The Numerics Annex and Related Material

K. W. Dritz*
Argonne National Laboratory
Argonne, IL 60439

The features of Ada 9X that support numerically oriented applications are distributed over several parts of the (draft) Ada 9X reference manual [7]. Some of them are collected in the Numerics Annex, which, like the other special-needs annexes, is entirely optional; some appear in the core or in other mandatory annexes.

The basic concepts underlying the real (i.e., floating-point and fixed-point) types are covered in Chapters 3 and 4 of the core and will not be discussed here. These chapters no longer include the details of a model of real arithmetic or the accuracy requirements for predefined arithmetic operations based on the model. Indeed, there are no accuracy requirements, unless the Numerics Annex is supported. The model and the accuracy requirements are discussed there.

Included in Annex A (Core Language-Defined Attributes) are the definitions of the attributes of real types. Those that relate intimately to the accuracy of the predefined arithmetic operations are left implementation defined in Annex A and are revisited in the Numerics Annex.

Annex C (Predefined Language Environment) includes the specifications of a generic package of elementary functions and a package that defines types and operations related to random number generation. By virtue of their placement in Annex C, these packages must be provided by all implementations. This requirement is appropriate because of their utility in a wide variety of applications, including some that may not be numerically intensive. For example, square roots and logarithms are useful in encoding and decoding, and random numbers are indispensable to simulations.

A package defining Fortran-compatible types is included in Annex M (Interface to Other Languages), along with a discussion of the uses of the interfacing pragmas in a Fortran context. C and COBOL are also covered in this annex. Interface support for these languages is, of course, optional; if provided, it must conform with the specifications of this annex.

Annex K is the Numerics Annex. It introduces a pair of user-selectable modes, *strict* and *relaxed*, pertaining to the required accuracy of the predefined

*This chapter was written with financial support from Intermetrics, Inc., to Argonne National Laboratory under proposal No. P-91122.

arithmetic operations. The model of real arithmetic and the accuracy requirements based on it apply only in the strict mode; no accuracy requirements apply in the relaxed mode, or if the Numerics Annex is not implemented. The Numerics Annex also specifies generic packages that define a complex type and arithmetic operations, complex elementary functions, and complex I/O operations, and it recommends that the Fortran and C interfaces be provided if those languages are widely supported in the target environment. Finally, it includes strict-mode requirements for the accuracy or performance of appropriate operations provided in the predefined numerical packages.

In the remainder of this chapter, we present an overview of the numerical features of Ada 9X. With one exception, we merely discuss the features, rather than displaying package specifications, etc. Readers needing further detail are urged to consult the draft reference manual and the rationale [8].

1 Numerical Packages in the Predefined Language Environment

All the predefined numerical packages—the mandatory ones of Annex C as well as the optional ones of Annex K—are children of a package called `Numerics`, which is itself a child of the package `Ada`. `Ada.Numerics` exists primarily to define an exception, `Argument_Error`, that can be raised by operations in several different numerical packages to signal that a parameter or operand is outside the domain of the corresponding mathematical function. In addition, `Ada.Numerics` defines as named numbers the two most common mathematical constants, `Pi` and `e`.

1.1 Elementary Functions

The generic package `Ada.Numerics.Generic_Elementary_Functions` provides the most common elementary functions (`Sqrt`, `Log`, and `Exp`; the forward trigonometric functions `Sin`, `Cos`, `Tan`, and `Cot`; the inverse trigonometric functions `Arcsin`, `Arccos`, `Arctan`, and `Arccot`; the forward hyperbolic functions `Sinh`, `Cosh`, `Tanh`, and `Coth`; and the inverse hyperbolic functions `Arcsinh`, `Arccosh`, `Arctanh`, and `Arccoth`). In addition, the exponentiation operator (`"**"`) is provided for floating-point exponents.

Some of the functions are overloaded. Thus, natural logarithms and logarithms to an aribtrary base are both provided; likewise, the trigonometric functions are provided both for the natural cycle of 2π and for user-specified cycles.

The Ada 9X version of these functions is based on ISO/IEC DIS 11430 (for Ada 83), differing from it in two minor ways: the use of child packages for better hierarchical structuring, and the use of the unconstrained subtype of the generic formal parameter as the parameter and result types of all the functions. The latter improvement is intended to increase the chances of receiving a meaningful result from the evaluation of an elementary function reference when the generic formal parameter happens to be a subtype with range constraints. The Ada 9X feature underlying this improvement—use of the `Base` attribute other than in a prefix for another attribute—also benefits implementations of the elementary functions by allowing for the declaration of working variables with the precision, but not the range, of the generic formal parameter. In fact, implementations are no longer allowed to restrict the generic actual parameter to be an unconstrained subtype, as they are in the aforementioned draft international standard.

With the appropriate changes implied by the above discussion, the account of the proposed secondary standard [2] in the *Ada Yearbook 1991* is relevant also to the version contained in Ada 9X.[1]

1.2 Random Numbers

A basic capability for generating random numbers is provided by the package `Ada.Numerics.Random_Numbers`, whose specification is shown in Figure 1.

Objects of type `Generator` represent generators of random numbers represented in floating point and uniformly distributed on the semi-open interval $[0.0, 1.0)$. One obtains the "next" random number from the sequence produced by a given generator by invoking the `Random` function with that generator as the actual parameter. In the simplest applications, nothing more is required; the rest of the package provides a variety of services that may be needed by advanced applications.

The repeatable behavior expected during program development is obtained by default. Once a program becomes operational, unique sequences of random numbers can be obtained in each run by invoking the `Reset` operation on a generator before using it to generate any random numbers. When invoked without an `Initiator` parameter, this operation resets the state of a generator to one that depends in an implementation-dependent way on the current date and time.

Tasking applications that require separate sequences of random numbers in each of several tasks can declare generator objects in the relevant tasks. In this case, separate sequences are obtained by performing the `Reset` operation (*with* a unique integer-valued `Initiator` parameter) on the generator in each task. By using the same set of initiator values on subsequent runs, the same set of separate sequences will be repeated. Resetting with an initiator is defined to change the state of the generator to one that depends in an implementation-dependent way on the value of the initiator. The specifications provide that, if the generator period is long enough, each initiator value shall initiate a distinct subsequence

[1] Subsequent to the publication of the 1991 yearbook, the names of the formal parameters of the exponentiation operator were changed from `X` and `Y` to `Left` and `Right` so as to agree with the Ada 9X version, where the latter names are absolutely necessary. Also added was the explicit permission for implementations to restrict the generic actual parameter.

```
package Ada.Numerics.Random_Numbers is

   type Generator is limited private;

   Number_Of_Seed_Components : constant := implementation-defined;
   type Seed is array (1 .. Number_Of_Seed_Components) of Integer;
   Seed_Error : exception;
   subtype Uniformlly_Distributed is Float range 0.0 .. 1.0;

   function Random (Gen : Generator) return Uniformly_Distributed;

   procedure Get_Seed (Gen   : in  Generator;
                       Value : out Seed);
   procedure Set_Seed (Gen   : in  Generator;
                       Value : in  Seed);

   procedure Reset (Gen       : in Generator;
                    Initiator : in Integer);
   procedure Reset (Gen       : in Generator);

private

   -- not specified

end Ada.Numerics.Random_Numbers;
```

Figure 1: Specification of the Ada 9X random numbers package

of the full period that does not overlap with any other such subsequence in a practical sense; if that is not possible, then the resulting state is, at least, a rapidly varying function of the initiator value.

It is customary, when multiple generators are provided, to equate a generator with its "seed"—that is, its changeable state. A different approach has been chosen for Ada 9X, however, so as to allow the `Random` operation to be realized as a function and to make it impossible to corrupt the state between calls to `Random` (or, alternatively, to make it unnecessary to validate the state at each such call, in case it has been corrupted). These benefits are achieved partly by making the `Generator` type private. Since the mode of the generator parameter in all the operations on generators is `in`, a level of indirection is required to access the actual state information; consequently, the `Generator` type is limited in addition to private.

Nevertheless, there is a need in some applications to obtain and maybe even set the state of a generator. For example, a long run may be checkpointed and restarted later. Or, it might be desirable to know the actual state of a generator just prior to the appearance of unexplained behavior. Finally, for experimentation it might be desirable to force the generator into a given state. For all these purposes, we provide also a type called `Seed` whose values represent actual generator states. The `Get_Seed` operation obtains the current state of a given generator and delivers it as a value of type `Seed`; `Set_Seed` changes the state of a given generator to that corresponding to the seed value supplied. The type `Seed` is defined as an array of integers for portability, the number of them being given by the named number `Number_Of_Seed_Components`. This does not restrict the implementation of generators to use the same representation internally; the only requirement is that each distinct internal state must be capable of being represented as a unique vector of integers, the mapping being completely implementation defined. (It is possible that some values of the type `Seed` will not correspond to generator states. If such a value is passed to `Set_Seed`, that operation will raise the `Seed_Error` exception. Note that calls to `Set_Seed` are the only places where validation of the generator state is required. Such calls will be infrequent or absent.)

We do not prescribe a particular random number generation algorithm, the intent being to allow any respectable algorithm to be used. The Numerics Annex does specify some fairly stringent performance requirements for the strict mode, but these will be satisfied by good examples of congruential generators [10] (for which the internal state is usually a single integer or floating-point number), the phenomenally long period Fibonacci generators [9] (for which the internal state is usually a modest array of integer or floating-point components, plus a carry or borrow bit), or combination generators [4, 11] (for which the internal state is usually a small array of integers). Since no one algorithm is best for all applications, implementations are required to document the algorithm used, so that the user can judge its suitability for the application at hand.

1.3 Complex Arithmetic

Three library units having to do with complex arithmetic are defined in the Numerics Annex.

1.3.1 Complex Types and Arithmetic Operations

The predefined generic child package `Ada.Numerics.Generic_Complex_Types` defines a complex type and all the expected arithmetic operations on the type. It has one generic formal parameter, `Real`, which is a floating-point subtype. The type `Complex` that it defines is a record type reflecting a Cartesian representation of complex values; its components, `Re` and `Im`, have the precision of `Real`.

Several features of the complex types package in Ada 9X are somewhat unconventional:

- We have chosen to make `Complex` a visible type, rather than enforcing an abstraction, so that "complex literals" can be composed using aggregates, as in `(3.0, 5.0)`.

- The components of `Complex` are of the unconstrained subtype of `Real`, `Real'Base`, rather than of the subtype `Real` itself. Thus, a range constraint inherited from the generic actual parameter is effectively ignored. The motivation for this design is the observation that there is no guarantee that the components of the result of an arbitrary complex arithmetic operation belong to the same range as those of its operands. This provision thus increases the likelihood of being able to deliver the result of a complex arithmetic operation. For the same reason, real-valued operations on complex values, like `Modulus`, are defined to return results of the unconstrained subtype of `Real`.

- A pure-imaginary type, `Imaginary`, is also defined, along with constants `i` and `j` of that type and all the expected operations on the type. Unlike `Complex`, `Imaginary` *is* a private type; its full type declaration shows it to be derived from the unconstrained subtype of `Real`. We have chosen to make it private primarily to suppress implicit conversions of real literals to the type as well as to suppress implicit arithmetic operations on the type, some of which would be incorrect (for example, an implicitly declared multiplication operation would yield an imaginary, rather than a real, result). Arithmetic operations on appropriate combinations of `Real'Base`, `Imaginary`, and `Complex` are provided, one reason being to allow "complex literals" to be expressed in an alternative form—e.g., `3.0 + 5.0*i`.

In addition to the usual arithmetic operators, the provided operations include common functions such as complex conjugation, component selectors, Cartesian composition functions, polar metric functions (`Modulus` and `Argument`), and polar composition functions.

`Ada.Numerics.Generic_Complex_Types` is based on the generic package of complex types and arithmetic operations proposed for standardization (for Ada 83) by the WG9 Numerics Rapporteur Group.[2]

1.3.2 Complex Elementary Functions

The predefined generic child package `Ada.Numerics.Generic_Complex_Elementary_Functions`, similarly based on a proposed secondary standard (for Ada 83), provides complex analogs of the real elementary functions. In addition, an overloading of `Exp` for the pure-imaginary type is provided, which allows complex values to be composed from polar components in the alternative form of `Rho * Exp(i * Theta)`.

[2]The proposed secondary standards for complex arithmetic were changed in 1993 to maintain compatibility with the versions included in Ada 9X. Thus, the rationale [6] published in the *Ada Yearbook 1993* is now somewhat out of date.

This generic package has a generic formal package parameter. It is intended to be instantiated with an instance of `Ada.Numerics.Generic_Complex_Types`.

1.3.3 Complex I/O

The predefined generic child package `Ada.Text_IO.Complex_IO`, which has no counterpart in the proposed secondary Ada 83 standards for complex arithmetic, provides complex analogs of the procedures in `Ada.Text_IO.Float_IO`. It, too, is intended to be instantiated with an instance of `Ada.Numerics.Generic_Complex_Types`.

The form of output to a file is that of parenthesized aggregate notation, with the real and imaginary parts both formatted according to the specified or defaulted values of `Fore`, `Aft`, and `Exp`. Output to a string is similar, but it uses a value of zero for the `Fore` of the real part and a value for the `Fore` of the imaginary part that completely fills the string. The effect is that the left parenthesis, real part, and comma are left justified in the given string, while the imaginary part and right parenthesis are right justified. This behavior results in optimal use of the available space when the real and imaginary parts of the value to be transmitted have widely disparate magnitudes.

When reading from a file specifies a `Width` of zero, the expected form of the input is parenthesized aggregate notation, with blanks, line terminators, and page terminators freely allowed before each of its components. When the `Width` is nonzero, exactly `Width` characters, or the characters up to the end of the current line, whichever comes first, are consumed; the complex value may be punctuated in parenthesized aggregate notation, or the parentheses and comma may be omitted, provided that the real and imaginary parts are separated by at least one blank. The latter form is allowed (with a nonzero `Width`) for compatibility with existing data files produced by Fortran programs.

1.4 Fortran Interface

Because much mathematical software has been (and continues to be) written in Fortran, particularly libraries of various classes of functions, the combination of pieces of Fortran and Ada programs into a complete numeric application can ease the burden of the programmer who chooses to use Ada as the primary development medium for a numeric application.

The combination of program pieces written in diverse languages has been difficult in the past because each language (and sometimes each language processor) has its own conventions regarding the mapping of data, the addressing of parameters, etc. The purpose of the Fortran interface is to facilitate the creation of such combinations. To this end, the predefined child package `Ada.Interfaces.Fortran` defines a set of types, analogous to the Fortran intrinsic types, that are intended to have the same representation they have in

Fortran. Objects of these types can therefore be passed directly to, or received from, Fortran subprograms seamlessly.

Along with the interface package, Fortran interface support includes implementation of a convention identifier of `Fortran` in the `Import` and `Export` pragmas and, for the specified kinds of types, in the `Convention` pragma. The `Import` and `Export` pragmas arrange for the appropriate link name to be used for an external reference, and they cause the parameter addressing conventions of the foreign language to be observed. They can also be used to equate a Fortran named common block with an object of an appropriate Ada record type declared in a library package. The `Convention` pragma would need to be specified for the type of such a record object, to guarantee that it is mapped the same as in Fortran. The latter pragma can also be used to indicate that objects of a record type passed to, or received from, a Fortran 90 routine should be mapped as in Fortran 90 and that an access type should be represented as the corresponding pointer type in Fortran 90.

If several implementations of Fortran are provided in the target environment, there can be a uniquely named child package of `Ada.Interfaces` corresponding to each, along with a corresponding convention identifier for use in the interfacing pragmas. A Fortran interface package can declare types in addition to those required by the language definition. For example, an interface package for an implementation of Fortran 77 might declare additional types named `Integer_Star_2`, `Integer_Star_4`, `Logical_Star_1`, etc., while one for an implementation of Fortran 90 might declare types like `Real_Kind_0`, `Real_Kind_1`, `Character_Kind_1`, etc., to match the Fortran types `Integer*2`, `Character (Kind=1)`, and so forth.

2 Performance Requirements

One of the roles of the special-needs annexes is to impose performance requirements on features described elsewhere in the language but pertinent to the special need covered by an annex. In this spirit, the Numerics Annex includes accuracy requirements for the predefined arithmetic operations of real types defined in the core and for the elementary functions defined in Annex C, and it includes statistical and other performance requirements relevant to the random number generator defined in Annex C. It also specifies accuracy requirements for the complex arithmetic operations and complex elementary functions defined elsewhere in the Numerics Annex.

2.1 Accuracy Modes

The Numerics Annex defines a pair of user-selectable modes, strict and relaxed, that govern the accuracy or other performance characteristics exhibited by the operations subject to such requirements. The requirements apply only in the strict mode. The idea for the modes originated as a way of giving the user a choice between fast but not fully accurate elementary functions implemented partially in hardware and somewhat slower, but fully accurate, elementary functions implemented in software; it was subsequently extended to cover the predefined arithmetic operations as well, so that when conformance to the arithmetic model carries a price, the user can choose between speed and portable accuracy.

The language does not say how the user shall select a mode, or which mode shall be the default, but it is clear that the mode must govern both code generation and library searches. The modes need not actually be distinct, however; an implementation that meets the requirements of the strict mode at little or no cost to the user has no real incentive to offer the user another choice, and the two modes might be identical.

2.2 Accuracy Requirements for Predefined Floating-Point Operations

Ada 83 had a model of floating-point arithmetic based on the Brown model [1], which served as the basis for the accuracy requirements for the predefined floating-point arithmetic operations. Experience has shown that only certain aspects of the model have been used to any great extent, other than by implementors; that the model fostered confusion as often as insight; and that compromises built into the model weakened its role as an analytical tool. The same model has been retained in Ada 9X as the basis for the strict-mode accuracy requirements for the predefined floating-point arithmetic operations, but it has been somewhat simplified by the omission of seldom used features and at the same time strengthened by undoing the compromises. We expect the revised model to be more useful and more understandable.

The most significant changes in the model are as follows:

- Ada 83 had both model numbers and safe numbers, whereas Ada 9X has only model numbers. For a user-declared floating-point type, the Ada 83 model numbers were completely determined by the floating accuracy specification (the declared decimal precision), while the safe numbers reflected, with some compromises, the accuracy of the underlying representation (and were therefore partially implementation defined). In essence, the model numbers of a type represented the worst case of the safe numbers over all conceivable conforming implementations of the type. The Ada 9X model numbers play the same role as the Ada 83 safe numbers in reflecting the actual performance of the arithmetic, but without the compromises.

- The Ada 83 model and safe numbers had a binary radix, independent of the hardware, whereas the Ada 9X model numbers have the hardware radix.

- The Ada 83 model and safe numbers had quantized mantissa lengths, whereas the Ada 9X model numbers have the maximum mantissa length for which the accuracy requirements (expressed in terms of those model numbers) are satisfied. This change, together with the preceding one, goes a long way toward eliminating certain compromises and toward allowing the model numbers, and the attributes related to them, to exhibit a kind of "truth in labeling" with respect to the implementation's actual arithmetic performance.

- The Ada 83 model and safe numbers formed a finite set, whereas the Ada 9X model numbers form an infinite set. A finite subset, the model numbers belonging to the *safe range* of the type, plays a role in the rules governing the signaling of overflow by the raising of the `Constraint_Error` exception. This change fills a gap in the Ada 83 rules by specifying the accuracy of operations that are allowed to overflow but do not, or that successfully use as an operand a value beyond the overflow threshold.

- The Ada 83 model and safe numbers tied range and precision considerations together intimately by the "4*B Rule," whereas Ada 9X separates them better. As a consequence, all hardware floating-point types can be supported without penalizing their purported properties. Potential compatibility problems resulting from this change are largely avoided by the institution of a much weaker "4*D Rule," which provides a minimum range related to the declared decimal precision when a user-declared floating-point type lacks a range specification; when a range is explicitly specified, and when the specified range is sufficiently narrow in relation to the declared precision, a hardware type not eligible for selection in Ada 83 can be selected in Ada 9X, but it will satisfy both the requested precision and the requested range.

- The Ada 83 model and safe numbers had symmetric exponent ranges, whereas the minimum exponent of the Ada 9X model numbers is freed from a connection to the overflow threshold, allowing it to reflect underflow considerations alone.

These changes are expected to benefit mostly those applications that are designed to exploit the model explicitly. The vast majority of applications will be unaffected, except that a few rare anomalies will vanish.

2.3 Accuracy Requirements for Predefined Fixed-Point Operations

In Ada 83, the model we have been discussing above was actually a model of real arithmetic, not just floating-point arithmetic; hence it was used as the semantic basis for the accuracy of fixed-point arithmetic operations as well.

This proved to be a case of wielding much more machinery than the language features warranted. Many of the freedoms permitted by the model were never exploited, because there was no real need to do so. For something that could be viewed intuitively as scaled integer arithmetic, fixed-point arithmetic earned a reputation for being far more complicated than was deserved.

At the same time, the accuracy requirements were difficult enough to satisfy, for some combinations of source and target *smalls*, that many vendors prohibited representation clauses for nonbinary *smalls*, or perhaps for *smalls* not decomposable as products of powers of two and five. Algorithms for arbitrary *smalls* that made only modest assumptions about the hardware were published [5] but were never widely implemented.

In Ada 9X, we have abandoned the application of the model of real arithmetic to fixed-point types. There no longer are model numbers, safe numbers, and machine numbers of fixed-point types. The values of a fixed-point type are, simply, integer multiples of the type's *small* that lie within a certain range. An arithmetic operation that yields a result of a fixed-point type delivers one of these multiples. The accuracy requirements are expressed in terms of sets of these multiples called "perfect result sets" and "close result sets." The former case applies when the source and target *smalls* are "compatible" (related in a commensurate way typical of many carefully constructed applications); the perfect result set in many instances contains a single member, and at worst contains a pair of adjacent values, implying at most one rounding error. Close result sets are used when the *smalls* are incompatible; in this case, the result sets are small but implementation-defined sets of adjacent integer multiples of the type's *small* containing the perfect result set as a subset. It is hoped that the broader implementation freedoms permitted in the latter case will foster wider support for fixed-point types with arbitrary *smalls*.

Other changes involving fixed-point types are as follows:

- The default *small* in Ada 83 is the largest power of two less than or equal to the type's *delta*, whereas in Ada 9X it is an implementation-defined power of two less than or equal to the type's *delta*. This change allows an implementation that uses extra bits for extra precision rather than for extra range to continue to do so.

- Explicit conversion of the result of a fixed-point multiplication or division is no longer required when context determines a unique target type.

- An operand of fixed-point multiplication or division may be a real literal.

2.4 Accuracy Requirements for Functions in Predefined Packages

Accuracy requirements for the elementary functions, complex arithmetic operations, and complex elementary functions are defined in terms of the smallest model interval that contains all values of the form $f \cdot (1 + d)$, where f is the exact value of the corresponding mathematical function for the given arguments and $|d|$ is less than or equal to the maximum relative error or, in some cases, the maximum "box" error, quoted for the function and expressed usually as a constant coefficient times the result type's *epsilon*. For complex-valued functions for which the error bound applies separately to the real and imaginary parts, f is the value of one of these parts. For those in which cancellation is a distinct possibility, box error, which is derived from vector error, is used; in these cases, f and d are complex, and the bounds on the real and imaginary parts of the result are given by the smallest model interval that contains all values in the corresponding part of $f \cdot (1 + d)$.

These error bounds are superseded by requirements that certain functions must produce exact results for particular arguments, or results within the smallest model interval containing the exact result (when the exact result is not a model number). Unlike the error bounds, these *prescribed results* apply even in the relaxed mode.

2.5 Performance Requirements for the Random Number Generator

The analog of required accuracy for the random number generator, in the strict mode, is the passing of prescribed statistical tests. In addition, the minimum period of the random number generator and the granularity of the time-dependent reset function are specified in the strict mode.

2.6 Attributes Related to the Accuracy Requirements

Some of the attributes of the real types relate specifically to the accuracy model. They serve as "environmental enquiries" that can be used in various ways either to report the numerical performance achieved by a program in a given environment or, with careful planning and design, to allow the program to adapt to and exploit the particular properties of the host environment in a portable way.

Numerous changes have taken place in these attributes in Ada 9X. As an example, note that `T'Mantissa` gives the number of binary digits in the mantissa of the model numbers of the type `T` in Ada 83. Since the model numbers of Ada 9X have the hardware radix, the analogous attribute there no longer necessarily refers to binary digits. On hexadecimal hardware, for instance, the corresponding attribute in Ada 9X would be expected to yield only one-fourth the value that `T'Mantissa` gives. (In reality, the ratio is not exactly one-fourth, since the value of the Ada 83 attribute is necessarily selected from a quantized set and is not generally a multiple of four.) To avoid problems of upwards inconsistency, in which Ada 83 programs give different results in Ada 9X, we have given this attribute a new name, `Model_Mantissa`. The old attribute, `Mantissa`, is no longer defined by the language, which of course poses a problem of a different kind, classed as an upwards incompatibility (in which valid Ada 83 programs no longer compile successfully). To ameliorate this problem, we have recommended that implementations continue to provide the obsolescent attributes as implementation-defined attributes for an appropriate transition period; they should recognize the old attributes and allow them, but produce a warning message. Uses of the old attributes can be adjusted on a case-by-case basis to use the new attributes instead.

Like `Mantissa`, the `Emin` attribute also changes its value by a factor of about four in Ada 9X, and it has consequently been renamed `Model_Emin`. On the other hand, an attribute like `Epsilon` changes its value little, if at all. It is defined in terms of the radix and mantissa length of model numbers, and although these are both changing radically, their changes complement each other. If `Epsilon` changes at all, it is due to the fact that mantissa lengths are no longer quantized. Nevertheless, because its value might change, this attribute has also been renamed (as `Model_Epsilon`).

Several of the model-related attributes of floating-point types have been omitted without replacement, and there are a small number of new attributes. Many of the model-related attributes of fixed-point types have been omitted as well, since fixed-point arithmetic no longer uses the concept of model numbers for the basis of its accuracy requirements.

These attributes are all presented in Annex A, along with other attributes of real types that do not relate to the model. The model-related attributes have implementation-defined meanings and values if the Numerics Annex is not implemented, or if the relaxed mode is in effect. Their definition in terms of the model is given in the Numerics Annex.

3 "Primitive Function" Attributes

Several new attributes of floating-point types, called the "primitive function" attributes, are defined in Annex A. Based on ISO/IEC DIS 11729 (for Ada 83), these environmental enquiry and manipulation functions give low-level access to the parts (mantissa and exponent) of floating-point values, cover exact scaling by a power of the hardware radix, include various directed roundings, and handle exact remainder computations and other similar computations whose results must be accurate to the level of machine numbers. It is not possible to achieve otherwise in Ada results that are simultaneously as accurate, efficient, and portable.

In the aforementioned draft international standard, these attributes are represented as subprograms in a generic package. In order to avoid conflicts with other attributes of Ada 9X, we have changed some of their names, and there are slight differences in the complement of functions provided. With allowance for these changes, the account [3] of the proposed secondary standard in the

Ada Yearbook 1992 is also relevant to the version of the primitive functions in Ada 9X.[3]

References

[1] W. S. Brown. A Simple but Realistic Model of Floating-Point Computation. *TOMS* 7(4):445–480, December 1981.

[2] K. W. Dritz. Towards an Ada Standard for the Elementary Functions. F. Long (ed.), *Ada Yearbook 1991*, Chapman & Hall, London, 1991, pp. 283–290.

[3] K. W. Dritz. Development of an Ada Standard for Primitive Floating-Point Functions. F. Long (ed.), *Ada Yearbook 1992*, Chapman & Hall, London, 1992, pp. 295–302.

[4] P. L'Ecuyer. Efficient and Portable Combined Random Number Generators. *CACM* 31(6):742–749,774, June 1988.

[5] P. N. Hilfinger. Implementing Ada Fixed-Point Types Having Arbitrary Scales. Computer Science Report No. 582, Univ. of California at Berkeley, June 1990.

[6] G. S. Hodgson. The Developing Standards for Complex Scalar and Array Types, Basic Operations, and Elementary Functions for Ada. C. Loftus (ed.), *Ada Yearbook 1993*, IOS Press, Amsterdam, 1993, pp. 369–382.

[7] ISO/IEC CD 8652. Programming Languages—Ada. Committee Draft, September 1993.

[8] ISO/IEC JTC1/SC22 N 1455. Programming Languages—Rationale for the Programming Language Ada. Draft version 4.0, September 1993.

[9] G. Marsaglia and A. Zaman. A New Class of Random Number Generators. *Annals of Applied Probability* 1(3):462–480, August 1991.

[10] S. K. Park and K. W. Miller. Random Number Generators: Good Ones are Hard to Find. *CACM* 13(10):1192–1201, October 1988.

[11] B. A. Wichmann and I. D. Hill. A Pseudo-Random Number Generator. Report DITC 6/82, National Physical Laboratory, Teddington, England, June 1982.

[3] Subsequent to the publication of the 1992 yearbook, the proposed secondary standard for the primitive functions was also changed slightly in response to public comment.