stichting
mathematisch
centrum

MC

AFDELING NUMERIEKE WISKUNDE (DEPARTMENT OF NUMERICAL MATHEMATICS)   NW 143/82   NOVEMBER

G.T. SYMM, B.A. WICHMANN, J. KOK & D.T. WINTER

GUIDELINES FOR THE DESIGN OF LARGE MODULAR SCIENTIFIC LIBRARIES IN ADA

INTERIM REPORT

Preprint

kruislaan 413   1098 SJ   amsterdam

Printed at the Mathematical Centre, 413 Kruislaan, Amsterdam.

1980 Mathematics subject classification: 69D21, 69D49, 65-Ø4

Guidelines for the design of large modular scientific libraries in ADA[*]
Interim report

by

G.T. Symm,[**] B.A. Wichmann,[**] J. Kok & D.T. Winter

ABSTRACT

This report is an interim technical report on a project, entitled "Guidelines for the design of large scientific libraries in Ada", which is being pursued jointly by the Division of Information Technology and Computing, NPL, in the UK, and the Mathematisch Centrum, Amsterdam, in the Netherlands. This project is supported by the Commission of the European Communities, for whom a final report, entitled "Guidelines for the design of large modular scientific libraries in Ada", will be produced around the end of 1983.

The authors wish to thank their colleagues, Sven Hammarling (now with NAG Limited, Oxford) and Maurice Cox, NPL, and Piet Hemker, Mathematisch Centrum, for their assistance with this project and for their contributions to this report. Thanks are also due to Geoff Miller, NPL, for valuable editorial comments.

KEY WORDS & PHRASES: *ADA Programming language, Scientific software*

*) This report will be submitted for publication elsewhere.

**) National Physical Laboratory, Teddington, Middlesex TW11 OLW, UK

This report consists of a draft of the contents and first four chapters, together with appropriate references, of the proposed final report:

GUIDELINES FOR THE DESIGN OF LARGE MODULAR SCIENTIFIC LIBRARIES IN ADA*

CONTENTS

Note that references, in this draft, to the 1980 version of the Language Reference Manual will be revised later, when the ANSI standard version becomes available. Meanwhile it may be observed that some of the notation in this draft differs from that in the 1980 Manual in anticipation of the ANSI standard.

Note also that chapters 5 to 10 and the appendices, included in the above contents, have yet to be drafted.

*Ada is a Registered Trademark of the Ada Joint Program Office - U. S. Government

1. INTRODUCTION

The new programming language Ada (United States Department of Defense, 1980) has been designed primarily for real-time computation. However, in view of the scale of effort that has been invested in its design, it is generally expected that it will also be widely used in other areas, including the important one of large-scale scientific computation.

Preliminary evaluations of the suitability of Ada for scientific computation (Cox and Hammarling, 1980; Hammarling and Wichmann, 1982) have indicated that several features of the language require careful consideration if large portable and modular scientific algorithms libraries are to be implemented successfully. Accordingly, the present project is concerned with the problems associated with the overall design and implementation of such libraries in Ada and with recommendations for their solution.

The main objective of the project is to help numerical analysts who wish to develop large libraries in Ada, comparable with the NAG FORTRAN Library (Ford et al., 1979) or the NUMAL Library in Algol 60 (Hemker, 1981), to do so in the most efficient manner, by providing them with appropriate guidelines. Without such guidelines there is, owing to the structure of the language, an ever-present risk that any library packages developed will be incompatible.

In this work, the guidelines of the Portability Subgroup of Ada-Europe (Nissen et al., 1981) are taken into account. These guidelines, which aim to aid programmers in designing and coding portable Ada programs, are extended as necessary to ensure that individually compiled modules of large scientific libraries can retain this portability whilst also being compatible with each other and with users' programs. Incidentally, the need for portability rules out the possibility of simply providing interfaces with existing libraries in other languages. The guidelines proposed here should contribute to the construction of library packages for basic computations and hence also to applications packages. Such packages should be coherent and easy-to-use and the guidelines aim to allow for their exploitation by commercial organisations in the future.

Throughout this report on the project, references to the Language Reference Manual (US Department of Defense, 1980) are abbreviated to LRM xxx, where xxx indicates chapter, chapter and section or sub-section (punctuated by full stops) or appendix, as appropriate. Multiple references are separated by commas. Details of the Language Reference Manual and all other references are gathered together, in alphabetical order of author, at the end of the work.

It is assumed in this work that the implementation of Ada supports floating-point arithmetic (LRM 3.5.7, 3.5.8), since this is invariably required in large scientific libraries. It is also assumed that the exception NUMERIC_ERROR is raised in overflow situations (cf. LRM 4.5.8).

In Chapter 2 we outline the basic problems which face designers of large modular scientific libraries in Ada. In Chapters 3 to 9 we discuss each problem area in turn and derive solutions to the problems through examples of Ada coding, the largest of which appear in Appendices. Finally we summarise our recommendations in Chapter 10.

2. THE PROBLEMS

In this chapter we outline the problems, as we see them, which face designers of large modular scientific libraries in Ada.

a) Precision

The first and most fundamental problem in the design of large scientific libraries in Ada is concerned with precision.

Every object in the language has a type, which characterises a set of values and a set of operations applicable to those values (LRM 3.2, 3.3). In particular, for floating-point computation, the language includes at least one predefined type FLOAT. An implementation may also have predefined types such as SHORT_FLOAT and LONG_FLOAT which have, respectively, substantially less or more precision than FLOAT (LRM 3.5.7). These and all other predefined identifiers are contained in the package STANDARD to which the user may be assumed to have access (LRM C). The user is also permitted to declare his own floating-point types, e.g.

```
   type REAL is digits D;
```

where D is any (positive integral) number of decimal digits supported by the implementation. In this case, the type REAL is derived by the implementation from one of the predefined types which has at least D digits of precision. Explicit type conversions are allowed between closely related types (LRM 4.6); for example, REAL(2*J) represents the integer expression 2*J in the floating-point form of the type REAL.

The user must decide how best to use these facilities and, since the rules of the language require that types must match on a procedure call (LRM 6.4.1), the choices are particularly important in the design of large numerical libraries. In such libraries, separately compiled program units must be compatible with each other, with units of other libraries and with users' units. Also intercommunication between units, of any kind, should involve as little recompilation as possible. In Ada a compilation unit (LRM 10) can be a subprogram (i.e. procedure or function) declaration or body, a package declaration or body, a generic declaration or body, or a generic instantiation. Alternatively, it can be a subunit, in which case it includes the body of a subprogram, package, task unit or generic unit declared within another compilation unit.

The main problem arises from the strong type-checking rules of the language whereby any two type definitions specify distinct types even if their descriptions are identical. Thus, for example, if

```
   type REALA is digits 6;
   type REALB is digits 6;
   A : REALA;
   B : REALB;
```

then A and B are of different types. Similarly, if one compilation unit declares

```
   type REAL is digits 10;
   X : REAL;
```

while another declares

```
    type REAL is digits 10;
    Y : REAL;
```

then X and Y are of different types and the two units are incompatible.

Ways around this difficulty are discussed in Chapter 3 of these Guidelines.

b) Basic functions

The basic mathematical functions, which, in Fortran and other languages, are denoted by SQRT, EXP, SIN, etc., are not (apart from ABS, which is covered by the reserved word **abs**) included in the Ada language and must therefore be provided in a library package. If all computations could be carried out successfully in terms of the predefined type FLOAT, this package might have a specification of the form:

```
    package MATH_FUNCTIONS is

       function SQRT(X : FLOAT) return FLOAT;
       function EXP(X : FLOAT) return FLOAT;
       function SIN(X : FLOAT) return FLOAT;

       -- etc.

    end MATH_FUNCTIONS;
```

In practice, however, types SHORT_FLOAT, LONG_FLOAT and, more generally, user-defined real types must also be accommodated. How this may be achieved is clearly dependent upon the way in which the precision problem is solved (in Chapter 3 of these Guidelines).

Problems relating to the package MATH_FUNCTIONS and its contents are discussed in Chapter 4.

c) Structured data types

Structured data types, such as COMPLEX, VECTOR and MATRIX, are not included in the Ada language and must therefore be provided in a package or packages. For example, COMPLEX may be provided as a record type, with its associated operators (cf. Wichmann, 1981), in a package of the form:

```
    package COMPLEX_OPERATORS is

       type COMPLEX is
          record
             RE,IM : REAL;
          end record;

       function "+"(X : COMPLEX) return COMPLEX;
       function "-"(X : COMPLEX) return COMPLEX;
       function "abs"(X : COMPLEX) return REAL;
       function ARG(X : COMPLEX) return REAL;
       function "+"(X,Y : COMPLEX) return COMPLEX;
       function "-"(X,Y : COMPLEX) return COMPLEX;
       function "*"(X,Y : COMPLEX) return COMPLEX;
       function "/"(X,Y : COMPLEX) return COMPLEX;

    end COMPLEX_OPERATORS;
```

where it is assumed that a type REAL is available. If it is further assumed that the basic mathematical functions, in the package MATH_FUNCTIONS, are applicable to such REAL variables, then the package body, corresponding to the above specification, could take the form:

```
with MATH_FUNCTIONS;
package body COMPLEX_OPERATORS is

   function "+"(X : COMPLEX) return COMPLEX is
   begin
      return X;
   end "+";

   function "-"(X : COMPLEX) return COMPLEX is
   begin
      return (- X.RE, - X.IM);
   end "-";

   function "abs"(X : COMPLEX) return REAL is
      A,B : REAL;
   begin
      if abs X.RE > abs X.IM then
         A := abs X.RE;
         B := abs X.IM;
      else
         A := abs X.IM;
         B := abs X.RE;
      end if;
      if A > 0.0 then
         return A * MATH_FUNCTIONS.SQRT(1.0 + (B/A)**2);
      else
         return 0.0;
      end if;
   end "abs";

   -- etc.

end COMPLEX_OPERATORS;
```

Similar packages might be provided for vectors and matrices, though we consider that these types, being useful in their own right, are best packaged separately from their associated operators.
Such packages are discussed in detail in Chapter 5.

d) Parameter passing

Ada does not permit function or procedure names as parameters in procedure calls but such information may be passed by means of generics (LRM 12). For example, a simple procedure for numerical integration (quadrature) of a function F of a single real variable X, between fixed limits of integration A and B, may have a declaration:

```
generic
   with function F(X : REAL) return REAL;
procedure QUAD(A,B : in REAL; R : out REAL);
```

Then integration of a specific function F1 with declaration:

```
function F1(X : REAL) return REAL;
```

may be achieved by means of an instantiation of the generic procedure:

```
procedure QUAD_F1 is new QUAD(F1);
```

followed by a procedure call:

```
QUAD_F1(A,B,R);
```

Issues raised by constructions of this form and other uses of generics, e.g. with types as parameters, are discussed in Chapter 6.

e) Error handling

The Ada concept of exceptions (LRM 11) provides an error handling mechanism which must be fully explored. An exception is an event that causes suspension of normal program execution. On detecting the event the corresponding exception is raised. Executing some actions, in response to the occurrence of an exception, is called handling the exception.

Exception names, other than a few predefined exceptions, such as CONSTRAINT_ERROR and NUMERIC_ERROR, are introduced by exception declarations, e.g.

```
SINGULAR : exception;
```

Exceptions can be raised by raise statements or by subprograms, blocks or language-defined operations that propagate the exceptions. When an exception occurs, control can be passed to a user-provided exception handler at the end of a unit, i.e. at the end of a block or of the body of a subprogram, package or task. This handler acts as a substitute for the remainder of that unit; so that, for example, a handler within a function body may execute a return statement on its behalf.

The handling of an exception raised during execution of a sequence of statements depends on the innermost block or body that encloses the relevant statement (LRM 11.4.1). However, if an exception occurs during the elaboration of the declarative part of a block or body, or during the elaboration of a subprogram, package or task declaration, this elaboration is abandoned. The exception is then propagated to the unit causing the elaboration, if there is one; otherwise, the program is abandoned (LRM 11.4.2). It follows that one may sometimes wish to avoid the raising of exceptions in the declarative part of a library unit, possibly by enclosing the necessary declarations in an inner block so that exceptions due to errors in input parameters can be handled in the surrounding body.

Such issues and more general questions regarding error handling in Ada are discussed in Chapter 7.

f) Working-space organisation

Working-space must be efficiently organised. In Ada, this may depend upon how arrays are stored, particularly on a machine with paging.

Storage which is no longer required may be reclaimed, to be used again, by a garbage collector. However, in Ada, the existence of a garbage collector is implementation dependent and software which relies upon it should therefore make this clear. In any case, the programmer may prefer to do his own tidying-up, e.g. in a real-time program where he may achieve better timing control by so doing (Barnes, 1982, p.253). For access types, he may use the predefined generic procedure UNCHECKED_DEALLOCATION which has the specification:

```
generic
   type OBJECT is limited private;
   type NAME is access OBJECT;
procedure UNCHECKED_DEALLOCATION(X : in out NAME);
```

with a typical instantiation of the form:

```
procedure FREE is new
   UNCHECKED_DEALLOCATION(object_type_name, access_type_name);
```

All aspects of working-space organisation are discussed in Chapter 8.

g) Real-time environment

Ada has been specifically designed for real-time computation and the needs of real-time users must therefore be taken into account. For example, they may require that a program should continue to run in all circumstances - no matter what errors may arise during its execution. This may be achieved by the inclusion of an exception handler of the form:

```
when others =>
   -- sequence of statements
```

where the sequence of statements carries out appropriate remedial action to enable the computation to continue in the event of any unforeseen error arising.

In real-time situations, such as process control, a result of a computation may be required at a particular time; the precise response moment may not be known in advance but, when it arrives, the answer must be immediate. This can affect the choice of an algorithm or the way in which it is implemented. For example, if an iterative process consists of several parts (which may run concurrently), of which the results are normally added together at the end of the process (when each part has reached a specified accuracy), it would be preferable in this case to keep a running total (with an estimate of its accuracy) to be used in the event of a rendez-vous being met before the iteration is complete.

Issues such as these are discussed in Chapter 9.

3. PRECISION

In this chapter we consider the problems concerned with the accuracy of real types in Ada, introduced in section (a) of Chapter 2. Our discussion takes the form of a series of notes, labelled alphabetically for easy reference.

a) Hardware types

The predefined types FLOAT, SHORT_FLOAT and LONG_FLOAT correspond to the hardware. Since one view of numerical packages is to consider them as additions to the hardware, one might conclude that all library software should be written in terms of these predefined types. However, this would not be a good idea for reasons of portability. The language does not state any specific accuracy for FLOAT and, since this is the name assigned if there is only one floating-point type, the actual accuracy is likely to vary considerably. Hence the use of the predefined types cannot be recommended in general. (Since the names FLOAT, SHORT_FLOAT and LONG_FLOAT are not reserved in Ada, one could possibly redeclare them, to achieve the portability that would otherwise be lacking, but this is rejected on the grounds of obscurity.)

b) Derived types

It may appear that the type compatibility rules make it very difficult to write any portable library software at all. Yet, if LONG_FLOAT is available as well as FLOAT, one can certainly imitate standard FORTRAN practice by declaring

```
type REAL is new FLOAT;
type DOUBLE is new LONG_FLOAT;
```

and writing all program units in terms of these 'derived' types. Alternatively, if SHORT_FLOAT is available, one may declare

```
type REAL is new SHORT_FLOAT;
type DOUBLE is new FLOAT;
```

and use these derived types in all program units. In either case, we have a possible solution (though not necessarily the best) to the problem of providing portable software.

c) Attributes

In Ada, most of the properties of a real type can be accessed by its 'attributes' which are defined as part of the language (LRM 3.5.8, 3.5.10). This enables one, when writing software, to anticipate the problems of moving code to another machine. For instance, an approximation may be known to be good for 10 digits but not more, in which case one can write

```
if REAL'DIGITS <= 10 then
   SIMPLE_APPROXIMATION;
else
   MORE_COMPLEX_CASE;
end if;
```

where, if the static condition is TRUE, the code for the MORE_COMPLEX_CASE (though it must be valid) need not be compiled (cf. section (e) below). Careful use of these facilities permits one to write code which is robust and numerically correct across almost all conceivable machines. In this, one is aided by the fact that the numerical properties of real types are defined in the language reference manual (LRM 4.5.8).

d) User-defined types

The contrary view to that expressed in section (a) above is that of the applications programmer who wishes (not unnaturally) to ignore details of the specific hardware in use. His concern is to program in a portable manner knowing that, for example, 10 digits of accuracy will suffice for his particular application. He therefore declares

```
type MY_REAL is digits 10;
```

whereupon the problem is that, since MY_REAL is dependent upon the application, numerical library packages (written in terms of a different real type) cannot be called directly.

One approach to this problem is the use of generics, as in the input-output system (LRM 14.4). There, for example, the output procedure PUT may be made available for MY_REAL by instantiating the generic package FLOAT_IO, which is inside the package TEXT_IO, thus:

```
with TEXT_IO;
procedure MAIN is
   ...
   package MY_IO is new TEXT_IO.FLOAT_IO(MY_REAL); use MY_IO;
   X : MY_REAL;
begin
   ...
   PUT(X);
   ...
end MAIN;
```

As a consequence of the need to instantiate the generic, this solution has some severe disadvantages. It is very unlikely that the instantiation of a generic will be a cheap operation for the compiler. At worst, it could amount to an overhead comparable with the recompilation of the instantiated body. With a large mathematical library, such an overhead might not be acceptable. Moreover, the body of the instantiated package could need to call other packages which would themselves need to be instantiated. The compiler overhead for such an activity is likely to be even greater than that for the ordinary text.

In practice, perhaps such generic packages will be precompiled for each of the relevant predefined types, such as the hardware types of section (a), and the appropriate version selected at instantiation. However, the conclusion here is that generics need to be used with care, at least within the context of a large library. The advantage of generics is that they do allow one to write a subprogram or package for any accuracy and let the user select the appropriate accuracy. Thus they are ideal for the user who is prepared to tailor a system to his own specific requirements.

e) Use of generics

On the assumption that some use is made of generics, subprograms or packages can call any low-level routines that may be provided for the hardware types by means of tests on the attributes and conversions. A simple example might be

```
generic
   type REAL is digits <>;
function SQRT(X : REAL) return REAL;       -- specification

function SQRT(X : REAL) return REAL is     -- body
begin
   if REAL'DIGITS <= FLOAT'DIGITS then
      return REAL(SQRT(FLOAT(X)));
   else
      return REAL(SQRT(LONG_FLOAT(X)));
   end if;
end SQRT;
```

Note the use of explicit conversion and the two distinct calls of the overloaded function SQRT. Of course, for a specific instantiation of this generic, a compiler should optimise the code so that no condition is tested or code produced for the other leg. Note, however, that the condition involving REAL'DIGITS is no longer static (cf. section (c) above) when REAL is a generic actual parameter (LRM 4.9).

Unfortunately, the code given here is not portable, since it has been assumed that both FLOAT and LONG_FLOAT are available, which may not be the case. Moreover, no allowance is made for the possibility that REAL'DIGITS >= LONG_FLOAT'DIGITS for which an exception could be raised.

f) Library design

For a large library, the use of other subroutines by existing routines implies that a standard type or set of types must be used for real types. Such standard types may be collected together in one package:

```
package REAL_TYPES is
   type REAL is digits ...;         -- an implementation choice
   ...
end REAL_TYPES;
```

Then a library package may operate in terms of these, for example:

```
with REAL_TYPES; use REAL_TYPES;
package MATH_FUNCTIONS is                    -- specification

   function SQRT(X : REAL) return REAL;

   -- SIN, COS, etc.

end MATH_FUNCTIONS;
```

However, if the corresponding package body is written for only the standard types, with their specified accuracy, this approach lacks generality, since there may well be a need for a function, e.g. a square root, of higher accuracy. Moreover, the textual body of SQRT may well admit any accuracy.

It is preferable therefore to implement MATH_FUNCTIONS by means of a generic package:

```
generic
   type REAL is digits <>;
package GEN_MATH_FUNCTIONS is

   function SQRT(X : REAL) return REAL;

   -- SIN, COS, etc.

end GEN_MATH_FUNCTIONS;
```

The body of this package, written for any accuracy, takes the form:

```
package body GEN_MATH_FUNCTIONS is

   function SQRT(X : REAL) return REAL is
      ...

   -- SIN, COS, etc.

end GEN_MATH_FUNCTIONS;
```

Then the library package specification above may be replaced by the instantiation:

```
package MATH_FUNCTIONS is new GEN_MATH_FUNCTIONS(REAL);
```

in which case:

```
use MATH_FUNCTIONS;
```

permits one to call, for example, SQRT(X) for X : REAL.

At the same time, this approach allows a sophisticated user, who is not satisfied with the package REAL_TYPES, to declare his own real types:

```
package MY_TYPES is
   type MY_REAL is digits ...;
   ...
end MY_TYPES;
```

and to call the basic mathematical functions for these types:

```
with MY_TYPES; use MY_TYPES;
with GEN_MATH_FUNCTIONS;
procedure MAIN is
   ...
   package MY_FUNCTIONS is new GEN_MATH_FUNCTIONS(MY_REAL);
   X,Y : MY_REAL;
begin
   ...
   Y := MY_FUNCTIONS.SQRT(X);
   ...
end MAIN;
```

4. BASIC FUNCTIONS

In this chapter the following problems concerning basic functions are identified and discussed:
- contents of a package of basic mathematical functions,
- naming of basic mathematical functions,
- method of use for user-defined types,
- efficiency of execution,
- calling sequences,
- exception handling,
- package specification.

Each of these problems is considered in a separate section.

a) Contents of a package of basic mathematical functions

Although large sets of mathematical functions are sometimes required, we propose that only Square Root and the Elementary Transcendental Functions, as given in Abramowitz and Stegun (1965) but omitting the secant and cosecant functions, should be components of a basic Mathematical Functions package (see section (b)). In this package, we also include number declarations for PI and the base of natural logarithms e (here named EXP_1), and the circular functions with period 2 instead of 2*PI (named SIN_PI, COS_PI, TAN_PI and COT_PI). All other functions can be contained in several packages of Special Mathematical or Statistical Functions.

It must be mentioned that due to the proposed structure of this Mathematical Functions package there is no need for (visible) type declarations in the package (see section (c)). In our opinion, the package obtained through an instantiation with a floating-point type FPT, chosen by the user, should provide all the basic mathematical functions for this type FPT, each of the form:

```
function MATH_FUNCTION(X : FPT) return FPT;
```

We reject a set-up in which every basic function has its specific types and subtypes, to which a user has to accommodate.

Through each instantiation the user receives a package with the familiar basic functions (as an extension of the set of arithmetic operators) for his chosen floating-point type. In this connection we note that such an instantiation is not necessary if the user-defined type is a derived type (like **type** REAL **is new** FLOAT) and an instantiation of GEN_MATH_FUNCTIONS (see section (b)) is already available for the parent type.

The package is not subdivided into smaller local packages, each containing some connected basic functions, since this would make calls of these functions too verbose.

We do not propose a separate non-generic version of the basic Mathematical Functions package. We propose instead that the program library contains at least one standard instantiation of this package with FLOAT (or the library type REAL) as generic actual parameter. (Note that a particular installation might prefer to create such an instantiation from an Ada text by expanding the generic declaration.)

Finally, we do not propose a DEGREES version of the circular functions here, as these functions would have different types for argument and result and this would not fit into a general package. The user should be warned that he does not get a proper DEGREES version by merely instantiating GEN_MATH_FUNCTIONS with the type DEGREES.

b) Naming of basic mathematical functions

The package itself should be named:

GEN_MATH_FUNCTIONS,

where 'GEN_' signifies that the package is a generic one with respect to the provided (user-defined) floating-point real type (see also section (c)). Its components should be named:

PI, EXP_1 (the base e of natural logarithms),
SQRT,
LN (and alternatively LOG_E), LOG_2, LOG_10
(logarithms for bases e, 2 and 10 respectively),
EXP, TWO_EXP, TEN_EXP
(powers of e, 2 and 10, respectively, with real exponent),
SIN, COS, TAN, COT,
SIN_PI, COS_PI, TAN_PI, COT_PI,
ARCSIN, ARCCOS, ARCTAN, ARCCOT,
SINH, COSH, TANH, COTH,
ARCSINH, ARCCOSH, ARCTANH, ARCCOTH.

Although we agree with other authors, such as Barnes (1982), that identifiers should be meaningful and that abbreviations should not be used where there is any risk of confusion, we think that for the basic mathematical functions the traditional names above are sufficiently familiar. We use the name EXP_1 rather than E, for the base of natural logarithms, on the grounds that there is a significant risk of misuse of E, e.g. when 1.0*E-1 is written instead of 1.0E-1 (assuming a mixed-type subtraction operation to be available) or when E occurs naturally in a sequence of real variables A, B, C, ...

c) Method of use for user-defined types

In accordance with section (f) of Chapter 3, the package structure should be as follows:

```
generic
   type REAL is digits <>;
package GEN_MATH_FUNCTIONS is

   function SQRT(X : REAL) return REAL;

   -- LN, EXP, etc.

end GEN_MATH_FUNCTIONS;
```

Then the package may be made available for any user-defined floating-point type, and also for the standard types FLOAT, SHORT_FLOAT and LONG_FLOAT (if present) with implementation-dependent accuracies, by an instantiation of the package for the type concerned; for example:

```
type REAL_6 is digits 6;
package MATH_FUNCTIONS_6 is new GEN_MATH_FUNCTIONS(REAL_6);

-- and for the standard type FLOAT:

package STD_MATH_FUNCTIONS is new GEN_MATH_FUNCTIONS(FLOAT);
```

(For completeness we remark that the program unit containing such an instantiation must include GEN_MATH_FUNCTIONS in its context specification.) For derived types, the package is automatically available from the parent type.

No allowance is made here for mixed-type expressions, as when a specification like

```
function SQRT(A : AREA) return LENGTH;
```

is needed. We assume that any such application will be effected by the user by means of type conversions or overloadings.

Finally we remark that it is perfectly acceptable for every instantiation to deliver the same numbers PI and EXP_1 (since they do not depend upon the generic actual parameter).

d) Efficiency of execution

When writing an Ada source text suitable for calculating values of some basic function for every possible accuracy, the following problems are faced:

- Whatever the machine arithmetic, the algorithm executed must deliver values as specified with maximal accuracy if the argument is inside its range. In agreement with the recommendations of the Ada-Europe Portability Group (Nissen et al., 1981), algorithms must be given for accuracies ranging from **digits** 5 up to **digits** 10 at least, but in the present context we propose an extension of this requirement up to **digits** 35.

  The exception SIGNIFICANCE_ERROR should be raised for calls when the argument cannot be used for calculating the value of the basic function with useful accuracy (e.g. for a call of SIN(10.0 ** REAL'DIGITS)). A problem here is that the function body cannot be made aware that the user (the function call) expects a smaller precision than normally, as would be the case if the type provided for the function result had a less stringent accuracy constraint than the type for the parameter. Here all functions have the same floating-point type for parameter(s) and function result. The (arbitrary) solution is that SIGNIFICANCE_ERROR is raised only if more than a specified number of digits will be lost. The alternative, restricting calls of the functions SIN, COS, TAN and COT to arguments in [- 2*PI, + 2*PI], is not supported.
- Algorithms may have many branches conditional upon the number of bits of the mantissa of model numbers (LRM 3.5.8) (and perhaps also the machine mantissa, machine exponent and other machine properties).
- Expressions must be built by the elementary operators only, though some basic functions may call other (more basic) ones from the same package.
- If some branching depends on the value of an argument then it should be distinctly separated from branching which depends on attributes of the generic type. In this way optimising compilers will not be prevented from deleting dead branches.
- The standard type FLOAT cannot be used inside the packages for local declarations and calculations, as this might imply an undesirable loss of accuracy in the final results. Alternatively, it might signify a waste of computer time if FLOAT is much more accurate than necessary. The algorithm might use different approximations for different accuracy constraints. For this reason we advise that branching of algorithms is not by the MACHINE_MANTISSA attribute but by the DIGITS or the MANTISSA attribute.

- As static expressions in floating-point type definitions cannot depend on attributes of the generic actual parameter (LRM 4.9), it is not possible to make a local floating-point type definition with a (slightly) larger accuracy for performing the internal calculations. All algorithms for basic functions must simply deliver the best results possible using the user-supplied floating-point type. If this user-supplied type has unexpected additional constraints, then an exception (CONSTRAINT_ERROR) will be raised upon violation. This exception can also be raised in the package body (elaborated upon instantiation) if the user-defined type is unfit for any calculation at all.
- In the same way static expressions in fixed-point type definitions cannot depend on attributes of the generic actual parameter. So the idea of Wichmann (1981) of using local fixed-point arithmetic for evaluating polynomials cannot apply here, because the appropriate fixed-point types cannot be defined (unless the types are declared inside the different branches). Besides, it will be uncertain whether a fixed-point type with as large a mantissa as that of the floating-point type is supported.
- No exception occurring in intermediate calculations should be propagated to the user's call (provided that the final result would not be exceptional). Only when the final result is exceptional, due to a bad argument of the function call, should an appropriate exception be raised (see section (f)).
- Program units using the basic Mathematical Functions package should not each make their own instantiation of GEN_MATH_FUNCTIONS, as this might imply that several copies are made. Consider for example:

```
generic
   type REAL is digits <>;
package GEN_CHOLESKY is
   type SYMMATRIX is array(INTEGER range <>) of REAL;
   procedure CHOLESKY_DECOMPOSITION(MAT : in out SYMMATRIX);
end CHOLESKY;

with GEN_MATH_FUNCTIONS;
package body GEN_CHOLESKY is

   package MATH_FUNCTIONS is
      new GEN_MATH_FUNCTIONS(REAL);
   use MATH_FUNCTIONS;

   procedure CHOLESKY_DECOMPOSITION(MAT : in out SYMMATRIX) is

      -- Local declarations

   begin
      DECOMPOSE_MAT;
   end CHOLESKY_DECOMPOSITION;

end CHOLESKY;
```

Such a package, which itself must be instantiated, would require an instantiation of the basic Mathematical Functions package and so would all other similar numeric packages.

A solution might be that a numeric package (in the above and following examples for the Cholesky decomposition of symmetric positive-definite matrices, which needs the SQRT function) is given as a generic package with, as generic parameters (besides the user-supplied floating-point type), those basic mathematical functions which it uses. These generic subprogram parameters must be declared with themselves as defaults, e.g.

```
generic
   type REAL is digits <>;
   with function SQRT(X : in REAL) return REAL is <>;
      -- visible through a use clause
package GEN_CHOLESKY is
   type SYMMATRIX is array(INTEGER range <>) of REAL;
   procedure CHOLESKY_DECOMPOSITION(MAT : in out SYMMATRIX);
end GEN_CHOLESKY;
```

Such a generic package can be used in the following way:

```
with GEN_MATH_FUNCTIONS, REAL_TYPES; use REAL_TYPES;
with GEN_CHOLESKY; -- and other numeric packages, etc.
procedure MAIN is

   -- Instantiations:

   package MATH_FUNCTIONS is
      new GEN_MATH_FUNCTIONS(REAL);
   use MATH_FUNCTIONS;

   package MY_CHOLESKY is new GEN_CHOLESKY(REAL);

      -- Note that the name SQRT is visible and that SQRT
      -- can be used as the default for the generic actual
      -- parameter, as it has the correct subprogram
      -- specification.

   -- etc.

begin
   MAIN_PROGRAM_STATEMENTS;
end MAIN;
```

e) Calling sequences

Assuming the availability of the instantiation:

```
type REAL_6 is digits 6; -- as an example
package MATH_FUNCTIONS_6 is
   new GEN_MATH_FUNCTIONS(REAL_6);
```

and the use clause:

```
use MATH_FUNCTIONS_6;
```

it follows from the full declarations given in section (g), below, that each of the basic mathematical functions can be called, taking SQRT as an example, in each of the following ways:

```
MATH_FUNCTIONS_6.SQRT(REAL_6_EXPRESSION) -- as a primary

SQRT(REAL_6_EXPRESSION) -- when the component SQRT of the
                        -- package is visible

SQRT(X => REAL_6_EXPRESSION) -- using the name of the
                             -- formal parameter.
```

The declarations of ARCTAN and ARCCOT allow a particular function call for arguments close to INFINITY. Their declarations read:

```
function ARCTAN(X : REAL; Y : REAL := 1.0) return REAL;
function ARCCOT(X : REAL; Y : REAL := 1.0) return REAL;
```

These are the only basic functions with a default (second) parameter, to the effect that a call:

```
ARCTAN(REAL_EXPRESSION)
```

delivers the normal arctangent value in the range [- PI/2, PI/2], whereas:

```
ARCTAN(REAL_EXPR1, REAL_EXPR2)
```

delivers the angle between the X-axis and the radius vector of the Cartesian point (REAL_EXPR2, REAL_EXPR1) (note the different orders of the coordinates and the parameters of ARCTAN) lying in the range (- PI, PI].

f) Exception handling

Possible exceptions are:

```
NUMERIC_ERROR,
ARGUMENT_ERROR,
CONSTRAINT_ERROR,
SIGNIFICANCE_ERROR,
(PROGRAM_ERROR).
```

We propose that an exception is raised if an algorithm fails to deliver the required result, but only if the final result itself would be exceptional. In most cases the exception that is raised by the machine hardware (usually NUMERIC_ERROR or CONSTRAINT_ERROR) can be propagated, but it is allowed that a basic function handles these exceptions and raises one of the other exceptions as the case may be. More specifically, if the hardware does not raise NUMERIC_ERROR but returns special values, then the function body should not raise an exception, as it might be the user's wish to continue the calculations with these special values. This may be compared with the IEEE recommendations for binary floating-point arithmetic (IEEE, 1981): they advise that exceptions (like invalid operations, division by zero, overflow, underflow) must be detected by the hardware, but that the user should have the means to enable and disable the corresponding traps.

As has been stated in section (d), SIGNIFICANCE_ERROR should be raised when the argument is insufficiently accurate to permit computation of accurate results. No guidelines are offered in respect of certain special exceptions, like the raising of PROGRAM_ERROR if storage is exhausted when instantiating the generic package or calling one of its constituents.

g) Package specification

To conclude this chapter, we give here the complete generic package declaration:

```
-- ------------------------------------------------------------------
generic
   type REAL is digits <>;
package GEN_MATH_FUNCTIONS is
   -----------------------------------------------------------------
   -- Declare constants.                                           --
   -----------------------------------------------------------------
   PI : constant := 3.1415_92653_58979_32384_62643_38327_95029;
   EXP_1 : constant := 2.7182_81828_45904_52353_60287_47135_26625;
   -----------------------------------------------------------------
   -- Declare the basic mathematical functions.                    --
   -----------------------------------------------------------------
   function SQRT(X : in REAL) return REAL;
   function LN(X : in REAL) return REAL;
   function LOG_E(X : in REAL) return REAL renames LN;
   function LOG_2(X : in REAL) return REAL;
   function LOG_10(X : in REAL) return REAL;
   function EXP(X : in REAL) return REAL;
   function TWO_EXP(X : in REAL) return REAL;
   function TEN_EXP(X : in REAL) return REAL;
   function SIN(X : in REAL) return REAL;
   function COS(X : in REAL) return REAL;
   function TAN(X : in REAL) return REAL;
   function COT(X : in REAL) return REAL;
   function SIN_PI(X : in REAL) return REAL;
   function COS_PI(X : in REAL) return REAL;
   function TAN_PI(X : in REAL) return REAL;
   function COT_PI(X : in REAL) return REAL;
   function ARCSIN(X : in REAL) return REAL;
   function ARCCOS(X : in REAL) return REAL;
   function ARCTAN(X : in REAL; Y : in REAL := 1.0) return REAL;
   function ARCCOT(X : in REAL; Y : in REAL := 1.0) return REAL;
   function SINH(X : in REAL) return REAL;
   function COSH(X : in REAL) return REAL;
   function TANH(X : in REAL) return REAL;
   function COTH(X : in REAL) return REAL;
   function ARCSINH(X : in REAL) return REAL;
   function ARCCOSH(X : in REAL) return REAL;
   function ARCTANH(X : in REAL) return REAL;
   function ARCCOTH(X : in REAL) return REAL;
   -----------------------------------------------------------------
   -- Declare exceptions.                                          --
   -----------------------------------------------------------------
   ARGUMENT_ERROR, SIGNIFICANCE_ERROR : exception;
   -----------------------------------------------------------------
end GEN_MATH_FUNCTIONS;
-- ------------------------------------------------------------------
```

For the package body, guidelines about the delivered accuracy and the raising of exceptions are given in sections (d) and (f). No error messages should be issued (and the package body should not itself instantiate one of the I/O packages). We advise that all the program components of the package body are given as body stubs with separate subunits. An example of a package body is offered in Appendix 1.

REFERENCES

Abramowitz, M. and Stegun, I.A., eds. _Handbook of mathematical functions_, Dover, New York, 1965.

Barnes, J.G.P. _Programming in Ada_, Addison-Wesley, London, 1982.

Cox, M.G. and Hammarling, S.J. Evaluation of the language Ada for use in numerical computations. _NPL Report_ DNACS 30/80, July 1980.

Ford, B., Bentley, J., Du Croz, J.J. and Hague, S.J. The NAG Library "machine". _Software Pract. Exper._, 1979, _9_, 56-72.

Hammarling, S.J. and Wichmann, B.A. Numerical packages in Ada. _Proceedings of the 1981 IFIP WG 2.7 workshop on the relationship between numerical computation and programming languages_, held at Boulder, Colorado, August 1981, edited by J.K. Reid, North Holland, Amsterdam, 1982.

Hemker, P.W., ed. _NUMAL, a library of numerical procedures in Algol 60_, Mathematisch Centrum, Amsterdam, 1981.

IEEE. A proposed standard for binary floating-point arithmetic. _Computer_, 1981, _14_(3), 51-62.

Nissen, J.C.D., Wallis, P., Wichmann, B.A. and others. Ada-Europe guidelines for the portability of Ada programs. _NPL Report_ DNACS 52/81, November 1981.

United States Department of Defense. _Reference manual for the Ada programming language_, July 1980.

Wichmann, B.A. Tutorial material on the real data-types in Ada. _Final Technical Report_, NTIS No. AD-A103482/6, NPL, January, 1981.

36463

[illegible]