternative type is interpreted as the same base type in the same subprogram. However, other corresponding case is required to be considered in some programs. For example, it is required that, in some program, when parameter A has the type T1 or T2, parameter B must have the types T3 or T4. For this reason, the concept of *heterogeneously corresponding alternative types* is introduced in Pascal/N. This kind of type is declared as follows;

```
type HCAT=corresponding
                AltType1,
```

```
                AltType2,
                ......
                AltTypen
            end
```

where AltType1, AltType2, ..., AltTypen are all the names (identifiers) of general alternative types or the names of homogeneously corresponding alternative types. Names of heterogeneously corresponding alternative types cannot be used alone and can only be used as the prefix of the alternative types in it, e. g. , HCAT. AltType1. Similar to other alternative types, if a heterogeneously corresponding alternative type needs to be used in a subprogram body, it must first be used in the formal parameter declarations of the subprogram, otherwise, the use is illegal. In a subprogram, if a formal parameter declaration such as

PARM: HCAT. AltType1;

is used first in the formal parameter part, and another component alternative type in the heterogeneously corresponding alternative type, such as HCAT. AltType2 or HCAT. AltTypen, is used in the other parameter declarations or the declaration part in the subprogram body, when the type of the actual parameter corresponding to formal parameter PARM is the ith base type of AltType1 in a invocation to the subprogram, using other component alternative type (e. g. , AltType2 or AltTypen) is equivalent to using the ith base type in it. Therefore, the correspondence checking should be made at the compile-time.

Example 3. 1

The following example on stack operations is used to explain the use of heterogeneously corresponding alternative types. For simplicity, the program segment implements only Init, Push and Pop operations, and the other operations may be implemented similarly.

```
type IntStackType = record
                        S:array[1..100]of INTEGER;
                        PTR: 0..100
                    end;
    CharStackType = record
                        S:array[1..100]of CHAR;
                        PTR: 0..100
                    end;
    BoolStackType = record
                        S:array[1..100]of BOOLEAN;
                        PTR: 0..100
                    end;
    ElementType = alternative
                        INTEGER,
                        CHAR,
                        BOOLEAN
                  end;
    StackType = alternative
                    IntStackType,
                    CharStackType,
                    BoolStackType
                end;
```



```
    CorrES = corresponding
                ElementType,
                StackType
             end;

var IntStack: IntStackType;
    I: INTEGER;
    CharStack: CharStackType;
    C: CHAR;
    BoolStack: BoolStackType;
    B1,B2: BOOLEAN;

procedure Init(var Stack: StackType);
begin
    Stack. PTR:=0
end{Init};

procedure Push(Elem: CorrES. ElementType;
               var Stack: CorrES. StackType);
begin
    Stack. PTR:=Stack. PTR+1;
    Stack. S[Stack. PTR]:=Elem
end{Push};

function Pop(var Stack: CorrES. StackType): CorrES. ElementType;
begin
    Pop:=Stack. S[Stack. PTR];
    Stack. PTR:=Stack. PTR-1
end;{Pop};

... ...

Push(15+31;IntStack);
Push(' C' ;CharStack);
Push(B1;BoolStack);
... ...
I:=Pop(IntStack);
C:=Pop(CharStack);
B2:=Pop(BoolStack);
```

4. Examples

The two mechanisms discussed in the two previous sections can be combined to use variable number of parameters of varying types. It needs to be pointed out that the type checking can be made completely at compile-time in this way.

Example 4.1

As described at the beginning of the paper, the headings of the predefined input and output procedures READ and WRITE cannot be written in Pascal itself, but they can be easily written in Pascal/N:

```
procedure READ (var F(0..1):FILETYPE; var R(+):SCALAR);
procedure WRITE (var F(0..1):FILETYPE; W(+):SCALAR);
```

Of course, they are subtly different from standard procedures READ and WRITE in Pascal. For example, the input and the output of enumeration types are not allowed in Pascal, but this can be done with the more restricted alternative type in replacement of SCALAR.

Example 4.2

Following is the revised version of the example on stack operations described in the previous section. In this version, Push procedure can push successively one or more elements on a stack, and Pop procedure can pop one or more elements from a stack. The definitions of all the types are the same to the previous version, and the rest is rewritten as follows:

```
var IntStack: IntStackType;
    I1,I2,I3,I4: INTEGER;
    CharStack: CharStackType;
    C1,C2,C3: CHAR;
    BoolStack: BoolStackType;
    B1,B2: BOOLEAN;

procedure Init(var Stack: StackType);
begin
    Stack.PTR:=0
end{Init};

procedure Push(var Stack: CorrES.StackType;
               Elem(+): CorrES.ElementType);
    var INTEGER;
begin
    for I:=1 to #Elem do
        begin
            Stack.PTR:=Stack.PTR+1;
            Stack.S[Stack.PTR]:=Elem(I)
        end
end{Push};

procedure Pop(var Stack: CorrES.StackType;
              var Elem(+): CorrES.StackType);
    var I: INTEGER;
begin
    for I:=1 to #Elem do
        begin
            Elem(I):=Stack.S[Stack.PTR];
            Stack.PTR:=Stack.PTR-1
        end
end{Pop};

... ...

Push(IntStack; 1,4,7,10,I1,I2);
Push(CharStack; 'N','A','I');
Push(BoolStack; TRUE,B2,FALSE,B1);
... ...
Pop(IntStack; B1,B2,B3);
Pop(CharStack; I2,I4,I4,I3);
Pop(BoolStack; C1,C1,C1,C2,C3);
```


5. Conclusions

C. S. R. Carvalho et al have introduced the open array concept to implement subprograms with a variable number of parameters[1], which can solve the problems of syntax and semantics in subprograms with a variable number of parameters to some extent. However, this approach, which requires to specify the subscript types and bounds of open arrays in formal parameter declarations, seems unclear and poorly intuitive, and restricts parameter declaration of an open array type as the last formal parameter declaration in a subprogram formal parameter part. Other formal parameters except the last one cannot be of open array types. This greatly decreases its flexibility. C language allows functions with a variable number of parameters, but the type compatibility checking is not done at compile-time. The new mechanisms introduced in this paper not only allows real subprograms with a variable number of parameters, but ensures the type compatibility checking at compile-time.

The concept of type-variable parameters is also not first introduced in the paper. In fact, many predefined subprogram may be explained by this mechanism, but must be implemented in pseudo subprograms. The generic mechanism in Ada is similar to the mechanism introduced in the paper, but is more complicated and no more ease to use.

To sum up, the two parameter mechanisms described in this paper are not only effective, practical and convenient to use, but is safe and reliable, and can be implemented effectively.

References

[1] C. S. R. Carvalho et al, On Open Arrays and Variable Number of Parameters, Computer Languages, Vol. 17, No. 1, 1992, pp67-74

[2] US DOD, Reference Manual for Ada Programming Language, MIL/STD 1815A, 1983

[3] Jensen, K. et al, Pascal User Manual and Report, 3rd ed., Springer, 1985

[4] Kernighan, B. W. et al, The C Programming Language, Prentice-Hall, Englewood Cliffs, NJ, 1978