EUROPEAN ORGANIZATION FOR NUCLEAR RESEARCH

CERN-DD /89/ 10

March 17, 1989

Recent Progress in Fortran Standardization

Michael Metcalf

Data Handling Division, CERN

Abstract

CERN LIBRARIES, GENEVA

CM-P00059910

To be presented at the Computing in High-Energy Physics Conference
Oxford, 10 – 14 April, 1989.

RECENT PROGRESS IN FORTRAN STANDARDIZATION

Michael Metcalf

CERN, Geneva, Switzerland

Abstract

The fortunes of Fortran have changed markedly over the last two years. An increasingly gloomy outlook for the new standard, Fortran 8x, culminated in a complete impasse in the summer of 1988. The deadlock was broken as the result of firm action by the international community and X3J3's positive response to that pressure. In this short talk, the consequent changes which have been made to the draft standard since last autumn will be outlined.

1. Standardization

At the last conference in this series, I reported on the state of the work on the new Fortran standard, and described the new features it was planned to contain [1]. Since that time a number of significant developments have taken place, and I shall describe those events which have occurred both in the American technical committee, X3J3, and at the international level.

X3J3 is a body composed of about forty-five representatives of computer hardware and software vendors, users, and academia. As its work achieves more prominence, its membership rises meeting by meeting. It is accredited to ANSI, the body which publishes final American standards, but reports directly to its parent committee, X3, which is responsible for actually adopting, or rejecting, the proposed draft standards presented to it. In these decisions, X3 tries to ensure that the proposals really do represent a consensus of those concerned. The work of X3J3 is tracked by a corresponding international group, ISO/IEC JTC1/SC22/WG5, consisting of international experts responsible for recommending that a draft standard become an international standard as well as a national one. X3J3 maintains other close contacts with the international community by having half-a-dozen foreign members.

As part of the lengthy standardization procedure, it is necessary to gain at least a two-thirds majority of X3J3 to be able to forward a proposed draft standard to X3, before that committee can consider it. Once X3 agrees to begin consideration, a period of public comment is initiated, and X3J3 is required to take note of and act on the public comments, and to resubmit a revised draft for consideration by X3. This cycle can be repeated several times.

Formally, X3J3 is currently at the fourteenth of the eighteen milestones which have to be passed before a new standard receives full international recognition, and there are still those objecting that the advances it has made have gone too far. A ballot held in January 1987 — 29 to 7 — showed that no true consensus had been achieved in the committee, and many of the dissenting votes were those of large vendors claiming to act as representatives of their users. This failure to agree was confirmed in the final ballot to forward the document to X3, held in May 1987: 26 to 9, a sufficient but not overwhelming result. A complete description of this version of

the language is given in [2], shortly to appear in a revised edition.

In its turn, X3 decided by a large majority to allow the first draft standard to go to public comment, and over 400 letters were received. Many shades of opinion were represented in these replies, but a general feeling was that the language was too complex, lacked certain existing, popular features, and was burdened by other features which were unattractive. As a result of this rather sobering experience, a number of members of X3J3 attempted to prepare plans to respond in various ways to this public sentiment, but none succeeded quickly in securing sufficient support from the committee. Finally, in September 1988, WG5, despairing of ever seeing a standard emerge from X3J3, defined exactly which changes *it* wanted to the original draft, and set its convenor a time-table to prepare a second draft conforming to those requirements. Defiantly, it renamed the proposed language Fortran 88! This action sent shock waves through the standards world, as an international committee was essentially reprimanding a national one and setting it an ultimatum. WG5's action received support from other bodies, including X3, which sent X3J3 a message reminding it of its international as well as its national obligations. In response to this pressure, X3J3 voted 24 to 9 in November 1988 to accept the principle of WG5's plan and, at a rather extraordinary meeting, passed proposals corresponding to 18 of WG5's wishes, rejected two, and continued work on the the two remaining, more difficult ones — pointers and stream I/O. Work on these was completed in February 1989 and, judging by the result of informal soundings in X3J3 and WG5, it is to be expected that the resulting text will be forwarded again this summer for both American and international public comment. Only one significant feature has been added by X3J3 to the plan adopted by WG5 — the ability to have variables of derived-types in COMMON. Two lesser additions, vector-valued subscripts and the DO...WHILE, have less impact on overall complexity, and are seen to be acceptable to WG5.

There is a definite feeling that the work is now on the home straight, and that at least some of the vendors have changed their position from outright opposition to reluctant acceptance of the inevitable.

2. Summary of Changes

At the beginning 1987, the draft described in [1] was not yet in its final state, however, no substantial changes were made before its actual publication. The main new features it contained were:

1. the concept of the obsolescence of existing redundant features such as the arithmetic-IF;
2. the concept of the deprecation both of features made redundant by new features in Fortran 8x and of storage association;
3. a new source form;
4. an alternative style of relational operators;
5. a new method of specifying the attributes of variables and constants;

6. a new method to specify the range and precision of real entities;
7. a CASE construct;
8. various new forms of DO construct;
9. optional and keyword procedure arguments;
10. recursive procedures;
11. interface blocks for defining inter-procedure information at compile-time;
12. a NAMELIST facility;
13. comprehensive array processing features;
14. various ways of controlling dynamic storage including dynamic aliasing (IDENTIFY);
15. the dynamic reduction of array bounds (SET RANGE);
16. internal procedures (an extension of statement functions);
17. derived-data types;
18. MODULES which, in addition to global data definitions, allow data abstraction;
19. many new intrinsic procedures (over 100 compared to 38 in FORTRAN 77).

At its meeting in Paris, WG5 required changes which are summarised below; the numbers in parentheses indicate, where appropriate, the voting figures on similar points at the Fortran 8x meeting held for HEP at CERN in January, 1988 (the figures are for/against/undecided).

1. remove the concept of deprecation (37/1/13 for *keeping* the concept);
2. remove SET RANGE (36/3/12);
3. remove IDENTIFY;
4. simplify specified real range and precision by replacing it by a parameterized real which maps more obviously onto the underlying hardware (10/19/22);
5. add also parameterized integer, character and logical data types, for the optional representation of short integers, multi-byte characters (15/22/14) and packed logicals, respectively;
6. remove internal procedures, and change the association rules correspondingly;
7. remove the square bracket symbols, [], from the Fortran syntax;
8. add pointers (18/22/11);

9. add the MIL-STD bit intrinsic procedures (43/0/8);
10. add significant blanks to the new source form (29/11/11);
11. add the INCLUDE statement;
12. remove user-defined elemental functions (the ability of a function written for scalars to be used automatically for array arguments, returning an array-valued result);
13. remove a new form of the DATA statement, retaining the present one;
14. change interface blocks to allow, in particular, a simple way of writing user-defined generic procedures;
15. change the syntax of array constructors to be modelled on that of existing I/O lists;
16. remove the possibility to parameterize derived-data types;
17. add stream I/O intrinsic procedures;
18. add binary, octal and hexadecimal constants and edit descriptors (47/3/1).

At its meetings since Paris, X3J3 has agreed to and passed all these proposals, some of them modified significantly, with the exception of that relating to internal procedures. In addition, it passed proposals which add vector-valued subscripts (30/10/11) and the DO...WHILE to the language, as well as another allowing structures in COMMON.

This is the current state of the work, and the one likely to be forwarded to the parent committees and to the public this summer. This is what could well become the final standard. HEP must now judge whether it is a language it can willingly live with to the end of the century and beyond.

In the final section, those features which have recently been added are described, thus complementing the previous paper.

3. New Features

3.1 Pointers

Pointers have been added to Fortran in the form of an *attribute*. The name of an object with the pointer attribute designates a descriptor of the object, rather than the object itself. A pointer may reference any dynamic object, such as an array whose actual bounds are defined at run-time, or any static object defined with the TARGET attribute. The use of the name of the pointer is, in most contexts, interpreted as a reference to the contents of the target. This is known as *automatic dereferencing*, i.e. no explicit notation using a pointer symbol is required. This means that existing code can readily be modified to use pointers, as only the specification statements need to be changed. Pointers are strongly typed, that is, they may point

only at objects of the same type and type parameters as the ones with which they themselves are defined. Simple examples are:

```
REAL, POINTER :: arrow ! may point to a real scalar
REAL, DIMENSION(:, :), POINTER :: pfeil ! may point to a 2D array
ALLOCATE(pfeil(10, 10)) ! allocate storage to pfeil
```

Sometimes, of course, it is necessary to change the value of a pointer. For this case, a *pointer assignment* is foreseen. An example is shown in

```
REAL, DIMENSION(:, :), POINTER :: a, b, c
:
ALLOCATE (a(10,10), b(10,10)) ! give actual space to a and b
:
! define a and b
:
c => a ! c now points to the array a
DO i = 1, 2
  IF (i == 2) c => b ! change target of c, no copy of data
:
! operations on c
:
END DO
:
```

Pointers may be components of a structure, and of the same type as the structure. In other words, a recursive definition is allowed which enables structures representing lists, trees and graphs to be defined. An example is

```
TYPE pixel
  REAL x, y
  INTEGER colour
  TYPE (pixel), POINTER :: next
END TYPE pixel
```

Pointers become associated with a target by execution of a pointer assignment or of an ALLOCATE statement. They become disassociated by, in general, execution of a NULLIFY statement. Storage occupied by a target may be returned to the processor by execution of a DEALLOCATE statement; this can result in a pointer being left 'dangling', and programmers must guard against their further use until they are newly associated. An intrinsic inquiry function, ASSOCIATED, indicates whether an association exists between a pointer and its target, or whether a pointer is associated at all.

Pointers may be used for static objects. In order that this should not inhibit optimization, any static object which might become a pointer target must be declared with the TARGET attribute. The compiler then knows which assumptions it can and cannot make about the references to static objects.

Whilst it is clear that these pointers, which are simple and safe to use, do not possess the full generality of those we are used to in, say, ZEBRA [4], they go a long way to helping those with less demanding applications, and to providing even HEP with a first step away from having to supplement Fortran with so many home-grown extensions.

3.2 Parameterized Data Types

The dispute about the complexity of the user-specified range and precision for real data in the first draft has led to a complete reworking of this area of the standard, such that all the intrinsic data types are now parameterized. The way this is done is best explained by examples.

Suppose a given processor gives access to two *kinds* of integers – a default kind, represented by four bytes, and a short kind represented by two bytes. The processor might designate the default kind by a *kind type value* of 4, and the short kind by a value of 2. This value can be associated with the *kind type parameter* of a given variable as shown:

```
INTEGER(KIND=4) :: i      ! same as default
INTEGER(KIND=2) :: j      ! half the size
```

Constants corresponding to these kinds may be written. The value 99 might also be written as 99_4, and 63_2 is a short constant. It is, of course, not very portable to use explicit kind type values, and they can be parameterized, in this example for instance as

```
INTEGER, PARAMETER :: short=2
INTEGER(short)      :: j
:
j = 63_short
```

A similar, optional facility is available for logical entities which, when it is available, might pack down into bits or bytes, depending on the value of the kind type parameter, and for character entities, in order to accommodate non-European languages whose characters require more than one byte for their representation.

In the case of reals, at least two kinds must be provided, corresponding to the present single- and double-precision features. The kinds can be defined in terms of a range and/or precision, allowing a reasonable measure of portability. To define an array whose elements have a range of at least $10^{\pm 99}$ and a precision of at least nine decimal digits, we might write

```
INTEGER, PARAMETER :: long = SELECTED_REAL_KIND(9, 99)
REAL(long), DIMENSION :: a(1024)
a(1) = 1.23456789_long
```

where the intrinsic function `SELECTED_REAL_KIND` returns the kind value most closely but completely corresponding to the desired precision and range.

Other intrinsic functions are available as part of this facility, in particular an enquiry function, equivalent to the one just shown, for integers, returning effectively the number of significant bits.

3.3 Other Changes

3.3.1 Bit intrinsic procedures

The 10 functions and one subroutine of MIL-STD 1753 have been incorporated into the draft; their definitions have been extended to handle integer arrays in a completely upwards-compatible fashion.

3.3.2 INCLUDE facility

The widely used INCLUDE facility has been put into the language in response to American public comment. It is actually redundant, as the MODULE/USE facility is more general, more powerful and safe. An INCLUDE line cannot reference a file containing partial statements.

3.3.3 Partial record I/O

A new ADVANCE specifier on READ and WRITE statements for formatted, sequential files allows data to be read and written without being concerned with any record structure. It is, for instance, possible to read a specified number of characters from part of a record, to remain positioned at that point in the record, and to read from the remainder with a subsequent read. At present, a single read would cause the whole record to be read, regardless of the length of the input list.

3.3.4 Binary, octal and hexadecimal constants

The standard allows binary, octal and hexadecimal constants to be used in DATA statements, and integers to be read and written in the corresponding formats:

```
INTEGER :: i, j
DATA j / Z'abcd' /
:
WRITE (unit, '(Z4)' ) i
```

3.3.5 Structures in COMMON

Subject to the conditions that the type definition contains a SEQUENCE specifier and that its components all ultimately resolve to all numeric or all character types, objects of derived-data types may be placed in COMMON blocks. The purpose of the SEQUENCE specifier is to force the compiler to lay out the components of the structure in memory as would be expected for the same sequence as defined as a list in a COMMON block under the usual rules of storage association.

3.3.6 DO...WHILE construct

The DO...WHILE has been incorporated into the language to satisfy American public opinion. It is redundant, as the DO construct and the EXIT statement provide the same functionality.

3.3.7 Vector-valued subscripts

Many vector processors have a gather/scatter capability. This can be directly accessed in Fortran 8x by use of the vector-valued subscript notation:

```

      INTEGER, DIMENSION(1024) :: i, j
      REAL, DIMENSION(1024)    :: a, b
      :
! Define i, j and a
      :
      b(j) = a(i)                ! permute the elements of a into b

```

4. Conclusion

At the last conference in this series, J.J. Russell demonstrated emphatically how necessary it has become to improve our working language in high-energy physics, showing how much effort we spend on making Fortran usable, and demonstrating the superiority of ADA in many respects [3]. I hope that Fortran 8x in its current state will go a long way to answering many, if not all, of his criticisms, at the same time providing a much smoother path forward, both for programs and programmers, than would be possible by undertaking an abrupt change of language.

ACKNOWLEDGEMENTS

I wish to thank John Reid for permission to use some advance text from our book to expedite the preparation of this article.

Bibliography

1. M. Metcalf, *Fortran 8x – the Emerging Standard*. Comp. Phys. Comm. **45**, 1–3 (1987) pp. 259–268.
2. M. Metcalf and J. Reid, *Fortran 8x Explained* (Oxford University Press, Oxford and New York, 1987).
3. J.J. Russell, *Programming Languages: Time for a Change?* Comp. Phys. Comm. **45**, 1–3 (1987) pp. 269–273.
4. J. Brun, M. Goossens and J. Zoll, *ZEBRA Users Guide*. CERN DD/EE/85–6 (1985).