7

The Pascal Standard from the implementor's viewpoint

J. Welsh and A. Hay

Introduction

The Pascal Standard [1] presents a new challenge to implementors, a challenge that is reinforced by the Pascal Validation Suite. When compared with the previous Pascal Report [2] the length and detail of the new Standard must give an implementor cause for concern. This apprehension can only deepen at the prospect of 400 or so programs specifically designed to find shortcomings in any compiler. What then lies in store for an implementor who takes up this challenge?

This paper summarises one implementation project's experience to date in meeting the requirements of the Standard. The project began out of technical interest during the development of the draft standard itself, but has now become an NPL supported project to provide:

(1) a Standard Pascal static checker (SPSC) that will enforce all compile-time enforceable checks on Pascal programs, and

(2) a Standard Pascal model implementation (SPMI) which will demonstrate the techniques required for implementation of all run-time error checks implied by the Standard, albeit on an idealised P-machine architecture.

Both the SPSC and SPMI will be made available by NPL and BSI to help and encourage other implementors to achieve a high level of conformance to the Standard and the Validation Suite. At the time of writing, the SPSC is complete, in the sense that it passes all relevant tests in version 3.0 of the Validation Suite. The SPMI should be complete by mid-1982. The development of the SPSC and SPMI is based on considerable previous experience of implementing Pascal. In the case of the first author, this dates from the initial bootstrap of a Pascal compiler to an ICL 1900 at the Queen's University of Belfast in 1971 [3]. Subsequent developments in Belfast led to an improved modular structure for Pascal compilers [4], a systematic method for exploiting compile-time range analysis to avoid the need for run-time error checks [5], and, in conjunction with the University of Glasgow, a source-related run-time diagnostic package [6]. The resultant 1900 Pascal system has been used as the basis for several other Pascal implementations, and its distinctive features have been incorporated and refined in the SPSC/SPMI development. In the case of the second author, previous experience includes a P-code implementation and maintenance of the highly successful Pascal 6000 compiler for CDC computers.

Conformant arrays

Conformant array parameters are undoubtedly the major addition to Pascal

from the implementor's point of view. Although strictly an optional feature, their definition within the Standard makes their implementation mandatory for all serious implementations of the language.

Handling variable length array parameters is not a new problem and the basic techniques have been well established in other languages. Indeed, some Pascal implementations, such as the Pascal 6000 compiler [7], already offer similar facilities with a somewhat different syntax. However, the Pascal Standard's equivalence rules for array types (which equate multidimensional arrays (one dimensional) arrays of arrays of etc.) do create some traps for the unwary implementor. Suppose we have declarations, as follows:

```
type vector = array [1..3] of real ;
var vmatrix : array [1..10, 1..20] of vector ;
```

The entire variable **vmatrix** may be passed as parameter to a procedure P declared as:

```
procedure P (var a: array [m1..n1: integer; m2..n2: integer]
            of vector)
```

Alternatively, a single row of **vmatrix**, **vmatrix[i]** say, may be passed as parameter to a procedure Q declared as:

```
procedure Q (var b: array [m3..n3: integer] of vector)
```

Within Q the parameter b may in turn be passed as a parameter to a procedure R declared as as:

```
procedure R (var c: array [m4..n4: integer; m5..n5: integer]
            of real)
```

Because of this multi-level view of array structures in Pascal, each bound pair required to describe an actual conformant array parameter must be constructed or copied independently. Because several actual array parameters may share the same set of bound pairs, the base address for each must also be passed and stored separately from the bound pairs. Within the receiving procedure, the set of bound pairs is used in conjunction with the approximate base address to provide an effective array descriptor. For these reasons, the copying of fixed-format array descriptors is not an adequate means of handling Pascal's conformant array parameters.

The Standard's definition of value conformant arrays is carefully worded to allow the necessary copy of the actual array parameter to be made in the calling environment where its size is known, so enabling fixed size procedure activation records to be maintained. In practice, however, implementations that use extensible activation records may find it more convenient to create the copy in the called procedure after control has been transferred, so avoiding any interaction between any copies created and further evaluation of the parameter list, and any problems with their subsequent disposal.

Provided the implications of the Standard are understood, the implementation of the conformant array parameter mechanism reduces to a

straightforward if painstaking encoding of the Standard's carefully worded rules. In the static checker the analysis required for conformant arrays forms an alternative parallel path to that for handling fixed arrays and constitutes an approximate 7% increase in the checker's overall length. The corresponding logic to generate conformant array access code in the SPMI is not yet complete but is expected to produce a similar increase in the overall compiler size.

The relative speed of element access for conformant and fixed arrays depends on the target machine involved and the indexing techniques used on it. For some, conformant array access will be significantly slower, for others it will not. With the mechanism used in the Pascal 6000 compiler access to 2-dimensional conformant array elements is about 20% slower than that for equivalent fixed arrays. What is generally true is that elimination of subscript checking by compile-time range analysis as described in [5] is not immediately applicable to conformant arrays, since the necessary subscript bounds are not known at compile-time. Thus, the considerable overheads normally associated with subscript checking in FORTRAN or Algol will remain for conformant arrays in Pascal. The development of some means of eliminating such checks, at least in simple contexts such as :

```
procedure P (var a : array [m..n:integer] of T)
   var  i : integer ;
   begin
     for i := m to n do .... a[i] .....
   end ;
```

is an obvious priority for compilers supporting conformant arrays, which we hope to pursue in a second phase of SPMI development.

Additional compile-time requirements

The Standard imposes a number of requirements on Pascal implementations which involve additional compile-time processing, but no change in the executable object programs produced. These arise mainly from additional or more precise definition of language features to enable more compile-time detection of programming errors, but in one case from a greater freedom of expression now permitted than that previously.

Formal procedures and functions

A significant syntactic extension introduced by the Standard is the notation required to define the parameter list requirements of procedures and functions which are themselves formal parameters of other procedures and functions. At first sight, this additional feature appears to require significant additional syntax analysis code within a compiler, but in practice this is not so. Because the declaration of a formal procedure or function takes the same form as a heading used in actual procedure and function declarations, a well-structured compiler will use the same analysis code for both - giving rise to a reduction in the overall code required rather than an increase.

Identifiers of arbitrary length

The Standard now makes clear that Pascal identifiers may be of any length and that all characters of identifiers are significant in distinguishing between them. In the past, most compilers have followed the recommendation of the Report in limiting significant length to 8 or 10 characters, and, therefore, have limited the storage required for identifier spellings at compile-time. For these compilers the new Standard demands a change, which should be implemented to achieve the following objectives:

(i) to avoid any time overhead in distinguishing identifiers from word symbols,

(ii) to minimise the time overhead in comparing identifiers, and,

(iii) to minimise the storage overhead in holding the full identifier spellings.

In the SPSC these objectives have been realised by representing identifier spellings as records of type alfa defined as follows:

```
alfahead = packed array [1..headlength] of char ;
alfatail =  tailrecord ;
tailrecord = record
               chunk : packed array [1..chunklength] of char ;
               rest : alfatail
             end ;
alfa = record
         head : alfahead ;
         tail : alfatail
       end
```

The headlength is chosen to be greater than the length of the longest word-symbol (**procedure**), so that identifiers and word symbols may be distinguished by direct comparison of heads only, and in this way objective (i) is achieved. To achieve objective (ii) identifier comparison during table searching uses an in-line comparison of heads, followed by a function call to compare tails if necessary. Because the vast majority of comparisons are resolved by the head comparison alone the resultant degradation of table searching speed is negligible.

The length of chunks used within identifier tails should be chosen to give an economic balance on the particular computer involved between the chunk storage itself and the overhead of their linking pointers. In practice, reasonable variations in this balance have little impact on the overall storage requirement as the majority of identifiers used by Pascal programmers generate little or no tail storage. The major overhead in meeting the requirement of the new Standard is, therefore, the additional tail pointer (usually **nil**) that must be stored for every identifier. For many current implementations this represents a storage overhead of around 10% per identifier table entry. Since identifier storage is a significant proportion of the total storage requirement in compiling large programs, (25% for the 1900 Pascal compiler itself), the new requirement will produce a measurable storage increase for compiling such programs.

Identifier Scope Rules

The rules of scope in the new Standard now make clear that a use of a non-local definition of an identifier in a block B (or any nested block) must not precede its redefinition in block B.

Thus, in the following fragment both N and X are redefined illegally within procedure P:

```
const N = 10 ;
      X = 100 ;
procedure P ;
   type T = 1..N ;                {using constant N}
   var N : integer ;              {redefining N}
   procedure Q ;
       begin
         write (X)                {using constant X within Q}
       end {a} ;
   procedure X ;                  {redefining X in P}
       .
       .
```

In the past, most Pascal compilers allowed such programs by considering the scope of a new definition to start from its defining point - an easy implementation option. Sale [8] has described a simple algorithm for enforcing the new rules but it involves numbering all scopes encountered in the program in order of their opening, and recording in each identifier table entry the number of the latest scope in which it is used. The algorithm produces no significant overhead in compilation-time but does mean the storage of one new field in each identifier table entry. The precise impact of this additional field on the total storage will depend on the limit set for the total number of scopes in any program, and on the packing possibilities within the existing identifier table entries, but in general it may lead to a measurable increase in table storage of the order of that described for the implementation of identifiers of arbitrary length.

For-statement restrictions

The Standard imposes new restrictions on the control variable of a for-statement which makes illegal any assigning reference that threatens to alter the value of the control variable within the body of the for-statement, or within any procedure or function declared in the same block. To enforce these rules the compiler must create and examine two sets of variables for each block B:

(i) the set T_B of local variables that are threatened by any local procedure or function,

(ii) the set C_B of local variables currently in use as control variables of for-statements.

The set T_B is constructed during compilation of the nested procedures and functions of B, and examined during compilation of the statement part of B, so it exists throughout the compilation of B. The set C_B exists only

during compilation of B's statement part.

Each set may be represented either as a corresponding boolean flag held in the table entry of each variable identifier, or as a chained list of pointers to relevant table entries. If the additional boolean flags can be accommodated within existing packed identifier table entries without increasing the overall entry size the first method is preferable, otherwise the second may be more economic. When using the latter it should be remembered that only those variables that can occur as control variables, i.e. those of ordinal type, need to be recorded in the threatened set T of any block.

The processing required on meeting any assigning reference to an entire variable v declared in block B to be of ordinal type is as follows:

```
if B is not the local block
then T_B := T_B + [v]
else if v in C_B
     then error {assigning control variable
                 within for statement}
```

On encountering a **for** statement for v := .... the compile-time processing required is:

```
if v is not of ordinal type
then {control variable must be of ordinal type}
else if v is not local to B
     then error {control variable must be local}
     else if v in T_B
               then error {control variable is threatened
                           by procedure or function}

     else if v in C_B
               then error {same control variable in nested fors}
               else C_B := C_B + [v]
```

and at the end of the statement the processing required is simply:

```
C_B := C_B - [v]
```

With either of the set representations suggested, implementation of this logic does not result in a significant increase in the compiler code length or in the compilation time for typical Pascal programs. Measurements with the SPSC indicate that even with the chained representation of identifier sets the storage overhead for maintaining the sets is not significant.

Label/Goto Restrictions

The Standard requires the enforcement of checks on the accessibility of labels by **goto** statements, with an additional specific restriction on the non-local **goto** statements allowed.

To enforce the checks each declared is represented by a record defined as follows:

```
labeldepth = 1..maxint ;
labelrec = record
             ..... ;
             case sited : boolean of
               false : (maxdepth : labeldepth) ;
               true  : (case accessible : boolean of
                          false : ( ) ;
                          true  : (depth  :  labeldepth))
           end
```

At the declaration of a label its record L is initialised as follows:

```
with L do
begin
  sited := false ;
  maxdepth := maxint
end
```

At an occurrence of a **goto** statement that references a label with record L the checking required is as follows:

```
with L do
  if L is non local
  then maxdepth := 1
  else if sited
       then begin
              if not accessible then error {label too deep
                                            this goto}
            end
       else if maxdepth > depthnow
            then maxdepth := depthnow
```

At an occurence of a statement labelled with the label with record L, the checking code required is

```
with L do
  if sited
  then error {doubly sited label}
  else begin
         if maxdepth < depthnow
         then error {label too deep for previous goto} ;
         sited := true ;
         accessible := true ;
         depth := depthnow
       end
```

The variable depthnow is initialised to 1 at the start of each statement part, and is increased by 1 at the start of each unlabelled statement, i.e. *after* processing the statement label, if any, as above. At the end of each statement the following processing is applied to *all* label records for the current block.

```
for all label records L do
  with L do
    if sited
    then begin
```

```
            if accessible
            then if depth = depthnow
                 then accessible := false
          end
      else if maxdepth = depthnow
           then maxdepth := maxdepth-1 ;
  depthnow := depthnow-1
```

The cost of this label processing at the end of each statement is insignificant only because the number of labels declared in any Pascal block is usually very small, if not zero.

Additional run-time requirements

Apart from its introduction of conformant array parameters, the Standard impinges on the run-time behaviour of Pascal programs in two ways:

(1) A number of minor adjustments and clarifications of the executable effect of existing Pascal features may require corresponding changes in the generated code or run-time support routines of existing Pascal implementations. The changes required are clearly dependent on the existing implementation concerned and cannot be catalogued in any general way, but typical examples are the code generated for the **div** and **mod** operators, or the conversion routines used for numeric output.

(2) The Standard now provides a precise definition of those run-time events that are designated errors and should be treated as such. Although compliance with the Standard does not **require** the detection of errors at run-time, their clear definition now makes it possible, and highly desirable, for implementors to provide such detection, at least as a user option.

Appendix D of the Standard catalogues 59 errors whose occurrence at run-time should be detected. In the following sections we summarise the problems and run-time cost of detecting these errors under six general headings.

File errors

Of the 59 errors listed, 14 relate to the state of file variables during operations on them, or the values of parameters involved in such operations. None of these errors is difficult to detect within the file processing code involved, and most are already handled by existing implementations or require only minor adjustment of these. Because of the nature of file processing and the way it is implemented, the cost of implementing the detection of these errors is not significant, either within the compiler or within the programs generated. There seems no justification for an implementation that fails to provide such detection, or for providing it on an optional basis.

Range errors

Almost half the errors listed (29) relate to limits on the range values taken by scalar variables and expressions in certain contexts. These include the

familiar array subscript error, case index error and subrange variable assignment error, but also include overflow during real or integer arithmetic, which are also range variations of an implementation-defined kind. These errors have always been well-defined in Pascal, and their detection is supported by most implementations, at least on an optional basis. The significant feature of Pascal in comparison with other languages is that the careful use of subrange declarations can enable the compile-time elimination of run-time range checks in many cases [5], and so avoid the high run-time overheads traditionally associated with range checking. In the case of the 1900 Pascal compiler the compile-time analysis to achieve this elimination of run-time check led to a 3% increase in compiler code with a decrease in compilation speed of less than 1%. Such costs are negligible when compared to the benefits obtained, and the adoption of similar range analysis techniques in other Pascal compilers is strongly recommended. The difficulties in exploiting the technique for conformant array indices does not diminish its effectiveness in other areas.

Undefined values

Six errors are listed which involve the use of undefined (i.e. unassigned) variables or function results. These errors have always been clearly recognised, but few implementations have attempted to detect them because of the prohibitive run-time overheads involved.

In principle, each possibly undefined variable v of type T must carry with it a defined tag, if undefined value errors are to be detected. In effect, the required representation has the form :

```
v : record
      case defined : boolean of
       false : ( ) ;
       true : (value : T)
    end
```

The defined tag is set false at the creation of v, becomes true when v is assigned a value, and must be checked in each context where v is required to have a defined value.

In practice, the defined tag can often be accommodated within the representation of the value itself without additional storage overhead, but the high cost of setting up the undefined value representation and of checking for it subsequently still remain.

For variables of structured types the error detecting requirements are even more demanding. Since each component of a variable of (unpacked) structures type may be passed as a variable parameter and manipulated without any awareness of its surrounding structure, it follows that each component must carry it own defined tag with it. This in turn means that a (partially) undefined test for a structured variable involves inspecting the defined tags of each of its components in turn. For variables of a packed structured type, the components are not independently accessible, and for these an alternative strategy of grouping defined tags together for easier inspection could be considered. It is not clear, however, that in general the advantages of doing so would outweigh the disadvantages for individual

component updating.

Some reduction of the high run-time cost of undefined checks may be achieved at compile-time, by maintaining a record of the accessible variables in any block that have definitely been assigned values at or since block entry. For value parameters, and for simple local variables which are often assigned values before any conditional control statements are encountered, this technique should enable the elimination of the corresponding undefined checks, with a significant reduction in the code storage overheads incurred. Unfortunately, however, the highest time overheads result from checks applied during repetitive processing, usually of data accessed either by array indexing or via pointers. The dynamic nature of this access often precludes elimination of the corresponding checks by any simple compile-time analysis. In such cases, hardware assistance in implementing the checks is the most likely means of reducing the overheads involved.

Variant errors

The Standard provides a precise definition of the rules governing variant records with and without tag fields. These rules give rise to six detectable errors that occur through illegal variant field manipulation or through inconsistent use of the extended form of new and dispose. The checks for these errors are the most complex of those required by the Standard, both in the compile-time processing required for their generation and in the run-time processing that their application involves. A tentative model for their implementation within the SPMI has been defined, but it would be foolhardy to describe it in detail in advance of its implementation and testing. Its essential ingredients are the maintenance of a variant check record for each active level of variant nesting within each record together with a cascade of variant checks, from the outermost variant level inwards, to establish the validity of variant manipulaton at any level.

Whatever the precise form of the checks involved, it is clear that their introduction will put considerable overheads on the processng of variant record data. The scale of these overheads makes it imperative that compile-time techniques for the elimination of some of these run-time checks be investigated. The with-statement in Pascal gives a basis for a limited reduction in such checks - by propagating local assertions about the currently established variant status of the record within the with-statement it should be possible in suitable cases to perform the necessary variant checks once per with-statement rather than once per variant field access. If a case-statement with the tag field as case-index is used within the with-statement it may be possible to eliminate the checks altogether. In general, however, the dynamic access (via pointers) used for many variant records will make elimination of variant checks difficult to achieve on any global basis.

Pointer errors

The new Standard lists only two errors specifically relating to pointers, other than those concerning undefined values or the creation and disposal of variant records. These errors involve the occurrence of the special pointer value nil in an identified-variable access or in a dispose operation, and as

such are easily detected.

However, the undefined value error for pointers is a more significant problem than it is for variables of other types. The Standard states than when an identified variable is disposed all pointer variables pointing to it become undefined, but it is impractical for an implementation to locate such variables and adjust their values at that point. Thus 'dangling' pointers are created whose subsequent use must be detected and reported as an undefined pointer error.

To implement these checks the technique described in [9] seems the logical choice. This relies on the incorporation in each pointer value created by a new operation of a unique key value which is also stored within the identified variable storage allocated. The representation of a pointer is thus equivalent to that of the following record type :

```
pointer = record
              case defined : boolean of
             false : ( )
             true : (case nilvalue : boolean of
                      true : ( ) ;
                      false : (key : keyvalue ;
                               address : heapaddress))
          end ;
```

and the representation at every heap variable allocated has the form :

```
record
   key : keyvalue ;
   variable : ......
end
```

The check required for each identified variable p↑, and for each dispose(p) operation, is then as follows:

```
with p do
  if not defined
  then error {use of undefined pointer}
  else if nilvalue
       then error {use of nil pointer}
       else if p.key <> p↑.key
            then error {use of dangling pointer}
            else.....
```

Since the key value allocated by each call of new is unique and the stored copy of it in a disposed variable is destroyed in the dispose operation, the above logic is sufficient to detect most dangling pointer errors. It is not totally secure because subsequent re-partitioning and re-use of the storage addressed by a dangling pointer may accidentally recreate the exact bit pattern of the original key! In addition, if the storage set aside for keys within pointers and heap variables is to be fixed in size, some limit must be imposed on the number of unique key values available, and hence on the number of new operations that can be carried out by any program. Such a limit, however large, may be unacceptable for some application programs. With these provisos, the unique key technique provides a practical means of detecting

the vast majority of dangling pointer errors.

The cost of achieving this pointer security is significant, in terms of both run-time storage and execution speed, and all means of reducing it must be considered. Unfortunately, the nature of pointer processing makes it difficult to achieve much reduction by additional compile-time analysis. Even in programs areas of passive pointer inspection, involving no modification of the data structures represented by pointers, it is difficult for a compiler to establish that any pointer value is 'safe' and not in need of checks. If significant reductions in the cost of checking Pascal pointers are to be achieved it is likely to be by hardware assistance that makes the checks more efficient, rather than by compile-time analysis that eliminates the need for them.

Existence errors

The Standard requires that a variable must continue to exist as long as any reference to it exists. Many references are transient and pose no problems in this respect, but references of extended duration do arise from passing a variable as a variable parameter, or from a with-statement in the case of a record variable. Assignment statements and even indexed variables, may also give rise to extended references depending on the implementation chosen.

Appendix D of the Standard explicitly identifies two ways in which this requirement may be violated - by performing a file operation when a reference to the file-buffer exists and by disposing of a heap variable when a reference to it exists. However, the same error may arise through a change of variant in a variant record and is a special case of a more general variant record error.

To detect these errors requires the association of a reference history with each file-buffer variable, heap-variable and variant part. This history must be updated when an extended reference is established and checked when a file operation, dispose or change of variant occurs. Resetting the history must take place on leaving the procedure, with-statement or assignment that established the reference, either normally or abnormally via a **goto** statement. To do so seems to require the maintenance of a stack of extended references at run-time. Code to unwind this stack (to compile-time determined levels) must be inserted at all points where the history may have to be reset, including all statement labels that may be used for abnormal exit from procedures or statements.

A precise moel for the implementation of these checks has not yet been defined, but for straightforward one-pass compilers, it appears that they will impose some storage overhead on all variant records, heap variables and file buffers, and an additional overhead, in storage and time, on many procedure calls. The extent to which these overheads can be avoided by compile-time analysis requires further investigation.

Conclusions

Experience to date on the SPSC/SPMI project suggests that the new Pascal

Standard can be fully complied with, both in its mandatory compile-time requirements and in its optional run-time error detection. The new conformant array facility is a significant addition for many existing implementations, but is clearly implementable at a reasonable cost. The other additional compile-time requirements are easily met, and the cost of doing so is significant only in the increased symbol table storage that may result. The additional run-time checks, some of which were not previously well defined, are also implementable, but at a significant cost in the running time of the resultant programs. The benefits of retaining such checks in running programs, not just during development but throughout the programs' useful lifetime, more than justify further investigation of techniques to reduce the cost involved by some judicious blend of additional compile-time analysis and run-time hardware assistance.

References

[1] Specification for the Computer Programming Language Pascal, ISO 7185.

[2] Jensen K and Wirth N, 'Pascal-User Manual and Report', Lecture notes in Computer Science, 18, Springer-Verlag (1974).

[3] Welsh J and Quinn C, 'A Pascal Compiler for ICL 1900 Series Computers', Software - Practice and Experience, 2 Vol. 1, 73-77, 1972.

[4] Welsh J, 'Two ICL 1900 Pascal Compilers', in Pascal - the language and its implementation (ed. D W Barron), John Wiley & Sons, 1981.

[5] Welsh J, 'Economic Range Checks in Pascal', Software - Practice and Experience, 8, 85-97, 1978.

[6] Watt D A and Findlay W, 'A Pascal Diagnostics System', in Pascal - the language and its implementation (ed. D W Barron), John Wiley & Sons, 1981..

[7] Strait J P and Mickel A B, 'Pascal 6000 Release 3', University Computer Centre, University of Minnesota, Minneapolis, USA (1979).

[8] Sale A H J, 'A Note on Scope, One-Pass Compilers and Pascal', Australian Computer Science Communications, 1, 1, 80-82, 1979.

[9] Fischer C N and LeBlanc R J, 'The Implementation of Run-time Diagnostics in Pascal, IEEE Transactions on Software Engineering, 6.4, 313-319, 1980.