The advantages of Fortran 90

by

John Reid

Abstract

Fortran 77 is the most widely used language for scientific programming. Its long-awaited revision is now called Fortran 90. It was finalized (down to the last editorial detail) on 11 April 1991, published as an ISO Standard in August 1991, and the first compiler is now on the market. This seems an appropriate moment to review its history and explain its advantages.

Submitted for publication in Computing.

Central Computing Department,
Atlas Centre,
Rutherford Appleton Laboratory,
Oxon OX11 0QX.

March 1994.

CONTENTS

1 Introduction

The scientific and engineering computing community has a huge investment in codes written in Fortran 77. It also has a huge human investment in Fortran 77 experience and familiarity. A basic assumption made by the Fortran committee X3J3 in designing a revision was that this investment must not be lost. It would have been far easier and intellectually stimulating to have designed a new language, but that was simply not a possibility. The aim was to enhance the language, using experience gained with extensions of Fortran 77 and in other languages, to

- provide greater expressive power,
- enhance safety (likelihood that errors would be detected),
- enhance regularity,
- provide extra fundamental features (such as dynamic storage),
- fix problems encountered with Fortran 77,
- exploit modern hardware better, and
- improve portability between different machine ranges.

This paper is addressed to the reader who accepts the objective of enhancing Fortran 77. Its aim is to review the history of Fortran standardization and explain the advantages of Fortran 90 over Fortran 77. Although we do make some references to other languages and to non-standard extensions of Fortran 77, we make no attempt at a comprehensive comparison.

2 History

Fortran was the first computer language ever to be standardized. The original standard of 1966 was replaced by a new standard in 1978 (Anon 1978a) and the languages have become informally known as Fortran 66 and Fortran 77 (it is not Fortran 78 because the technical content was completed in 1977). Fortran 77 was a modest revision of Fortran 66 and the ANSI committee X3J3 felt under tremendous pressure to add further features such as dynamic storage and array syntax. It therefore went immediately into a 'tutorial' mode, learning about experiences in other languages, and aimed to produce a new standard in 1982. In fact, by 1982 most of the new features had been agreed, each as a separate extension, and the task of integrating everything into a new document began. The draft was then known as 'Fortran 8*x*' and we expected that the unknown *x* would have the value 8 at most.

The basic reason for the delay was the difference of opinion between those who wished to see a large range of new features and those with more modest goals. A ballot of X3J3 held in April 1986 was 16 for and 19 against the draft of that date and led to some slimming down of the language. A second ballot held in January 1987 showed that better agreement had been reached (29-7) and a draft was issued for public comment later that year. Over 400 letters were received, representing varying shades of opinion, some welcoming the power and safety of the new features but many saying that the language was too complex and lacked certain popular extensions. The rules of X3J3 require that at this stage no change may be made without a 2/3 majority vote. This led to deadlock for two meetings. Finally, in September 1988, the ISO committee WG5, despairing of ever seeing a standard emerge from X3J3, defined exactly which changes it required for the ISO Fortran standard and set a timetable for the preparation of a second draft.

From 1988, progress was steady, although there were always about 10 no votes within X3J3 for successive versions of the whole standard, not enough to prevent acceptance by a 2/3 majority. A second draft was issued for public comment in 1989 and the 150 responses were largely favourable. A third draft in 1990 provoked only 29 letters. The new standard was finalized, down to the last editorial detail, on 11 April 1991 and was published as an ISO Standard (Anon, 1991) in August 1991. The technical content was completed in 1990, which is why the name 'Fortran 90' has been chosen.

The first compiler for the whole language is now available from the Numerical Algorithms Group Ltd (NAG, 1991). This compiler has been used to check all of the examples in this paper.

3 Language evolution

Fortran has been around for a long time and there is a huge volume of working code. To protect this investment, Fortran 90 is a proper superset of Fortran 77 – a program that conforms to Fortran 77 will conform to Fortran 90, too. Since the early deliberations, this aspect has not been controversial within the committee.

Looking to the future, the committee has given everyone a warning of possible deletions in the next revision by labelling a small number of features that have replacements in Fortran 77 as 'obsolescent'. Many groups already advise against the use of these features and again this has not been controversial within the committee. The features involved are:–

- Arithmetic `IF,`
- Noninteger `DO` index,
- DO termination other than on a `CONTINUE` or `END DO` statement,
- Branching to `END IF` from outside its block,
- Shared `DO` termination,
- Alternate return,
- `PAUSE,`
- `ASSIGN` and assigned `GO TO,` and
- Assigned `FORMAT` specifiers.

4 Array features

The fact that all Fortran 77 arrays are static is a very big deficiency. Fortran 90 contains 'automatic' arrays, created on entry to a subprogram and destroyed on return; and it contains 'allocatable' arrays whose number of subscripts (rank) is fixed but whose actual size and lifetime are fully under the programmer's control through explicit `ALLOCATE` and `DEALLOCATE` statements. The declarations in Figure 1 include an automatic array `WORK` and an allocatable array `HEAP`. Note that a stack is an adequate storage mechanism for the implementation of automatic arrays, but a heap will probably be needed for allocatable arrays.

Figure 1. Declarations of an automatic and an allocatable array.

```
SUBROUTINE X(N,A,B)
  INTEGER N, A(N), B(N)
  INTEGER WORK(N,N)
  INTEGER, ALLOCATABLE :: HEAP(:,:)
```

These two changes represent an enormous advance from Fortran 77 with its static storage. There will no longer be any need for workspace to be set up by the user of a library procedure or for the argument list to be cluttered with workspace arguments. It will now be straightforward to structure global storage according to the size of the problem at hand, and there will no longer be any need for complicated and unsafe storage management schemes within the code itself.

Arrays may be used in whole-array expressions such as

```
B + C*SIN(D)
```

The operations are performed element-by-element, that is, the sine function is applied to each element of `D`, multiplied by the corresponding element of `C`, and added to the corresponding element of `B`. The arrays must have exactly the same shape, but scalars may be intermixed freely. Array expressions may be used as actual arguments. They may be used in whole array assignments such as

```
A = B + C*SIN(D)
```

provided the left-hand side array has exactly the same shape as the expression. Note that there is scope for a computer to fully exploit multi-dimensional arrays in a statement such as this, whereas if it is rewritten in the form of nested `DO` loops, existing vectorization techniques often vectorize only the inner-most loop.

Rectangular subarrays, called ‘sections’, may be used as arrays. Examples are `A(:,7)` which is the 7-th column of `A` and `A(2:10:2,7)` which consists of components 2, 4, 6, 8, 10 of the 7-th column of `A`.

Dummy arrays may be ‘assumed-shape’ (take their shapes from the corresponding actual arguments). No longer will we need to specify the leading dimensions of arrays as separate arguments when calling library codes. For example, a call of the BLAS (Basic Linear Algebra Subroutine) `SDOT` of Lawson, Hanson, Kincaid, and Krogh (1979) to calculate the dot product of row `I` of array `A` and row `J` of array `B` takes the form

```
CALL SDOT(N,A(I,1),LDA,B(J,1),LDB)
```

where `N` is the row length, `LDA` is the leading dimension of array `A`, and `LDB` is the leading dimension of array `B`. The corresponding Fortran 90 call would need just two arguments to specify the two vectors, for example,

```
CALL DOT(A(I,1:N),B(J,1:N))
```

using the array section notation explained in the previous paragraph. Arrays may be of size zero, which will mean that we no longer have to write special-case code in case it happens. For example, the basic step of Gaussian elimination is to add a multiple of row `K` of the reduced matrix to row `I`:

```
A(I,K+1:N) = A(I,K+1:N) + AMULT*A(K,K+1:N)
```

and this will execute correctly on the last step when `K` equals `N` (it will do nothing). Functions may be array-valued. All but one of the Fortran 77 intrinsics (and a few new ones) may be called ‘elementally’ in the way `SIN` was called in the above example. The exception is `LEN`, which has become an inquiry function that always returns a scalar result; since the character lengths of the elements of an array are all the same and scalars are permitted to be mixed freely in array expressions and assignments, the difference is merely a technicality. There are many new inquiry intrinsics that return the array properties of their arguments and many new array-valued intrinsics, for example `MATMUL` for matrix multiplication, `MAXVAL` for the largest element, and `SUM` for summation. Arrays of rank one may be constructed as lists of scalars and other arrays of rank one, just as in input-output statements in Fortran 77. An example of an array constant of size 10 is

```
(/ 21.0, 2.7, (21.0,2.0,I=1,4) /)
```

There is a `RESHAPE` intrinsic function to allow arrays of other shapes to be constructed.

`WHERE` statements allow array assignment statements to be masked. For example

```
WHERE (A.GT.0) B=LOG(A)
```

causes the evaluation and assignment of logarithms only for elements that are positive. There is also a block form with an optional `ELSEWHERE` block.

The array features represent a major advance over Fortran 77. They were envisioned from the earliest days of Fortran, as evidenced by the whole-array input-output statements. It is obvious that a language intended for ‘FORmula TRANslation’ should include a notation for arrays, the more so in an era when an increasing number of computers have the hardware capability to perform operations on vectors or arrays of operands. Vendors have always been well represented on the Fortran committee and care has been taken to avoid any serious performance losses. The introduction of assumed-shape and pointer arrays has been criticized by some on performance grounds because they cannot be addressed quite so directly, but their use is not obligatory and the extra power and convenience will often be judged more important than a small performance penalty. Note also that new optimization techniques will be required to take advantage of the concise array syntax. In the longer term, we will surely regard our present reliance on `DO` loops as a primitive and obscure style of programming.

5 Parallel processing

A strong plea was made in the summer of 1983 during the X3J3 meeting at Los Alamos for features for explicit control of parallel processing to be added to the language. The committee responded by asking exactly what features were wanted, and from then on there was a consensus that it was too early to standardize explicit syntax for parallel processing in Fortran. In the event, a separate ad hoc committee called PCF (Parallel Computing in Fortran) was formed in 1987 and published some draft proposals in 1989. These were widely criticized as too complicated and the PCF committee accepted that some simplification was needed. However, it is my opinion that the latest published revision (PCF, 1991) is also too complicated. I have written a detailed critique of this revision (Reid, 1992). The slow progress of the PCF committee demonstrates that X3J3 was wise not to attempt to include such features in Fortran 90. If it had, the standard would still be incomplete.

The work has now been transferred to an ANSI committee, X3H5, that was formed in 1990 with many of the PCF committee as members. It has the task of defining a language-independent model and bindings for Fortran 77, Fortran 90, Pascal, and C. The model chosen is based on a shared-memory machine and a new program construct, the parallel construct. When a base process encounters a parallel construct, a 'team' of processes is formed to share the work that it contains. When the work is complete, the team is dissolved and the base process continues execution. Each data object that is referenced must be specified (perhaps implicitly) as either private (having a separate instance for each team member) or shared (by all team members). Within the parallel construct are work-sharing constructs that allow the work to be shared. Examples for Fortran are the parallel `DO` (the iterations may be executed in parallel by different team members) and the parallel sections (containing distinct sequences of statements that may be executed in parallel by different team members). There is an implicit synchronization of the team members at the start and end of a work-sharing construct, and there are explicit synchronization features, needed for example when a shared data object is changed by one process and accessed by another. We will not attempt to describe the features fully since work is still in progress. It is disappointing that the emphasis is on the Fortran 77 binding and no start has yet been made on the Fortran 90 binding.

Another effort at adding features for parallel execution is the High Performance Fortran Forum (HPFF), which met for the first time in Houston in January 1992. Here the starting point is Fortran 90 with directives to distribute arrays among the partitions of a distributed-memory machine.

The array features provide scope for implicit parallelization. Their design was much influenced by early experience (Flanders, 1979 and ICL, 1979) on the ICL DAP, a SIMD machine consisting of a 64x64 array of bit processors, and they are proving effective on recent hardware such as the Connection Machine of Thinking Machines Corp. (see, for example, Bailey, 1990). The parallelism is immediately apparent in a whole-array statement such as that illustrated in Section 4. It was noted there that the Fortran 77 intrinsic functions have been extended to allow them to be called for an array argument and return an array result obtained by applying the function to each array element. There are also 7 new elemental functions that manipulate real numbers (see Section 9), 11 new elemental procedures for bit processing (Section 12), an elemental function to merge two arrays under the control of a mask array, and 11 other new elemental procedures that we do not describe here.

Besides the elemental intrinsics, there are 15 functions that take one or more array arguments and produce an array result: `DOT_PRODUCT` for dot products, `MATMUL` for matrix multiplication, `TRANSPOSE` for transposition, 7 functions that perform simple operations such as summing array elements or counting the number of true elements, 2 functions to pack and unpack required array elements (specified by a mask array), a function to replicate an array, and functions for circularly and end-off shifting arrays. All these functions provide scope for significant parallel processing.

6 Derived data types

Fortran 90 permits data to be grouped into a structure. For example, the code in Figure 2 shows the declaration of a ‘type’ for the *x* and *y* coordinates of a point together with the declaration of a scalar and an array of this type. The symbol `%` is used to select a component; for example, `A%X` is the `X` component of `A` (unfortunately ‘.’ is unavailable because of its use for operators such as `.GE.`).

Figure 2. Declaration of a derived type and objects of this type.

```
TYPE POINT
  REAL X, Y
END TYPE POINT
TYPE(POINT) A, B(10,20)
```

Functions may be used to define operations on such types of data and subroutines may be used to define assignments between them. The operators may be intrinsic (for example, `+`, `*`, `.EQ.`), in which case the existing priorities are used for the new operators, or nonintrinsic (for example, `.MERGE.`), in which case the priority is maximum for unary operators and minimum for binary operators.

Derived data types provide the language with a powerful form of extensibility. It means that ordinary infix operator notation (operator between the operands) will be available for matrices, extended-precision arithmetic, interval arithmetic, and so on. We will defer showing an example until the next section since the most convenient way to program this involves the use of a module.

7 Modules

Modules are collections of data, type definitions, and procedure definitions. For example, a module for interval arithmetic is shown in Figure 3. It contains the definition of a type whose components are the lower and upper bounds of the intervals, a procedure for adding two intervals, an interface block that tells the compiler to associate this function with the operator `+`, similar code for doing other operations on intervals, and similar code for doing operations between reals and intervals.

Figure 3. A module for interval arithmetic

```
MODULE INTERVAL_ARITHMETIC

  TYPE INTERVAL
    REAL LOWER, UPPER
  END TYPE INTERVAL

  INTERFACE OPERATOR(+)
    MODULE PROCEDURE ADD_INTERVALS
  END INTERFACE
  :

CONTAINS

  FUNCTION ADD_INTERVALS(A,B)
    TYPE(INTERVAL) ADD_INTERVALS, A, B
    ADD_INTERVALS%LOWER = A%LOWER + B%LOWER
    ADD_INTERVALS%UPPER = A%UPPER + B%UPPER
  END FUNCTION ADD_INTERVALS
  :

END MODULE INTERVAL_ARITHMETIC
```

Access to this module requires a USE statement whose simplest form is

```
USE INTERVAL_ARITHMETIC
```

This will permit variables to be declared of this type and expressions and assignments to be written in the usual way but interpreted in this new way. A simple example is

```
INTERVAL = INTERVAL + REALA*INTERVALA + REALB*INTERVALAB
```

For another illustration of the power of derived types and modules, we consider the problem of calculating derivatives, often wanted in optimization calculations and when solving differential equations. We take the simple case of wanting the first and second derivatives with respect to t of the function

$$f(x,t).$$

We suppose that this is coded as in Figure 4. We define a new type, `REAL2` (Figure 5), to hold a value together with first and second derivative values and define associated operations. The operation for multiplication, shown in Figure 5, expresses the chain rule for calculating the first and second derivatives of the product of two functions whose values and first and second derivatives are known. This is placed in a module along with similar code for the other operations and associated with operators as in Figure 3. Once this has been done, we have only to insert a `USE` statement in `FUNCTION F`, declare `F` and `X` to be of type `REAL2`, and recompile. We then have a code that when given values of x and t, calculates $f(x,t)$ and its first and second derivatives with respect to t. The idea can be generalized to functions of many variables and the calculation of Jacobians and Hessians, though Griewank (1989) has shown that for efficiency some calculations should be performed in reverse order.

Figure 4. Code for $f(x,t)$.

```
FUNCTION F(X,T)
  REAL F, X, T
    :
END
```

Figure 5. Definition of the type `REAL2` and the multiply function.

```
    TYPE REAL2
      REAL VALUE, DERIV, DERIV2
    END TYPE

FUNCTION MULT(X,Y)
  TYPE(REAL2) MULT,X,Y
  MULT%VALUE  = X%VALUE*Y%VALUE
  MULT%DERIV  = X%VALUE*Y%DERIV  + X%DERIV*Y%VALUE
  MULT%DERIV2 = X%VALUE*Y%DERIV2 +      &
            2.0*X%DERIV*Y%DERIV  + X%DERIV2*Y%VALUE
END FUNCTION MULT
```

To allow for possible name clashes when accessing two or more modules, perhaps written by different people, there is a renaming facility within the `USE` statement. Also, the module may specify which entities are accessible and which are not. This allows changes to be made to the inaccessible entities in the certainty that code outside the module is not making any direct use of it.

Modules provide a safe replacement for `COMMON`. Note that the definitions are given only once. It is more general than `COMMON` in that type and procedure definitions are included. It is likely that libraries will become libraries of modules instead of libraries of procedures. Using modules will mean that silly mistakes, such as omitting an argument in a procedure call, are far more likely to be noticed at compile time by the compiler, as has already been our experience with the NAG compiler.

8 Procedures

A procedure may be called recursively provided its leading statement includes the qualifier RECURSIVE. A procedure may be internal, at one level only, to an external subprogram or to a subprogram in a module. Keyword calls, as in the input-output statements of Fortran 77, are available. The dummy argument names serve as keywords. Arguments may be omitted provided they are declared as OPTIONAL. The intrinsic function PRESENT may be used to inquire whether an optional argument is present. Dummy arguments may be declared to be IN, OUT, or INOUT. Figure 6 illustrates most of these features.

Figure 6. A recursive subroutine and a call of itself.

```
RECURSIVE SUBROUTINE CALC(LEVEL,A,B,C,FLAG)
  INTEGER, INTENT(INOUT)          :: LEVEL
  REAL, INTENT(OUT)               :: A
  REAL, INTENT(INOUT), OPTIONAL   :: B
  REAL, INTENT(OUT), OPTIONAL     :: C
  INTEGER, INTENT(OUT), OPTIONAL :: FLAG
  :
  IF(PRESENT(B))THEN
    :
  ELSE
    :
  END IF
  :
  IF(LEVEL.GT.0)CALL CALC(LEVEL-1,AA,FLAG=FL)
  :
```

Interface blocks that contain copies of the leading statements of a procedure may be used to specify the interface to an external or dummy procedure. For example, this permits keyword calls to be made to a procedure written in assembly language. An interface block may also be used to give a generic name to a set of procedures, provided they may be distinguished by the types or ranks of their arguments. This is exactly as for the specific and generic intrinsic functions in Fortran 77 and will allow a library to bundle together different precision versions of the same procedure so that the user is concerned only with the generic name. Figure 7 illustrates a generic interface to a single and a double precision version of a function that calculates the error function erf(x).

Figure 7. A generic interface block.

```
INTERFACE ERF
  FUNCTION SERF(X)
     REAL SERF, X
  END FUNCTION SERF
  FUNCTION DERF(X)
     DOUBLE PRECISION DERF, X
  END FUNCTION DERF
END INTERFACE
```

A generic reference to this function would be ERF(X); the appropriate subprogram SERF(X) or DERF(X) will be called, depending on the type of X.

9 Kind parameters

All the intrinsic types have been generalized to have a ‘kind’ parameter. This will permit processors to support short integers, very large character sets such as Japan’s Kanji, more than two precisions for real and complex, and packed logicals. Unlike Fortran 77, complex must be supported with the same set of precisions as real; there must be at least two kinds, corresponding to single and double precision. There is an intrinsic function that returns the kind value for a desired precision and exponent range. For example, the code in Figure 8 determines the kind value `SKIND` of the least precise machine representation that gives the equivalent of at least 10 significant decimals and a range of at least 10^{-99} to 10^{99}. This kind value is used to declare the real variable `A`.

Constants may be specified with the help of an underscore and an integer constant that gives the `KIND` value. Some examples are shown in Figure 9.

Figure 8. Using a named constant for a `KIND` value.

```
INTEGER, PARAMETER :: SKIND = SELECTED_REAL_KIND(10,99)
REAL(SKIND) A
```

Figure 9. Using `KIND` parameters.

```
INTEGER, PARAMETER :: LONG = SELECTED_REAL_KIND(10,99)
INTEGER, PARAMETER :: SHORT = SELECTED_INT_KIND(5)
REAL (LONG) PI
INTEGER (SHORT) ISHORT
PI = 3.141592654_LONG
ISHORT = 12_SHORT
```

Parameterization offers the same advantages for typing as it does for setting array sizes. The single statement giving a value to the constant `SKIND` may control a very large number of declarations of `REAL` and `COMPLEX` entities in a program unit. If placed in a module, it can be used throughout the whole program. The syntax `REAL*8`, `COMPLEX*16`, ..., widely used in extensions of Fortran 77, would require every declaration to be changed if a change of precision were needed.

There are many inquiry and manipulation intrinsic functions that return information on the representation or manipulate parts of a data value. They are described in terms of the model of Brown (1981) for the representation and behaviour of numbers on a processor. The model set for real `X` is determined by the parameters b (base), ν (number of digits), $e_{\max}$ (maximum exponent), and $e_{\min}$ (minimum exponent), which must be chosen by the implementor to best fit the machine. The set consists of the numbers

$$0 \quad \text{and} \quad s \times b^e \times \sum_{k=1}^{\nu} \left(f_k \times b^{-k} \right)$$

where s is +1 or –1, e is an integer in the range $e_{\min} \le e \le e_{\max}$, each f_k is an integer in the range $0 \le f \le b$, and $f_1 \neq 0$. For `X` real, the inquiry functions `RADIX(X)`, `DIGITS(X)`, `MAXEXPONENT(X)`, and `MINEXPONENT(X)` return the model parameters for the representation of `X.` No account is taken of the current value of `X`; indeed `X` is permitted to be undefined. There are five further inquiry functions that return values that could be deduced from the four parameters; for example, `HUGE(X)` is the largest real number. There are seven functions for manipulating real values in terms of the model; for example, `EXPONENT(X)` returns the exponent value e, `FRACTION(X)` returns the fractional part $X \times b^{-e}$, and `SCALE(X,I)` returns $X \times b^{I}$. These functions will permit code that works carefully with the arithmetic to be written in a portable fashion. For example, code could be written to scale the rows and columns of a matrix by powers of the base, thereby avoiding all roundoff.

10 Pointers

Data objects may be declared with the attribute `POINTER`. Such an object does not have any storage until storage is explicitly allocated for it by an `ALLOCATE` statement, or it is ‘pointer associated’ with an existing target object:

```
POINTER => TARGET
```

In the case of an array, only the rank is declared initially:

```
REAL, POINTER :: A(:,:)
```

and a shape is acquired when it is associated with a target.

As a simple example of the use of pointers, suppose we have code that performs the matrix-vector product $y = Ax$ and wish to calculate the product BCz. We might pointer associate y, A, x with r, C, z, respectively, use our code to find $r = Cz$, then pointer associate y, A, x with s, B, r and use our code to place the result we want in s. This is shown in Figure 10.

Figure 10. Use of pointers for a matrix-matrix-vector product.

```
REAL, TARGET  :: B(10,10), C(10,10), R(10), S(10), Z(10)
REAL, POINTER :: A(:,:), X(:), Y(:)
INTEGER MULT
:
DO MULT = 1, 2
  IF(MULT.EQ.1)THEN
    Y => R; A => C; X => Z  ! No data movement.
  ELSE
    Y => S; A => B; X => R  ! No data movement.
  END IF
  :  ! Compute  y = Ax
END DO
```

Components of derived types are permitted to have the pointer attribute. This permits a major application of pointers: the construction of linked lists. As a simple example, we might decide to hold a sparse vector as a chain of variables of the type shown in Figure 11, which allows us to access the entries one by one and create additional entries when necessary by an appropriate `ALLOCATE` statement. When an ordinary assignment is executed for a value of such a derived type, pointer assignment is executed for the pointer components.

Figure 11. A type for holding a sparse vector as a chain.

```
TYPE ENTRY
  REAL VALUE
  INTEGER INDEX
  TYPE(ENTRY), POINTER :: NEXT
END TYPE ENTRY
```

To avoid performance degradation for nonpointer objects, the attribute TARGET must be declared for a nonpointer object that is to be used as a target.

There is an intrinsic function that allows enquiries to be made about whether a pointer is pointer associated and whether it is pointer associated with a given target. There is a `NULLIFY` statement to disassociate a pointer.

Fortran’s treatment of pointers differs from that of most other languages in that no special syntax is required to access the target object, but is required for an assignment of the pointer itself. The reason for this choice is the expectation that in scientific and engineering applications references to the target objects are likely to occur far more frequently. It also builds on experience with dummy arguments, which are analogous since they may be associated with a variety of different actual arguments during the execution of a program.

Note that a Fortran pointer has a type and rank (number of dimensions) and that these must match those of the target. This makes for safety in the use of pointers; they cannot accidentally be used to alter the values of other sorts of data. This is often called ‘strong typing’.

11 Source form

The Fortran 77 limit of 6 characters in a name is raised to 31, and end-of-line comments following the character `!` are permitted. Underscores are allowed in names (as significant characters). These simple changes will make an enormous difference to the readability of code.

The alternative spellings `<`, `>`, `<=`, `>=`, `==`, and `/=` have been introduced for the relational operators `.LT.`, `.GT.`, `.LE.`, `.GE.`, `.EQ.`, and `.NE.` . Again, this helps readability.

There is also a free source form that is much more appropriate for work at a terminal. It attaches no particular significance to columns 1 to 6 or 72 onwards, does not allow blanks within tokens (with a few exceptions such as `END IF`), and uses a terminating `&` to indicate continuation to the next line. Lines may have length up to 132 characters, and statements may appear on up to 40 lines. Not allowing blanks within tokens will mean that it is far more likely for silly errors to be noticed by the compiler. For example, the classic case

```
DO 10 I=1.3
```

is noticed by the NAG compiler as an error.

12 Miscellaneous improvements

The only major new input-output features are `NAMELIST` and nonadvancing input-output. Nonadvancing input-output obviates the Fortran 77 insistence that records must be read as a whole and that their length be known beforehand. It is specified with `ADVANCE='NO'` on the `READ` or `WRITE` statement and inhibits the automatic advance to the next record on completion of the statement. On a `READ`, if the record contains insufficient values to satisfy the input list, an end-of-record condition results and a `SIZE=` specifier may be used to return the number of characters read.

There are two new control structures. The `CASE` construct is exemplified in Figure 12. There is also a form of the `DO` loop that does not use labels, exemplified in Figure 13. Together with the IF construct that is already present in Fortran 77, these mean that there will be far less need for labels. Labels are undesirable from the maintenance point of view since the reader must always be conscious that there may be a jump from elsewhere in the code.

Figure 12. The case construct.

```
INTEGER N
SELECT CASE(N)
  CASE(:0)     ! N negative or 0
  :
  CASE (1)     ! N = 1
  :
  CASE(5:7)    ! N = 5, 6 or 7
  :
  CASE DEFAULT ! Any other value
  :
END SELECT
```

Figure 13. The DO construct.

```
INTEGER I
OUTER: DO             ! Unlimited DO, named OUTER
  :
  DO I=1,N            ! I=1,2,...,N
    :
    IF(...)EXIT OUTER ! Possibly exit loop OUTER
    IF(...) CYCLE     ! Skip to end of the inner loop
    :
  END DO
END DO OUTER
```

The statement

```
IMPLICIT NONE
```

has been added. Its use will mean that it is far more likely for silly errors such as the transposition of two letters in a name to be noticed by the compiler.

The MIL-STD (Anon, 1978b) bit intrinsic functions have been added (and made elemental).

Binary, octal, and hexadecimal integer values are permitted in DATA statements and there are edit descriptors for them.

13 Program conformance checking

A very significant aspect of Fortran 90 is the requirement that a Fortran 90 processor must be capable of detecting certain errors in programs and of detecting the use of any syntax not specified by the standard. For Fortran 90 programs, the processors must be able to detect and report:

- the use of syntax not specified in the standard;
- the violation of a constraint of the syntax rules of Fortran 90;
- the use of unsupported kind values for the intrinsic data types;
- the use of obsolescent features;
- the use of nonFortran characters in the source text other than in character constants, character edit descriptors, and comments;
- the violation of the scope rules for names, labels, and operators; and
- the reason for rejecting a program.

Such a requirement is of tremendous benefit to the Fortran community. The Fortran 90 processors will be required to flag those constructs in programs that are not available on all Fortran 90 processors, in particular, extensions that are not available in all environments. Note, however, that this requirement does not prohibit extensions to Fortran 90; it only requires that the Fortran processor be capable of detecting extensions and report those extensions on request.

Using the NAG compiler, we illustrate the kinds of erroneous programs that must be flagged as nonconforming Fortran programs. For the first example, Figure 14 uses a `CASE` construct in which the case ranges overlap. This is a violation of one of the constraints for the syntax rules for the `CASE` construct. The diagnostic produced by the NAG compiler is shown in the figure.

Consider next the program in Figure 15. It specifies a kind value for the intrinsic type real that is not supported by the NAG compiler.

Figure 14. Required detection of an invalid CASE construct.

```
PROGRAM  ILLEGAL_CASE
  INTEGER  I

  SELECT CASE(I)
  CASE (1:10)
    CALL PROCESS_SMALL(I)
  CASE (9:20)
    CALL PROCESS_LARGE(I)
  CASE DEFAULT
    CALL PROCESS_OTHER(I)
  END SELECT
  :
END

Error: CASE(9:20) overlaps CASE(1:10) at line 7
```

Figure 15. Required detection of an invalid kind parameter.

```
PROGRAM ILLEGAL_KIND
  REAL(100)  R
  R = 1.1
  PRINT *, R
END

Error: KIND selector (100) does not specify a valid
representation method at line 2
```

Figure 16. Required detection of an invalid label.

```
PROGRAM ILLEGAL_LABEL
  ! This program references a label that is only available
  ! in the internal subroutine PROCESS_END_FILE

  READ(5, END=10)  R

CONTAINS

  SUBROUTINE  PROCESS_END_FILE
    :
10  PRINT *, 'An end of file occurred on unit 5'
    :
  END SUBROUTINE
END

Error: Missing label 10 at line 12
       detected at END@<end-of-statement>
```

As a third example, consider the program in Figure 16, which violates the scoping rules for a label in a READ statement; the intention is that the program transfer to statement 10 to handle an end-of-file condition on unit 5.

In addition, the language has introduced program statements that permit the compiler to detect many common programming errors. For example, when the

```
IMPLICIT NONE
```

statement is used, a mistyped identifier name will probably be flagged by the compiler. Where the interface to a procedure is visible to the compiler (for an internal procedure, for a module procedure, or if an interface is supplied), the compiler is able to detect an invalid reference. For example, mismatched types of actual and dummy arguments can be detected; incorrect arguments such as constants and expressions corresponding to dummy arguments specified as output arguments can also be detected.

Finally, in cases where the arrays have static bounds (or in other special cases), the compiler can detect invalid array expressions when the array shapes do not conform. The new standard thus encourages Fortran processors to detect invalid programs. But what is most impressive of the first complete implementation by NAG of Fortran 90 is that it has gone considerably beyond the requirements. It detects many erroneous programs at compile time by maintaining and developing more complete symbol tables throughout the compilation process. It uses this information to diagnose erroneous constructs. For example, it will detect that several calls to the same procedure are inconsistent in their argument pattern, as illustrated in Figure 17.

Figure 17. Detection of an invalid subroutine reference.

```
PROGRAM  ILLEGAL_CALL
  INTEGER  I
  REAL   R
  CALL  XXX(I)
  CALL  XXX(R)
END
```

```
Error: Inconsistent datatype for arg 1 in call to XXX at line 5
```

As a second example, given the program in Figure 18, the NAG compiler indicates that the vector sizes do not conform for a valid array expression.

Figure 18. Detection of an invalid array expression.

```
PROGRAM  ILLEGAL_ARRAY_EXPRESSIONS
   INTEGER  A(10), B(11), C(10)
   A = 1.0
   B = 1.0
   C = A + B
   C(2:4) = A(3:5) + B(1:5)
END
```

```
Error: Different vector lengths (10 and 11) at line 5
Error: Different vector lengths (3 and 5) at line 6
```

As a third example, consider the program in Figure 19 in which both references to the subroutine are invalid and are detected. In the first call, the third argument of `EQ_SOLVE` is an integer but the called program is expecting a real array. In the second call, the second argument is a constant, which cannot receive a value, and corresponds to a dummy argument that is specified as an output argument.

For this extra error reporting and reliability, what is the cost? The NAG compiler is the only one available at the time of writing and it is not an optimizing compiler. We have found that on a SUN SPARCstation 1, the compile time of pre-release 1 is broadly comparable with that of the SUN f77 compiler, release 1.3.1, though somewhat slower than the Edinburgh Portable Compiler epcf77, release 2.6.3. For example, a code of 3091 lines compiled in 27.6 seconds compared with 23.4 seconds for f77 and 8.7 seconds for epcf77. This provides some evidence that the cost need not be prohibitive.

Figure 19. Detection of invalid procedure references.

```
PROGRAM  ILLEGAL_CALLS

   INTERFACE
      SUBROUTINE  EQ_SOLVE(A, X, B)
         REAL  A(:,:), X(:), B(:)
      END SUBROUTINE
      SUBROUTINE  SEARCH( A, FOUND, LOCATION )
        INTEGER, INTENT(OUT) :: A(:)
        LOGICAL, INTENT(OUT) :: FOUND
        INTEGER, INTENT(OUT) :: LOCATION
      END SUBROUTINE
   END INTERFACE

   REAL  A(10,10), B(10), X(10)
   INTEGER  LIST(100)

   CALL  EQ_SOLVE(A, X, LIST)
   CALL  SEARCH( LIST, .TRUE., LOC)

END
```

```
Error: Incorrect data type for argument B (no. 3) of EQ_SOLVE
       at line 16
Error: Argument FOUND (no. 2) of SEARCH is OUT or INOUT -
       must be writable at line 18
```

14 Implementation

The last few years have seen a steady increase in the number of people attending X3J3 meetings, mainly from the vendors. Many of them have been actively working on implementations, but are unwilling to commit themselves to dates for release of compilers. We have already mentioned the NAG compiler, which was designed as a tool to aid the investigation of a NAG Fortran 90 library, but in fact is a full ISO-conforming Fortran 90 compiler. It has been used to check all the examples in the book of Metcalf and Reid (1990), and the figures in this paper. It is available on a wide range of hardware types. It is implemented in C and uses C as an intermediate language. Other compilers are likely to appear from 1992 onwards. Lahey have announced a system that aids the construction of compilers, and NAG offer a similar facility.

Performance, particularly at run time, has always been a major consideration for X3J3. If a Fortran 77 program is run under a Fortran 90 system, there is no fundamental reason for its speed differing from that achieved under a Fortran 77 processor. For example, an array must be explicitly given the attribute `POINTER` or `TARGET` if it is to be used as a pointer or accessed through a pointer, and a procedure must be declared as `RECURSIVE` if it is to be referenced recursively. It is reasonable to ask, however, what the effect will be when the program is changed to take advantage of the extra power and safety of the new features. We will concentrate on array processing since this is likely to be the most time consuming.

There is bound to be some storage-management overhead for the actual allocation and deallocation of dynamic arrays, but this will be small if such allocations are avoided within intensively executed code. Once an array is allocated, there is no reason for code that uses the array to be any less efficient than for any other array; it will be contiguous and have a known base address.

Pointer arrays and assumed-shape arrays introduce the new complication of the possibility of the elements of an array not being stored contiguously; a straightforward address calculation for an element will need an additional integer multiplication, but within many loops the compiler will avoid this calculation for every array element. A more significant overhead may occur an a machine that can access a vector held contiguously much more rapidly, but on such a machine, care must always be taken over data access if high efficiency is wanted. If a discontigous array is associated as an actual argument with a dummy argument that is always contiguous (an array that is not a pointer or of assumed-shape), ‘copy-in copy-out’ will probably occur. We anticipate that this will be a relatively rare occurrence since it really amounts to mixing a new style of actual argument with an old style of dummy argument, but it also offers an opportunity to provide contiguous storage for an intensive computation on a discontiguous array.

A further problem is associated with the maturity of optimization for Fortran 77. Some new optimization techniques will need to be developed for the new features. The only indication that we presently have of actual performance is that of the NAG compiler. It was originally designed as a development tool rather than as an optimizing compiler, but the increase of running times of converted code over using the native Fortran 77 compiler is reported by Metcalf (1992) to be in the range 30% to 60% on the Apollo, by Maine (1991) to be about 30% on the Sun, and by Metcalf (private communication) to be about 20% for different code on the Sun, all of which seem most reasonable.

15 Conclusions

We have aimed to give you a flavour of the advantages of Fortran 90. For example, the additions will make libraries much more friendly to use. Optional arguments, dynamic storage, and assumed-shape arrays will mean that the user need only specify what is truly special to the particular problem; and arguments may be grouped logically together in the form of a derived type. In one case, we found that 73 arguments could be reduced to 6. The safety features such as modules, `IMPLICIT NONE`, and the interpretation of spaces in the new source form will mean that it is far more likely for silly errors to be detected by the compiler.

For an informal description of the whole language, see Metcalf and Reid (1990). For a book that covers the principal new features and contains more examples, see Brainerd, Goldberg and Adams (1990).

If you want to have the power and safety of the new features, without the need to rewrite your code in another language, put pressure on a vendor whenever you get a chance.

Acknowledgements

I would like to thank Mike Metcalf of CERN, Geneva, for urging me to write this article and Brian Smith of the University of New Mexico, Albuquerque, for writing section 13. I would like to thank them both for carefully checking the original draft and making many helpful comments. I would also like to thank the referee for suggestions that have led to a strengthened paper.

References

Anon (1978a). ANSI X3.9-1978, ISO 1539:1980 (E). Programming language FORTRAN. ANSI 1430 Broadway, New York.

Anon (1978b). MIL-STD-1753. FORTRAN, DOD supplement to American National Standard X3.9-1978. Department of Defense, Washington DC.

Anon (1991). ISO/IEC 1539:1991, Fortran. ISO, Publications Dept, Case Postale 56, 1211 Geneva 20, Switzerland.

Bailey, J. (1990). Implementing fine-grained scientific algorithms on the Connection Machine supercomputer. Report TR90-1, Thinking Machines Corp.

Brainerd, W.S., Goldberg, C.H. and Adams, J.C. (1990). Programmer’s Guide to Fortran 90. McGraw-Hill, New York.

Brown, W.S. (1981). A simple but realistic model of floating-point computation. ACM Trans. Math. Softw., **7**, 445-480.

Flanders, P.M. (1979). Fortran extensions for a highly parallel processor. Infotech state of the art report on supercomputers, Pergamon Infotech Ltd., vol. **2**, 119-133.

Griewank, A. (1988). On automatic differentiation. In Mathematical Programming **88**, Kluwer Academic Publishers.

ICL (1979). DAP: FORTRAN language reference manual. ICL Tech. Pub. TP 6918.

Lawson, C.L., Hanson, R.J., Kincaid, D.R., and Krogh, F.T. (1979). Basic linear algebra subprograms for Fortran use. ACM Trans. Math. Softw. **5**, 308-325.

Maine, R. (1991). Review of NAG Fortran 90. To appear in Fortran Journal.

Metcalf, M. and Reid, J. (1990). Fortran 90 Explained. Oxford University Press, Oxford, New York and Tokio.

Metcalf, M. (1992). A first encounter with Fortran 90. ACM Fortran Forum, **11**, 24-32.

NAG (1991). NAGWare f90 compiler. Report of NAG Ltd, Wilkinson House, Jordan Hill Road, Oxford OX2 8DR.

PCF (1991). PCF parallel Fortran extensions. ACM Fortran Forum, **10**, 3 (special issue).

Reid, J. K. (1992). On PCF parallel Fortran extensions. ACM Fortran Forum, **11**, 17-23.