ON OPEN ARRAYS AND VARIABLE NUMBER OF PARAMETERS

Claudio Sergio Da Rós de Carvalho
and
Tomasz Kowaltowski

RELATÓRIO TÉCNICO Nº 25/90

Abstract. Two facilities for programming languages are described: *open arrays* (an extension of Pascal *conformant arrays*) and *automatic parameter conversion*. As a result of combining these two mechanisms, it is possible to give compile–time verifiable specification of procedures with a variable number of parameters and varying types. This ability is very useful in many applications, and in particular in specifing I/O procedures within a programming language itself.

Instituto de Matemática, Estatística e Ciência da Computação
Universidade Estadual de Campinas
IMECC - UNICAMP
Caixa Postal 6065
13.081, Campinas, S.P.
BRASIL

Julho – 1990

On Open Arrays and Variable Number of Parameters

Claudio Sergio Da Rós de Carvalho
Tomasz Kowaltowski

Departamento de Ciência da Computação
Universidade Estadual de Campinas
Caixa Postal 6065
13081 Campinas, SP, Brazil

April 4, 1990

Keywords: programming language design, open arrays, automatic parameter conversion, variable number of parameters

Abstract

Two facilities for programming languages are described: *open arrays* (an extension of Pascal *conformant arrays*) and *automatic parameter conversion*. As a result of combining these two mechanisms, it is possible to give compile-time verifiable specification of procedures with a variable number of parameters and varying types. This ability is very useful in many applications, and in particular in specifing I/O procedures within a programming language itself.

The results described in this paper are part of the first author's M. Sc. dissertation, developed under the supervision of the second author.

1 Introduction

Programming language design seems to be quite a lively activity and "new" languages pop up continuously. Most of these languages tend to have a very restricted usage, but the design activity itself contributes with new ideas which eventually find their way into widely used languages. This work focuses on the area of system programming languages, where two important trends appear. An increasingly large number of systems are being developed using the C language [KernRit 78], and its descendants and dialects. This trend is due to several factors, among them are the connection between C and the UNIX[1] operating system, low-level facilities available in C, and the possibility of writing very efficient code. On the other hand, mainly within the academic and research community, a large number of followers of the Pascal-like tradition exist, whose main representatives are Modula-2 [Wirth 85] and some of its descendants: Oberon [Wirth 88], Modula-3 [Cardelli et al. 88], and so on. This preference is due usually to the clean syntax, strong type checking, clear separate compilation and modular composition facility.

The authors decided to study the feasibility of designing a language which would incorporate the advantages of C and Modula-2. The result is quite satisfactory and described in [Carvalho 89] and [Carvalho 89a]. One of the goals of this design was to allow for the specification of procedures with a variable number of parameters and varying types, through a clean syntax which would make easy compile-time checking. As a consequence, typical I/O routines could be specified within the language itself.

The paper describes the two mechanisms used to achieve this goal: *open arrays* and *automatic parameter conversion*. None of these mechanisms is completely new. Open arrays, with diverse meanings appear in Algol 60 [Naur 63], PL/I [IBM PL/I 88], Algol 68 [Wijngaarden et al. 69], ISO Pascal [JensenWirth 85], Modula-2, C and Ada [DraftAda 83]. Several kinds of automatic conversions appear in PL/I, Algol 68 and C, and even Pascal and Modula-2 allow them under certain circumstances. However, the combination of these facilities in the way proposed in this paper allows simple solutions for some problems found in traditional programming language design. It may be said that this work followed Hoare's advice [Hoare 73] to whom language designer's task is mainly consolidation and not so much innovation.

It is assumed throughout the paper that Pascal-like type compatibility rules are used whenever applicable.

[1] UNIX is a trademark of AT&T.

2 Open Arrays

The *open array* concept is similar to the idea of *conformant arrays* of Pascal [JensenWirth 85], but extended to allow dynamic variables and procedures with a variable number of parameters.

An open array value will behave basically like a record which contains the actual bounds and the array components (or a pointer). The values of these bounds are established when the array value is created (or passed as a parameter) and from then on behave like constants.

Dynamic Variables

One of the contexts in which open arrays can be used are the dynamic (heap) variables. In this case, a type declaration of the form:

array [*L1..U1*: *T1*; ... ; *Ln..Un*: *Tn*] **of** *T*

is functionally but not syntactically equivalent to:

```
record
  L1,U1: T1;
  ...
  Ln,Un: Tn;
  "Components": array T1[L1..U1], ... , Tn[Ln..Un] of T
end
```

The notation T_i $[L_i..U_i]$ means that the elements of the range $L_i..U_i$ are of type T_i.

It is assumed that the allocation of a dynamic variable of this type can be done by the pseudo-procedure *New* with additional arguments which give the values of array bounds.

Example:

```
type
   V1 = array [LV1..UV1: integer] of integer;
   V2 = array [L1V2..U1V2: char; L2V2..U2V2: integer] of char;
   PtrV1 = pointer to V1;
   PtrV2 = pointer to V2;
var
```

```
    p,q: PtrV1;
    r:   PtrV2;
    ...
begin
    ...
    New (p,1,10);
      /* p: pointer to array [1..10] of integer */
    New (q,-1,1);
      /* q: pointer to array [-1..1] of integer */
    New (r,'A','Z',1,26);
      /* r: pointer to array ['A'..'Z',1..26] of char */
    ...
end
```

Once such an array is allocated, its bounds are constant, and accessible through the usual syntax for record components. In this example $p\uparrow.LV1 = 1$ and $r\uparrow.U1V2 = 'Z'$. The selector "*Components*" is not used and accessing array components is done through an abbreviated notation like $p\uparrow[i]$ or $r\uparrow[c,j]$ instead of $p\uparrow.$"*Components*"$[i]$ or $r\uparrow.$"*Components*"$[c,j]$. The compatibility rules of the language would be defined in a way to disallow assignments like $p\uparrow := q\uparrow$ (see [Carvalho 89]), but $p := q$ would be allowed, and p could be passed as an actual parameter, if the corresponding formal parameter is a compatible open array, as explained below.

Formal Parameters

A formal parameter can be declared as an open array. The corresponding actual parameter can be either a regular array or an open array, but compatible (conformant) with the formal parameter in a way similar to Pascal.

If the parameter is being passed by value, the formal parameter declaration of the form:

p: **array** [*L1*..*U1*: *T1*; ... ; *Ln*..*Un*: *Tn*] **of** *T*

is functionally but not syntactically equivalent to the declarations:

```
type Tpval =
       record
         L1,U1: T1;
         ...
```

```
        Ln,Un: Tn;
        "Components": array T1[L1..U1], ... , Tn[Ln..Un] of T
      end;
...
p: Tpval    /* formal parameter p */
```

The bounds $L_1, U_1, \ldots, L_n, U_n$ are established at the time of call and the elements of the array "*Components*" are copied at that time. The access is as before through the usual record syntax, except for the abbreviated form for the array components.

Example:

```
type
   V3 = array [1..5] of real;
   V4 = array [LV4..UV4: integer] of real;
var
   p1: V3;
   p2: pointer to V4;
   ...
procedure ArraySum (p: array [low..high: integer] of real): real;
   var
      i: integer; s: real;
begin
   s := 0.0;
   for i := p.low to p.high do s := s + p[i];
   return s
end ArraySum;
...
begin
   ...
   New (p2,0,99);
   ...
   Write (OutFile, ArraySum(p1));
   Write (OutFile, ArraySum(p2↑));
   ...
end
```

The case of **var** (reference) parameters is similar:

var *pp*: **array** [*L1*..*U1*: *T1*; ... ; *Ln*..*Un*: *Tn*] **of** *T*

is functionally but not syntactically equivalent to the declarations:

```
type Tpvar =
        record
          L1,U1: T1;
          ...
          Ln,Un: Tn;
          "ComponentsPtr": pointer to array T1[L1..U1], ...,
                                          Tn[Ln..Un] of T
        end;
...
var pp: Tpvar     /* formal parameter pp */
```

As before the bounds $L_1, U_1, \ldots, L_n, U_n$ are established at the time of call, and a pointer to the first element of the actual parameter is created or copied at that time. The bounds are referred to as $pp.L_i$ or $pp.U_i$, and the components through the abbreviated syntax $pp[i_1, \ldots, i_n]$.

Procedures with Variable Number of Parameters

In this proposal procedures (or functions) can handle a variable number of parameters through a special syntax mechanism which allows static type checking, and can be combined with automatic conversion to be described shortly. Procedure headers include the usual fixed formal parameters, followed by variable formal parameters declaration which is marked by the special symbol **pararray**, and treated like an open array.

The general form of a procedure header might look like:

procedure (*X1*: *T1*; ... ; *Xn*: *Tn*; *v*: **pararray** [*l*..*u*: *T*] **of** *Tpval*)

where X_i's are the fixed formal parameters, and v is the array of variable parameters. The parameter v could also be qualified by the symbol **var** to denote a reference parameter: in this case, the actual variable parameters must be all assignable variables. The value of the bound l will be the minimum value within the range type T.

Example:

```
procedure MaxReal (r: pararray [l..u: cardinal] of real): real;
   var
      i: cardinal; max: real;
begin
   max := r[r.l];
   for i := r.l +1 to r.u do
      if max < r[i] then max := r[i] end
   end;
   return max
end MaxReal;
...
s := MaxReal (1.2, x, y, 3.0, 5.0 * z);
    /* within this call: r.l = 0, r.u = 4 */
...
```

3 Automatic Conversion of Parameters

There are many situations where it is desirable to have automatic actual parameter conversion, specified by a procedure. Examples of such conversions might be integer to cardinal, integer to string (printable representation for I/O routines) and so on. In order to include such a facility this proposal introduces additional parameter passing mechanisms: **in**, **out** and **inout**. These are similar to the mechanisms existing in Ada [DraftAda 83], but it is assumed that the basic implementation is the call-by-copy rule.

These mechanisms are extended in this design by allowing the specification of conversion functions to be used at the time of call (for **in** parameters), of return (for **out** parameters) or both (for **inout** parameters). Let us consider, for example, a formal parameter specification of the form:

in (*f1*,*f2*,...,*fn*) *x*: *T*

It is assumed in this case that f_i's are one argument functions accepting values of type T_i and returning results of type T. When the procedure is called with an actual parameter e, its type must be compatible with one (and only one) of the types T_i (or T), and the corresponding function f_i is applied to the value of e before it is copied as the formal parameter x; if the type of e is T and no f_i is applicable, then no conversion is used. The type compatibility of the language

is defined in such a way that the decision about the choice of the conversion function is taken at compile-time.

A similar rule applies in the case of an **out** parameter which could be specified as:

out (*f1*,*f2*,...,*fn*) *x*: *T*

In this case, f_i's are one argument functions accepting values of type T and returning results of type T_i. The corresponding actual parameter must be an assignable variable of one of the types T_i (or T), and an automatic conversion is applied, if necessary, to the final value of x at the time of return.

Finally, a specification of the form:

in (*f1*,...,*fn*) **out** (*g1*,...,*gm*) *x*: *T*

combines both mechanisms.

Example:

```
procedure MaxNum (in (IntegerToReal) x,y: real): real;
begin
   if x > y
      then return x
      else return y
end MaxNum;
...
s := MaxNum (2,s);      /* s: real */
...
```

4 Combining a Variable Number of Parameters with Automatic Conversion

The mechanisms introduced in the previous two sections can be combined to specify variable number of parameters of varying types, but still maintaining the property of compile-time type checking. A particularly useful example of an application of this idea are the I/O routines. The problem is solved in Pascal by introducing pseudo-procedures *read* and *write* which do not follow the normal rules of the language. Functions in C allow variable number of parameters, but

the problem of type compatibility is left to the programmer.

Example 1:

```
procedure Read (f: File;
                out (StringToInt,StringToCard,StringToReal)
                rv: pararray [l..u: cardinal] of String);
   var
      i: cardinal;
begin
   for i := rv.l to rv.u do
      SkipSeparators (f);
      ReadString (f,rv[i])
   end
end Read;
```

It is assumed in this example that the type *String*, the conversion functions and the obvious file handling procedures are conveniently defined. Possible calls of the procedure *Read* could be:

```
var
   i,j: integer;
   d: cardinal;
   r: real;
   f: File;
...
Read (f,i,j,r);
...
Read (f,i,d);
...
```

Example 2:

Let's assume that *RealSort* is a procedure which sorts arrays of real numbers; its header might be for instance:

```
procedure RealSort (var a: array [l..u: integer] of real);
```

RealSort can be used to write another procedure which sorts a sequence of numerical variables:

```
procedure SortVariables (in (CardToReal,IntToReal)
                         out (RealToCard,RealToInt)
                         a: pararray [l..u: integer] of real);
begin
   RealSort (a)
end SortVariables;
```

Possible calls of the procedure *SortVariables*:

```
var
   i,j: integer;
   d: cardinal;
   r: real;
   v: array [-5..5] of cardinal;
...
SortVariables (i,j);
SortVariables (i,d,r);
SortVariables (v[-1],v[i],v[1]);
...
```

5 Conclusions

Two language mechanisms were described and combined in this proposal. In the first place, *open arrays* (also known as *semi-dynamic arrays*, cf. [Sebesta 89]) are used as values of dynamic (heap) variables and formal parameters. This kind of facility is basically an extension of the ISO Pascal concept of *conformant arrays*. In the second place, a controlled automatic parameter conversion is introduced, by including the specification of conversion functions in the procedure declarations.

It was shown that with this combination it is possible to solve some of the design problems in programming languages (typically I/O), without resorting to pseudo-procedures as in Pascal. This should be contrasted with C which allows for variable number of parameters which cannot be type checked. Ada has mechanisms for omission of some parameters, but their number and types are fixed; usual I/O operations require cumbersome multiple procedure calls, the same way as in Modula-2.

Finally, the new mechanisms can be implemented very efficiently, and their introduction does not affect the efficiency of other facilities already existing in programming languages like Pascal, Modula-2 or C.

References

[Cardelli et al. 88] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, G. Nelson. *Modula-3 Report.* Digital Research Center. August 1988.

[Carvalho 89] C. S. R. Carvalho. *Projeto de uma Linguagem de Programação.* Tese de Mestrado. DCC—IMECC—UNICAMP. Agosto 1989 (in Portuguese: *Design of a Programming Language*. Master Thesis).

[Carvalho 89a] C. S. R. Carvalho. *Linguagem MC — Manual de Referência.* DCC—IMECC—UNICAMP. Agosto 1989 (in Portuguese: *MC Language Reference Manual*).

[DraftAda 83] United States Department of Defense. *Ada Programming Language.* Draft ANSI/MIL-STD 1815A. January 1983.

[Hoare 73] C. A. R. Hoare. *Hints on Programming Language Design.* Sigact/Sigplan Symposium on Principles of Programming Languages. October 1973.

[IBM PL/I 88] IBM. *OS PL/I Version 2. Programming Language Reference, Release 2.* SC 26-4308-1. October 1988.

[JensenWirth 85] K. Jensen, N. Wirth. *Pascal User Manual and Report.* Third Edition, revised by A. B. Mickel, J. F. Miner. Springer-Verlag, 1985.

[KernRit 78] B. W. Kernighan, D. M. Ritchie. *The C Programming Language.* Prentice-Hall, 1978.

[Naur 63] P. Naur. *Report on the Algorithmic Language Algol 60.* Communications of ACM, vol. 6 #1, 1963. Pg. 1–17.

[Sebesta 89] R. W. Sebesta. *Concepts of Programming Languages.* Benjamin/Cummings Publishing Company, 1989.

[Wijngaarden et al. 69] A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster. *Report on the Algorithmic Language Algol 68.* Numerische Mathematic, vol. 14 #2, 1969. Pg. 79–218.

[Wirth 85] N. Wirth. *Programming in Modula-2. Third, Corrected Edition.* Springer-Verlag, 1985.

[Wirth 88] N. Wirth. The Programming Language Oberon. *Software–Practice and Experience, vol. 18 (7), 1988.* Pg. 671–690.

RELATÓRIOS TÉCNICOS — 1990

01/90 Harmonic Maps Into Periodic Flag Manifolds and Into Loop Groups — *Caio J. C. Negreiros.*

02/90 On Jacobi Expansions — *E. Capelas de Oliveira.*

03/90 On a Superlinear Sturm–Liouville Equation and a Related Bouncing Problem — *D. G. Figueiredo* and *B. Ruf.*

04/90 F– Quotients and Envelope of F–Holomorphy — *Luiza A. Moraes, Otília W. Paques* and *M. Carmelina F. Zaine.*

05/90 S-Rationally Convex Domains and The Approximation of Silva-Holomorphic Functions by S–Rational Functions — *Otília W. Paques* and *M. Carmelina F. Zaine.*

06/90 Linearization of Holomorphic Mappings On Locally Convex Spaces — *Jorge Mujica* and *Leopoldo Nachbin.*

07/90 On Kummer Expansions — *E. Capelas de Oliveira.*

08/90 On the Convergence of SOR and JOR Type Methods for Convex Linear Complementarity Problems — *Alvaro R. De Pierro* and *Alfredo N. Iusem.*

09/90 A Curvilinear Search Using Tridiagonal Secant Updates for Unconstrained Optimization — *J. E. Dennis Jr., N. Echebest, M. T. Guardarucci, J. M. Martínez, H. D. Scolnik* and *C. Vacchino.*

10/90 The Hypebolic Model of the Mean × Standard Deviation "Plane" — *Sueli I. R. Costa* and *Sandra A. Santos.*

11/90 A Condition for Positivity of Curvature — *A. Derdzinski* and *A. Rigas.*

12/90 On Generating Functions — *E. Capelas de Oliveira.*

13/90 An Introduction to the Conceptual Difficulties in the Foundations of Quantum Mechanics a Personal View — *V. Buonomano.*

14/90 Quasi-Invariance of product measures Under Lie Group Perturbations: Fisher Information And L^2-Differentiability — *Mauro S. de F. Marques* and *Luiz San Martin.*

15/90 On Cyclic Quartic Extensions with Normal Basis — *Miguel Ferrero, Antonio Paques* and *Andrzej Solecki.*

16/90 Semilinear Elliptic Equations with the Primitive of the Nonlinearity Away from the Spectrum — *Djairo G. de Figueiredo* and *Olimpio H. Miyagaki.*

17/90 On a Conjugate Orbit of G_2 — *Lucas M. Chaves* and *A. Rigas.*

18/90 Convergence Properties of Iterative Methods for Symmetric Positive Semidefinite Linear Complementarity Problems — *Álvaro R. de Pierro* and *Alfredo N. Iusem.*

19/90 **The Status of the Principle of Relativity** — *W. A. Rodrigues Jr.* and *Q. A. Gomes de Souza.*

20/90 **Geração de Gerenciadores de Sistemas Reativos** — *Antonio G. Figueiredo Filho* e *Hans K. E. Liesenberg.*

21/90 **Um Modelo Linear Geral Multivariado Não–Paramétrico** — *Belmer Garcia Negrillo.*

22/90 **A Method to Solve Matricial Equations of the Type $\sum_{i=1}^{p} A_i X B_i = C$** — *Vera Lúcia Rocha Lopes* and *José Vitório Zago.*

23/90 **$\mathbb{Z}_2$–Fixed Sets of Stationary Point Free $\mathbb{Z}_4$–Actions** — *Claudina Izepe Rodrigues.*

24/90 **The m–Ordered Real Free Pro–2–Group Cohomological Characterizations** — *Antonio José Engler.*