FORMAL TYPES AND THEIR APPLICATION TO DYNAMIC ARRAYS IN PASCAL

by

Sergei POKROVSKY

Computing Center of the Siberian Division
of the USSR Acad. Sci.
Novosibirsk 630090, USSR

Abstract. The formal type concept is presented as a means to uniformly introduce in the PASCAL language the dynamic array facility (which may be done as a pure extention) and formal procedure [4] specifications (which would require some changes in the standard language [1]).

1. Type Equivalence Relation

In any programming language with programmer-defined type capability there exists a type equivalence relation which arises from consistency rules for var-parameter types and from definition of the class of types, references to variables whereof may be assumed by a pointer. Unless this relation is trivial, an honest implementation must provide for a procedure to check the equivalence of programmer-defined types (this is the case of PL/1 and ALGOL 68).

There is a trend to make this equivalence more strong: most languages, like ALGOL 68, distinguish between, e.g.

```
type complex = record re, im : real end;
     polar = record ro, fi : real end;
```

which would be equivalent structures in PL/1; most of PASCAL implementations (e.g. [6]) go even further and treat, say,

```
type velocity = array [1..3] of real;
     acceleration = array [1..3] of real;
```

as distinct types, so that the equivalence becomes trivial: any type which is not just a type identifier forms an equivalence class of its own.

This reinforcement makes implementation easier, but it also may be viewed as a means to improve the "programming security" and to facilitate updating a program (all equivalent types being defined just once). It may also be useful to refine the program analysis for optimization (or correctness proofs).

This feature, however, complicates writing general procedures (functions), e.g. a norm function applicable to both velocity and acceleration.

In the next section we present a solution to this problem based on formal types - type patterns, which may be used to form some actual types (these are said to be "instantiations" of their formal type), and to qualify var-, procedure or function parameters and referenced variables. In other words, a formal type represents the equivalence class of all its instantiations.

2. Formal Arrays

We propose the following extentions to the Report's [1] syntax (to introduce specifications for formal procedures further changes are required, cf. Section 3):

```
⟨type definition section⟩ ::= ⟨empty⟩ |
type ⟨pseudotype definition⟩ {; ⟨pseudotype definition⟩} ;

⟨pseudotype definition⟩ ::=
⟨type definition⟩ | ⟨formal type identifier⟩ = ⟨formal type⟩

⟨formal type identifier⟩ ::= ⟨identifier⟩
⟨formal type⟩ ::= ⟨unpacked formal type⟩|
packed ⟨unpacked formal type⟩

⟨unpacked formal type⟩ ::=
⟨formal type identifier⟩ | ⟨formal array⟩
```

(There is a negligible ambiguity in "type x = y".)

⟨formal array⟩ ::=
<u>array</u> [⟨formal index type⟩ {, ⟨formal index type⟩}] <u>of</u> ⟨pseudotype⟩

⟨pseudotype⟩ ::= ⟨type⟩ | ⟨formal type⟩

⟨formal index type⟩ ::= * | * ⟨type identifier⟩

The first alternative in the last rule is a default for the integer ⟨formal index type⟩. - The following rules show how a formal array is used to construct actual types (its instantiations):

⟨array type⟩ ::=
<u>array</u> [⟨index type⟩ {, ⟨index type⟩}] <u>of</u> ⟨component type⟩ |
⟨formal type identifier⟩ [⟨index type⟩ {, ⟨index type⟩}]

In the second case the ⟨index type⟩s must coincide with the corresponding ⟨formal index type⟩s, or be subranges thereof.

⟨pointer type⟩ ::= ↑ ⟨pseudotype identifier⟩

⟨pseudotype identifier⟩ ::= ⟨identifier⟩

⟨file type⟩ ::= <u>file</u> <u>of</u> ⟨pseudotype⟩

⟨var parameters⟩ ::=
<u>var</u> ⟨identifier⟩ {, ⟨identifier⟩} : ⟨pseudotype identifier⟩

The latter production is an obvious modification in the ⟨formal parameter section⟩ notion.

Our change in the pointer type definition also implies some extentions to the NEW statement to make dynamic arrays more "flexible" (cf. ALGOL 68):

⟨new statement⟩ ::= new (⟨pointer expression⟩ ⟨allocation parameters⟩)

⟨pointer expression⟩ ::= ⟨variable⟩

⟨allocation parameters⟩ ::=
{, [⟨bound setting⟩{,⟨bound setting⟩ }] }{, ⟨case label expression⟩ }

⟨bound setting⟩ ::= ⟨type identifier⟩ | ⟨expression⟩..⟨expression⟩

⟨case label expression⟩ ::= ⟨expression⟩

Syntax and semantics of accessing thus created variables are evident from the examples below; in fact, they are covered by the standard PASCAL notation. Only two more operations are to be added (for the lack of obvious priorities we give them the "standard function" notation); these are "top" and "bottom", which produce upper and lower bound values using an array reference and an integer dimention number as arguments.

2.1. An Example of Formal-Array Parameter Specification.

In the scope of

```
type matrix = array [*,*] of real; refmatrix = ↑ matrix;
var  A10: matrix[1..10,1..10];  AA: refmatrix;
procedure times(coefficient: real, var A: matrix);
     var i,j: integer;
     begin for i := bottom(A,1) to top(A,1)
           do for j := bottom(A,2) to top(A,2)
              do A[i,j] := coefficient*A[i,j]
     end;
```

two kinds of procedure statements are possible:

```
times(3.0,A10); times(7.0,AA↑);
```

2.2. An Example of Formal-Array Record Field.

Although PASCAL textfiles are quite satisfactory in symbol manipulation, à titre d'exemple we present a couple of more conventional procedures (which also suggest something like getchar, setpos etc.).

```
type multialfa = array [*] of alfa;  refalfa = ↑ multialfa;
     stri = record pos: integer; main: refalfa end;
     string = ↑ stri;
procedure newstring(var s: string, maxlen: integer);
     begin new(s);
           with s
           do begin pos := 0;
                    if maxlen > 0
                    then new(main, [1..-(-maxlen div alfalen)])
                    else main := nil;
     end;      end;
```

```
procedure copy(var source, target: string);
    var i,k: integer;
    begin k := -(-source↑.pos div alfalen);
          if top(target↑.main↑,1) < k then new(target↑.main, [1..k]);
          for i := 1 to k do target↑.maim↑[i] := source↑.main↑[i];
          target↑.pos := source↑.pos;
    end;
```

3. Formal Procedures

Another controversial issue in PASCAL is the lack of sufficient specification for formal procedures ([4,5]; the subsequent discussion is a fortiriori valid for "formal functions"). In presence of parameterless functions, correct coding of

```
procedure p(procedure q; function f: real); begin q(f) end;
```

requires run-time type checking, or inefficient thunks, or a fairly tiresome analysis of procedure calls, which strangely contrasts with obligatory label specifications.

As the proposed solution is incompatible with some PASCAL's notational commitments, we will give only a few examples.

Like types, procedures may be arranged in equivalence classes; specification of a class could take the form of "projected" procedure heading with all constituent parameter identifiers being replaced with asterisks, e.g.

```
procedure class binary(var *,*,*: matrix);
```

(so, unlike [5] we would allow any mode of parameter passing).

Here is a more elaborate example:

```
function class sequence(*: integer): real;
sequence function harmonic(n);
    {parameter and result specifications need not be repeated;
     the "actual" function body:}
    begin if n = 0 then harmonic := 1 else harmonic := 1/n end;
function powerseries(function seq: sequence; x, eps: real): real;
    var t,tt,xx: real; i: integer;
    begin t := seq(0); i := 0; xx := 1;
          repeat xx := x*xx; tt := t;
                 t := t + seq(i)*xx; i := i + 1;
          until  abs(t-tt) ≤ eps;
          powerseries := t; end;
```

An example of function designator:

powerseries(harmonic,-0.5,0)

4. Conclusion

The proposed solutions present the following advantages:

- they are logical and uniform;
- they are consistent with systematic (top-down, classification) style of programming;
- they considerably enhance the expressive power of the language;
- they are efficient at both compile- and run-time;
- saving your presence, they are easily implementable.

I do not think that "complication (or extention) of an already complex language" would be a valid objection in this case. First, the modularity is preserved, and if you can do without dynamic arrays and formal procedures, you need not know anything about formal types; but if the nature of your problem requires these facilities, it is better to be aware of their limitations. This means, secondly, that formal type concept only visualizes a convention which should be explicitly stated anyway; this clarifies, rather than complicates, the matter.

In fact, improving modularity (or orthogonality), these proposals enable a student with purely numeric interests completely ignore the dynamic allocation and yet write general matrix manipulation procedures (using formal arrays in var parameters), which would be impossible with earlier proposals [2,3]. - I admit that this is achieved at the expense of those interested in genuinely dynamic arrays, for the consistency requires a fictitions dereferencing (cf. examples in Section 2)

top(AA↑,2) because of top(A,2) or top(A10,2)

The other case is

times(7,AA↑); because of times(3,A10);

but this is only implementation nuisance, for the programmer who needs only dynamic (anonymous) arrays would use a pointer value- or var parameter with quite natural notation (cf. [3]).

Acknowledgements

The Author thanks N.N.Dudorov and I.V.Pottosin for useful discussion.

References

1. K.Jensen, N.Wirth. "PASCAL: User manual and Report". Lecture Notes in Computer Science, No. 18, 1974.

2. B.J.MacLennan. "A note on dynamic arrays in PASCAL", SIGPLAN Notices 10, 9, pp. 39-40 (Sept. 1975).

3. N.Wirth. "Comment on a note on dynamic arrays in PASCAL", SIGPLAN Notices, 11, 1, pp. 37-38 (Jan. 1976).

4. O.Lecarme, P.Desjardins. "Reply to a paper by A.N.Habermann on the programming language PASCAL", SIGPLAN Notices 9, 10, pp. 21-27 (Oct. 1974). Cf. also

5. O.Lecarme, P.Desjardins. "More comments on the programming language Pascal", Acta Informatica 4, 3, pp. 231-244 (1975).

6. P.Brinch Hansen. "The programming language Concurrent Pascal". IEEE Fransactions on Software Engineering 1, 2 (June 1975).