66
4/24/89 JSH

ANL-89/2

Rationale for the Proposed Standard for a Generic Package of Elementary Functions for Ada

by Kenneth W. Dritz

Argonne National Laboratory, Argonne, Illinois 60439
operated by The University of Chicago
for the United States Department of Energy under Contract W-31-109-Eng-38

MASTER

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

Argonne National Laboratory, with facilities in the states of Illinois and Idaho, is owned by the United States government, and operated by The University of Chicago under the provisions of a contract with the Department of Energy.

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

This report has been reproduced from the best available copy.

Available from the
National Technical Information Service
NTIS Energy Distribution Center
P.O. Box 1300
Oak Ridge, TN 37831

Price: Printed Copy A02
Microfiche A01

Distribution Category:
Mathematics and Computer
Science (UC-405)

ANL--89/2

DE89 010282

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, Illinois 60439-4801

**Rationale for the
Proposed Standard for a Generic Package of
Elementary Functions for Ada**

by

Kenneth W. Dritz

Mathematics and Computer Science Division

March 1989

MASTER

This work was supported by the Strategic Defense Initiative Organization, Office of the Secretary of Defense, under WPD B423.

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

LEGIBILITY NOTICE

A major purpose of the Technical Information Center is to provide the broadest dissemination possible of information contained in DOE's Research and Development Reports to business, industry, the academic community, and federal, state and local governments.

Although a small portion of this report is not reproducible, it is being made available to expedite the availability of information on the research discussed herein.

Contents

Abstract	1
Introduction	1
Why is a standard needed, and how did this proposed standard come to be?	1
Why does the proposed standard define a generic package?	2
Can the generic actual type contain a range constraint?	2
What functions are included?	2
Are angles measured in radians, degrees, or what?	3
Why is the optionality of the CYCLE and BASE parameters handled by subprogram overloading instead of simply using default values for them?	3
What purposes do the accuracy requirements serve, and how were they determined?	4
What is the role of the range definitions?	6
How are exceptional conditions treated?	6
If overflow is signaled by an exception, why isn't underflow so signaled?	8
Why does $0.0^{**}0.0$ raise ARGUMENT_ERROR?	9
How are portable implementations of GENERIC_ELEMENTARY_FUNCTIONS accommodated?	9
Conclusions; a look at the future	10
References	11

Rationale for the Proposed Standard for a Generic Package of Elementary Functions for Ada

by

Kenneth W. Dritz

Abstract

This paper supplements the "Proposed Standard for a Generic Package of Elementary Functions for Ada," written by the ISO-IEC/JTC1/SC22/WG9 (Ada) Numerics Rapporteur Group for presentation to WG9. Based on recommendations made jointly by the ACM SIGAda Numerics Working Group and the Ada-Europe Numerics Working Group, the proposed elementary functions standard is the first of several anticipated collateral standards to address the interrelated issues of portability, efficiency, and robustness of numerical software written in Ada. Organized as a series of questions and answers, this supplement outlines the reasoning by which the proposed standard came to acquire certain features and exclude others.

Introduction

In the three years since the ACM SIGAda Numerics Working Group first began to work with the Ada-Europe Numerics Working Group on a standard specification for a generic package of elementary functions for Ada, it has received numerous inquiries about details of that specification from observers of the effort and from potential future implementors. Particular questions—especially one about the handling of certain optional parameters—have been answered more than once. This paper has been written not just to provide ready answers to those questions which, by their demonstrated popularity, we can expect to be asked again; it tries to anticipate others, and it collects in one place—as a kind of separate appendix to the proposed standard—the sometimes subtle reasons why certain decisions were reached during the development of the standard. With the latter, we include the reasons for deciding *not* to do things in certain obvious ways. This paper also discusses the less readily apparent implications of some of those decisions for implementors. Lastly, we hope that it will enhance the understanding of the proposed standard and facilitate its review.

Why is a standard needed, and how did this proposed standard come to be?

The absence of predefined elementary functions from Ada has been one of the deterrents to the portability of scientific and engineering applications software written in that language. The need for such functions has been widely recognized, as evidenced by the support given to them by compiler vendors in the form of proprietary packages, as well as by several purveyors of libraries of mathematical software. While this has served the immediate needs of programmers within their own environments, it has done little to solve the broader problem of portability of applications software using the elementary functions. The reason, of course, is the lack of commonality among the different packages: they differ in the number of functions implemented, their names, their parameter profiles, the handling of exceptional conditions, and even the use (or avoidance) of genericity.

Instead of including predefined elementary functions in Ada, its authors gave the language the necessary general features for defining and collecting subprograms together into libraries (e.g., packages, generics, and subprogram overloading), and for creating portable and efficient numerical software in particular (e.g., a model of floating-point arithmetic, and environmental enquiries in the form of attributes), and then left it to experienced numerical analysts to do what they are uniquely qualified to do: apply those features to the task of specifying and implementing high-quality libraries of mathematical software. Numerical analysts were already engaged in this work before Ada itself was standardized. The need for standards was recognized early, with several preliminary proposals [6, 14,

13] published between 1982 and 1987. Other seminal papers on the content, philosophy, and implementation of scientific libraries in Ada were collected together in [7] in 1986. People and papers came together beginning in about that same year to form committees with working documents. The standardization effort has been supported and encouraged in the United States by the Ada Joint Program Office of the U. S. Department of Defense, and in Europe by the Commission of the European Communities. The ACM SIGAda Numerics Working Group and the Ada-Europe Numerics Working Group have worked together closely since then. Interim reports on the work of the former committee were presented in 1987 at the International Conference on the Ada Programming Language [15] and in 1988 at the Sandia Workshop on Ada in Real-Time and Scientific Environments [4, 19]. Various drafts of the working papers that eventually resulted in the present proposal were circulated, and the response to them has been enthusiastic. The final product of this joint effort is the proposal [11], adopted by the WG9 Numerics Rapporteur Group in March of 1989 for submission to WG9; that is the document which this Rationale describes.

The rest of this paper is intended to be read in conjunction with the proposal.

Why does the proposed standard define a generic package?

The package construct is the obvious mechanism for encapsulating a functionally cohesive set of subprograms and their related exceptions, global data, etc. The facilities of TEXT_IO are made available through that mechanism, for example. Using a generic package instead of an ordinary package is appropriate, furthermore, when the facilities to be encapsulated need to be parameterized by some property of the application in which they are to be used. In view of the rules for parameter associations, the inability to anticipate which floating-point type (or types) the programmer will choose for the application dictates that the package containing the elementary functions be made generic on the type of their formal parameters and returned value.

For that reason the elementary functions package is indeed generic. It is named `GENERIC_ELEMENTARY_FUNCTIONS`, and it has one generic formal parameter, which is a generic formal type named `FLOAT_TYPE`. An instantiation of the generic package with a generic actual type (which can be any floating-point subtype) produces a package containing elementary functions that can be invoked with an argument or arguments of that subtype.

Can the generic actual type contain a range constraint?

The possibility of a range constraint in the floating-point subtype associated with `FLOAT_TYPE` poses certain potential problems for implementations. The proposal says that such a range constraint is allowed and implies that the consequences of such a range constraint shall be limited to those over which the implementation has no control, namely, the raising of `CONSTRAINT_ERROR` if an actual argument or the returned value of an elementary function violates the range constraint. In particular, such a range constraint cannot lead to the raising of `CONSTRAINT_ERROR` by an operation internal to the body of an elementary function, or during elaboration. To produce that behavior, an implementation must in general avoid using the type `FLOAT_TYPE` for variables (other than formal parameters) and constants local to the body of an elementary function. Several portable techniques are available for declaring local variables with a precision at least as great as that of `FLOAT_TYPE`, but without its range constraint (if it has one), and without simply using (always) the maximum precision available; one such technique is illustrated in [18].

This complication for implementations could have been avoided only by decreeing that the generic actual subtype not have a range constraint. Such a condition was viewed as inimical to the normal freedom programmers expect in choosing their types and subtypes, and its enforcement would have been difficult and awkward. In this tradeoff, the interests of users edged out those of implementors.

What functions are included?

Nineteen functions and one operator are defined in the elementary functions package. These are `SQRT`; the two functions (`EXP` and `LOG`) and one operator (`**`) of the exponential family; the four commonly encountered functions (`SIN`, `COS`, `TAN`, and `COT`) of the circular trigonometric family; their inverses (`ARCSIN`, `ARCCOS`, `ARCTAN`, and `ARCCOT`); the four commonly encountered functions (`SINH`, `COSH`, `TANH`, and `COTH`) of the hyperbolic trigonometric family; and their inverses (`ARCSINH`, `ARCCOSH`, `ARCTANH`, and `ARCCOTH`). They were chosen because of their widespread utility in scientific and engineering applications. Actually, the circular trigonometric functions and

their inverses are each represented by a pair of overloaded functions, with different numbers of parameters; the same is true of the LOG function. With overloadings included, a total of twenty-eight functions and one operator are defined in the elementary functions package.

Are angles measured in radians, degrees, or what?

Users have a choice of units in which angles are measured, and the overloadings in the circular trigonometric family and their inverses play a role in the exercise of that choice.

Most often, the desired angular measure is radians; consequently, it is the easiest to specify. Thus, `SIN(X)`, for example, yields the sine of `X`, where `X` is understood to be measured in radians, and similarly `ARCSIN(X)` yields the angle (in radians) whose sine is `X`. To specify some other angular measure, one would supply a value for `CYCLE` (which is the second parameter, except in the case of `ARCTAN` or `ARCCOT`, where it is the third). For example, the sine of `X`, where now `X` is understood to be measured in degrees, would be written as `SIN(X,360.0)`; by the same token, the angle (in degrees) whose sine is `X` is `ARCSIN(X,360.0)`. Other angular measures can be accommodated by using the appropriate value for `CYCLE`—e.g., 1.0 for bams, 6400.0 for mils, 400.0 for grads. From these examples it should be clear that the numerical value of `CYCLE` has the following interpretation: an angle `X` numerically equal to the value of `CYCLE` represents one complete cycle of revolution (i.e., one period of the function). It should also be clear that when the `CYCLE` parameter is omitted, as in `SIN(X)`, the effect is as if a `CYCLE` of 2π had been specified.

A similar choice exists with respect to the base of the LOG function. `LOG(X)` means the natural or Napierian logarithm (i.e., base e); for other bases, such as 2.0 or 10.0, which are perhaps the most common after e , one writes `LOG(X,2.0)`, `LOG(X,10.0)`, etc. The optional parameter is called `BASE`, and when it is omitted the effect is as if a `BASE` of e had been specified. There is no `BASE` parameter for the `EXP` function, which is the inverse of `LOG`, since that functionality can be obtained with the exponentiation operator: the number whose logarithm to the base `B` is `X` can be computed as `B**X`.

Why is the optionality of the CYCLE and BASE parameters handled by subprogram overloading instead of simply using default values for them?

The preceding discussion suggests that the optionality of the `CYCLE` and `BASE` parameters could have been handled very simply by defining appropriate default values for these parameters, and yet subprogram overloading was used instead, which unfortunately adds nine subprogram declarations to the specification of `GENERIC_ELEMENTARY_FUNCTIONS`. What is so disadvantageous about the default-value method to warrant this increase in the size of the specification of `GENERIC_ELEMENTARY_FUNCTIONS`?

The answer is rather subtle, and it required much debate during the development of this proposed standard. The problem is essentially that 2π and e , being irrational, are not representable exactly in any implementation, so the best that could possibly be done would be to use a default value that is a close approximation to 2π or e (implicit conversion of 2π or e , expressed as a numeric literal with an arbitrarily large number of trailing digits, to the type `FLOAT_TYPE` is only required to yield a value in the same safe interval as the literal). The use of such an approximation to 2π in the computation of the circular trigonometric functions, as if it were the true period, will produce results with unacceptable accuracy. The error is not like a simple roundoff error but has the nature of a cumulative phase shift that increases as the number of periods, or cycles, represented by `X` increases. For sufficiently large `X`, there will be no correct digits whatsoever in the result.

Better results can be obtained, and this problem ameliorated somewhat, if the implementation uses an internal representation of 2π that has more precision than `FLOAT_TYPE` has. The necessary additional precision may be obtained from hardware, if it is available; if not, there are techniques for “simulating” extra precision (see, e.g., [18]). But using extra precision only pushes the “phase shift” problem, wherein all accuracy is lost, farther away, i.e., to larger values of `X`; it does not eliminate it entirely. Therefore, the amount of extra precision required is related to the range of values of `X` for which some stated accuracy is to be achieved. Since the required accuracy and the domain over which it must be achieved are both spelled out in the proposed standard, implementors have the information they need to satisfy the standard’s requirements.

It turns out, in fact, that if the default-value method had been employed for CYCLE in this standard, and an implementation used that value as if it were the true period, then the range of values of X for which the stated accuracy could be achieved would not even extend as far as one complete period away from the origin. This also means that an implementation that calculates $\text{SIN}(X)$ by calling $\text{SIN}(X,P)$ for some value of P meant to approximate 2π , including a literal with an arbitrarily large number of digits, will fail to meet the specifications.

All of the other standard periods are given by CYCLE values that are representable on all machines. With the aid of an appropriate "exact-remainder" algorithm, implementations of the explicit-CYCLE forms of the functions will have no difficulty reducing arbitrarily large values of X to the primary interval near the origin without error. (In fact, the exact-remainder function is included in a generic package of floating-point manipulation functions also being proposed for standardization; see the latest draft of [1].) That is why the explicit-CYCLE forms of the circular trigonometric functions have no restrictions on the range of values of X for which the stated accuracy requirements must hold.

It might have been reasonable to employ the default-value method for CYCLE and expect implementations to meet the accuracy requirements over the given range of values for X by recognizing when CYCLE has the default value but not using it (in that case) as if it were the true period—i.e., by using the default value merely as a *signal* that a different argument-reduction method should be used. This was actually considered but ultimately abandoned because of the possibility of non-monotonic behavior as CYCLE sweeps through the default value, and because of potential surprises that could occur should users supply an explicit CYCLE that they believe to be a close approximation to 2π but that is not identical to the default value. Making CYCLE an enumeration type was also considered, but that, too, was abandoned, largely because it would have limited the available periods to a fixed set; furthermore, obtaining a numerical value of the period denoted by the enumeration value represents an unnecessary overhead.

Various other solutions of "the CYCLE problem" were also investigated, such as packaging the circular trigonometric functions and their inverses inside an inner generic package having CYCLE as a generic formal parameter (e.g., a generic formal object of mode "in"). While these offered some elegant advantages, they also had unacceptable disadvantages.

The inverses of the circular trigonometric functions do not exhibit the "phase shift" phenomenon with respect to errors in their periods. For them, it is sufficient to compute the result as (for example) a fraction of a period and then scale that by multiplying by CYCLE. The default-value method for handling the optionality of CYCLE would have posed no problems, but subprogram overloading was preferred simply for reasons of uniformity. The same reasoning applies to BASE, in the case of LOG.

What purposes do the accuracy requirements serve, and how were they determined?

One of the significant advances represented by this proposal is its inclusion of accuracy requirements for implementations of the elementary functions. Not usually considered in formal specifications of mathematical software, accuracy requirements are made possible largely by the model (adapted from Brown [3]) of real arithmetic incorporated into Ada, and the form they take is influenced by the availability of attributes that characterize properties of Ada implementations relative to that model. Because the accuracy requirements will have an effect on what implementors can do, they will translate into some assurance of quality for the user; in addition, they will permit users to carry out error analyses of programs containing references to the elementary functions.

Two kinds of accuracy requirements—maximum-relative-error bounds and "prescribed results"—are included. All of the functions have maximum-relative-error bounds that limit the relative error in the computed result, over the whole range of valid arguments (or, in some cases, over a stated portion of the range). In addition, the required results at certain key argument values are prescribed more precisely for some of the functions.

The maximum-relative-error bounds were determined by numerical analysts having broad knowledge of algorithms and implementation techniques for the elementary functions. They are, of course, tailored to the specific properties of each function. They are considered to be realistic and to give implementors some leeway for creativity and individualism in regard to the tradeoff between accuracy and efficiency. While they do rule out naive implementations, they have proven to be conservative in the sense that it is not especially difficult for a knowledgeable implementor to produce implementations exceeding the accuracy requirements.

The maximum-relative-error bounds are based on the implemented precision of the generic actual parameter associated with `FLOAT_TYPE` rather than on its declared precision. This is reflected in the use of `FLOAT_TYPE'BASE'EPSILON`, rather than `FLOAT_TYPE'EPSILON`, in the formulas for maximum relative error. Effectively, those formulas constrain the computed result to lie within an appropriate number of safe intervals of the mathematical result, just as Ada does for the predefined arithmetic operators, which is what motivated the use of the `BASE` attribute in these formulas. In practice, it means that the approximation technique employed by the implementation must be appropriate for the precision of the base type of `FLOAT_TYPE` rather than just that of `FLOAT_TYPE` itself, i.e., it must be capable of exploiting all the precision inherent in the underlying base type.

Error analyses of programs containing the elementary functions, using the maximum-relative-error bounds as given in the specifications of `GENERIC_ELEMENTARY_FUNCTIONS`, are qualitatively portable in the sense that their form does not change from one Ada implementation to another. They are not quantitatively portable, however, since the numerical size of the maximum-relative-error bounds depends on the Ada implementation's mapping between user-declared floating-point subtypes and the predefined floating-point types. Nevertheless, a quantitatively portable error analysis may also be carried out merely by substituting `FLOAT_TYPE'EPSILON` for `FLOAT_TYPE'BASE'EPSILON` wherever it arises in the analysis. The choice is equivalent to carrying out the error analysis either at the level of safe numbers or at the level of model numbers; both are qualitatively portable, but only the latter is quantitatively portable. Ada gives analysts that choice when they can "see" all the way down to the level of the basic operations and predefined operators in their Ada programs; the proposed standard for the elementary functions preserves that choice—without requiring one to look inside implementations of the elementary functions—by constraining and describing their behavior at the level of safe numbers (from which their behavior at the level of model numbers can be trivially inferred).

A considerable amount of debate was necessary to reach consensus on this form of the maximum-relative-error bounds. Some of the contributors to this proposal felt, and still feel, that an implementation should be permitted to use coarser and coarser approximation methods as the precision of `FLOAT_TYPE` decreases (e.g., in different instantiations), even when the precision of its base type remains the same. For example, a graphics application might well not need 6-digit accuracy in the elementary functions when the user's actual subtype is declared as "digits 2", and the user might not be willing to pay for the unneeded accuracy in the form of additional iterations through some loop, or additional terms in an approximating polynomial, inside the body of an elementary function. A majority of the contributors felt that it was better to require software to get the most out of the hardware it is given to work with, at least for standard-conforming implementations, reasoning that special requirements can always be met by additional implementations not conforming to the standard.

There was also a question as to whether the use of the `BASE` attribute in the accuracy requirements adequately reflects the *implemented* precision of the user's actual subtype (for example, in the case of a reduced-accuracy subtype, perhaps combined with the influence of representation clauses). "Ada Issue" AI-00407 [10] implies that the implemented precision of a reduced-accuracy subtype, as it affects the storage of variables of the subtype as well as parameter associations and function returns involving the subtype, may be less than the precision of the subtype's base type. Because that decision has profoundly undesirable consequences (including the obfuscation of the concept of "representation of a type"; the loss of the ability to specify and analyze the behavior of composite operations, represented by functions, using the same abstractions—including safe intervals—as are applicable to the basic and predefined operations; and the rendering of certain classes of attributes nearly useless), and because it appears to conflict with other requirements or implications of the language [20], it is felt that it cannot survive, and Ada Issue AI-00571, which calls for its reevaluation, has been submitted.

"Prescribed results" are used in some cases to constrain the computed result even more than the maximum-relative-error bounds constrain it. For example, `EXP(0.0)` is prescribed to yield exactly 1.0. The prescribed results reflect behavior that is both highly desirable from a numerical point of view and easy to achieve. In most cases, sensible algorithms will achieve the required behavior without extra effort; when necessary, it can always be achieved with a test for the special argument values.

Some of the prescribed results appear to require the function to deliver a value that cannot be computed exactly; for example, one of the prescribed results reads "`ARCSIN(1.0) =  $\pi/2$` ". What does this mean? The proposed standard says that a prescribed result that is a safe number must be delivered exactly; in the case of one that is not, such as this one, the implementation may deliver any value in the surrounding safe interval. The required behavior can be achieved without difficulty, even in portable implementations.

What is the role of the range definitions?

Range definitions (or restrictions) are included with some of the functions for several reasons. In the case of functions that are mathematically multivalued, they serve to define the principal range for the implementation, enabling it to be single-valued without ambiguity. In other cases, they impose highly desirable and easily achieved numerical constraints on the results—constraints that do not automatically follow from the maximum-relative-error requirements. In this latter context, they behave like additional prescribed results (in the form of an inequality, rather than an equality). And like prescribed results, range limits are sometimes given by values that cannot be computed exactly. In analogy to the meaning of prescribed results, the proposed standard defines the meaning of range limits like this: When a range limit is a safe number, the implementation must not exceed it; when it is not, the implementation may exceed it, but it may not exceed the next safe number beyond the range limit in the direction away from the interior of the range. The required behavior can be achieved without difficulty, even in portable implementations. (For more on range definitions, see the question on underflow, below.)

How are exceptional conditions treated?

Two types of exceptional condition are explicitly recognized by the proposed standard; in each case, the defined action is to raise an exception. Equally important, an implementation is prohibited from raising spurious exceptions if it is to conform to the standard.

The first type of exceptional condition under which an implementation is allowed to raise an exception instead of delivering a result occurs when the arguments of one of the elementary functions are such that its mathematical result is not defined—in other words, when its arguments are invalid. A familiar example, given that arguments and results in this package are restricted to the real domain (as opposed to the complex domain), is an attempt to compute the square root of a negative number. The validity or invalidity of arguments is completely defined by the “domain definitions” included with the description of each function in the proposed standard.

When faced with invalid arguments, an implementation is not merely allowed to raise an exception; it is required to do so. The standard prescribes the raising of the `ARGUMENT_ERROR` exception, which it defines and reserves for this situation.

The validity of given arguments is never influenced by hardware properties; if `ARGUMENT_ERROR` is raised by a function for certain arguments in one implementation, it will be raised for those arguments in any implementation. Argument validity can reliably be established by inspection of the arguments, that is, by subjecting them to appropriate tests. While it will usually be most convenient for an implementation to check for argument validity before attempting to compute a result, other strategies may be possible and appropriate in some cases.

The second type of exceptional condition under which an implementation is allowed to raise an exception instead of delivering a result occurs when the mathematical result is well defined for the given arguments but something else (from among a limited list of things) unavoidably stands in the way of actually delivering that result. There are in fact three misfortunes that can befall an implementation, interfering with its ability to deliver a numerical result that is close enough to the mathematical result to satisfy the accuracy requirements:

- It may happen that the given arguments fail to satisfy range constraints inherent in the user's generic actual subtype, or that the function's computed result fails to satisfy those constraints.
- It may happen that the function that is invoked is unable to obtain the storage it needs to perform the requested computation.
- It may happen that the computed result is so large, in magnitude, that it exceeds the hardware's representational capabilities—i.e., it overflows.

The first of these misfortunes can occur during any parameter association or on any function return; it is not peculiar to the functions in this package. It is a fact of life of Ada, calling for the raising of the `CONSTRAINT_ERROR` exception at the place of the call when it occurs during a parameter association, or at the place of the return statement when it occurs during a function return. In the former case the function is never entered, so clearly a numerical result cannot be delivered. In the latter case the function has computed an appropriate numerical result and has attempted to deliver it but has failed, because of the range constraints that the user has imposed on the arguments and results of all the elementary functions. In marginal cases, other (slightly different) results might have been produced that do satisfy the range constraints while still satisfying the accuracy requirements, but it is not highly likely. So, for

all practical purposes, it may be assumed that it is just not possible to deliver a satisfactory numerical result. The proposed standard for `GENERIC_ELEMENTARY_FUNCTIONS`, by allowing `CONSTRAINT_ERROR` to be raised naturally when this misfortune occurs, does not impose any special design requirements on implementations. Indeed, in the case of parameter associations, there is nothing else that it could do.

The second of the three misfortunes can occur during any subprogram invocation, or during the elaboration of a subprogram's declarative part just after its invocation; it, too, is not peculiar to the functions in this package. Thus it, too, is a fact of life of Ada, calling for the raising of the `STORAGE_ERROR` exception at the place of the call (for all practical purposes) when it occurs for either of these reasons. Clearly, it is just not possible to deliver a numerical result, since the function either has never been entered or has not finished elaborating its declarative part. Since storage requirements for elementary function implementations (of scalar arguments) are modest, it is highly likely that the application was running near the limit of available storage at the point where the elementary function was invoked, and that it is merely an accident that the raising of `STORAGE_ERROR` occurred there instead of somewhere else. The proposed standard for `GENERIC_ELEMENTARY_FUNCTIONS`, by allowing `STORAGE_ERROR` to be raised naturally when this misfortune occurs, does not impose any special design requirements on implementations. Indeed, since a handler for `STORAGE_ERROR` established by the function will not be entered in the cases described above, there is nothing else that it could do. (`STORAGE_ERROR` can also be raised by evaluation of an allocator in the sequence of statements of the function's body. Although in that case the propagation of the exception to the caller is *not* inevitable, it seemed hardly practical or appropriate for the proposed standard to distinguish that case from the earlier ones.)

The last of the three misfortunes, a computed result whose magnitude is so big that it cannot be represented in the hardware, is commonly called overflow. Since the overflow threshold is a hardware property, exactly which results are "too big" cannot be predicted with certainty without reference to the hardware. The model of real arithmetic in Ada allows one to say with certainty, however, that a computed result whose absolute value is less than or equal to `FLOAT_TYPE'SAFE_LARGE` is always capable of being represented. Thus it is reasonable to insist that, whenever all possible results permitted by the accuracy requirements are less than or equal to `FLOAT_TYPE'SAFE_LARGE` in absolute value, the implementation must deliver one of them (if it does not suffer one of the earlier misfortunes). On the other hand, if any result permitted by the accuracy requirements would exceed `FLOAT_TYPE'SAFE_LARGE` in absolute value, then it is possible that that result is the one that the implementation might try to compute, and that it would also exceed the hardware's overflow threshold. Thus, whenever any result permitted by the accuracy requirements exceeds `FLOAT_TYPE'SAFE_LARGE` in absolute value, an implementation is permitted to signal overflow instead of delivering a result. That does not mean that it *must*, of course; the actual result that it computes could be some other permitted result that does not exceed `FLOAT_TYPE'SAFE_LARGE` in absolute value—and therefore does not exceed the hardware's overflow threshold—or it could be one that does exceed `FLOAT_TYPE'SAFE_LARGE` but still does not exceed the hardware's overflow threshold. By the same token, the fact that some of the permitted results do not exceed `FLOAT_TYPE'SAFE_LARGE` in absolute value does not oblige the implementation to deliver a result in the case that others do exceed it. This is the rationale for the variety of behaviors that an implementation is allowed to exhibit in the vicinity of the overflow threshold.

If overflow needs to be signaled, the proposed standard calls for that to be done in the way that Ada mandates for its predefined operators, i.e., by raising `NUMERIC_ERROR` (which is changed to `CONSTRAINT_ERROR` by AI-00387).

Implementors have a choice of ways to deal with possible overflows in the final result. On the one hand, implementors can use a predictive technique—that is, examine the arguments before using them to compute the result and raise the appropriate exception (by a "raise" statement) if one of the permitted results (i.e., any value differing from the mathematical result by no more than the maximum relative error) would exceed `FLOAT_TYPE'SAFE_LARGE` in absolute value. Or, implementors can omit the argument prescreening and just go ahead and compute, relying on the hardware to detect and signal in the natural way an overflow in the final result—or an overflow (or other exceptional condition) that presages overflow in the final result, even though it may occur well before the final step in the process of obtaining that result. The latter technique entails a more sophisticated analysis on the part of the implementor, who must be certain that extreme arguments do not violate some implicit assumption of the algorithm, causing it to misbehave and leading to an unacceptable numerical result rather than to overflow, but that extra effort is usually rewarded by a marginally more efficient and useful product.

Recall that, whenever all possible results permitted by the accuracy requirements are less than or equal to `FLOAT_TYPE'SAFE_LARGE` in absolute value, the implementation must deliver one of them (if it does not suffer one of the earlier misfortunes). Through that provision the proposed standard imposes on implementors the responsibility

of ensuring either that their chosen algorithm does not inadvertently overflow (or suffer some other numerical exception) in the calculation of an intermediate result or that, if it does, it handles the exception locally and goes on to compute and deliver an acceptable result. To conform to this standard, an implementation cannot be so naive that it raises spurious exceptions.

The three “misfortunes” described above share two characteristics: (1) they are not peculiar to the elementary functions and can indeed occur in many places in arbitrary Ada programs, and (2) Ada provides predefined exceptions to be used to signal their occurrence. It seemed inappropriate for this standard to distinguish their occurrence in the context of the elementary functions by prescribing the raising of a new exception (a possibility that was considered); it would not have been possible to do so uniformly and consistently, in any case.

In fact, consideration was even given to carrying the preceding decision to its extreme, i.e., to dispensing with the `ARGUMENT_ERROR` exception and instead prescribing the raising of `CONSTRAINT_ERROR` for invalid arguments, on the grounds that an invalid argument represents the violation of a constraint in the broad sense of the term. While that treatment of invalid arguments would parallel the handling of division by zero, which raises `NUMERIC_ERROR` now but will raise `CONSTRAINT_ERROR` after AI-00387 becomes effective, it was felt that prescribing the use of a new exception to report exceptional conditions *uniquely related to the functionality of this package*, following the precedent set by the predefined I/O packages, was more useful to the application programmer.

The name `ARGUMENT_ERROR` exported by instantiations of `GENERIC_ELEMENTARY_FUNCTIONS` is a renaming of the exception of the same name declared in the package `ELEMENTARY_FUNCTIONS_EXCEPTIONS`, whose specification is also contained in this proposed standard. Therefore, multiple instantiations, should they be needed in the same application, do not give rise to multiple `ARGUMENT_ERROR` exceptions. Thus it is not necessary to establish a handler for multiple instances of this exception; one handler suffices for all the instantiations. Unfortunately, this convenience is partially offset by the fact that the name `ARGUMENT_ERROR` will not be directly visible in the case of multiple instantiations, and an expanded name will have to be used. The alternative of not exporting the name `ARGUMENT_ERROR` from instantiations, and requiring the user to “with” and “use” `ELEMENTARY_FUNCTIONS_EXCEPTIONS` to gain direct visibility of the name `ARGUMENT_ERROR`, was considered, but it was rejected on the grounds that the present proposal is more convenient for the user when there is only one instantiation, as is expected to be the common case. The precedent set by the exporting of renamed exceptions by some of the predefined I/O packages, which unfortunately suffer from the same name-visibility problem, also figured in this decision.

If overflow is signaled by an exception, why isn't underflow so signaled?

The proposed standard does not call for the raising of an exception upon underflow primarily because of the precedent set by Ada, which does not have a predefined exception for underflow. In fact, underflow is not considered an exceptional condition by Ada, by virtue of the fact that the interval between zero and `T'SAFE_SMALL` (for any real subtype `T`) is a normal safe interval. This treatment by Ada is consonant both with the returning of “denormalized numbers” by hardware obeying the IEEE Floating-Point Arithmetic Standard 754 [8] or 854 [9], and with the classical recovery from hardware underflow, namely, flushing to zero.

The possibility of underflow and, in particular, flushing to zero, introduces the chance that the computed result of some of the elementary functions will be 0.0 even though 0.0 is not in their mathematical range for any finite argument. For example, `EXP` of an extremely negative argument may underflow to 0.0, even though e^x cannot mathematically be exactly zero for any finite x . Similarly, `X**Y` may underflow to 0.0 for positive x and extremely negative y , even though mathematically x^y cannot be exactly zero for any finite y (when x is non-zero). Another disadvantage of flushing to zero is that it fails to satisfy small-relative-error requirements. For these reasons, an earlier version of the proposed standard actually called for the raising of an exception, rather than returning zero, in underflow situations—for those functions whose mathematical range does not include zero. That was abandoned, however, because it would have been too radical a departure from common practice, not to mention inconsistent with Ada's handling of the predefined operators.

The proposed standard substitutes a small-absolute-error requirement for the small-relative-error requirement in underflow situations (where the latter is unsatisfiable). Furthermore, the range definitions in the proposed standard include asymptotic limits. Their inclusion not only legalizes the returning of 0.0 in the cases described above, but it legalizes the returning of ± 1.0 in the case of sufficiently extreme arguments to `TANH` or `COTH`; even though there

is no finite X for which the hyperbolic tangent or cotangent of X is exactly $+1.0$ or -1.0 , these values may be closer to the mathematical result than any other representable number.

Why does $0.0^{**}0.0$ raise `ARGUMENT_ERROR`?

The case of $X = 0.0$ and $Y = 0.0$ is excluded from the domain of $X^{**}Y$ because it is not possible to assign a unique value to the result. The limiting value of X^Y as both X and Y approach 0.0 depends on exactly how X and Y approach 0.0 .

The committee considered defining the result of $0.0^{**}0.0$ in this standard to be 1.0 , as advocated by some numerical analysts [12]. This would have the virtue of agreeing with the predefined `**` operator (whose right operand has type `INTEGER`). Use of default values, sometimes backed up by “sticky” flags to detect their occurrence when that is important, is an appropriate way to deal with exceptional conditions in a language lacking exceptions and exception handlers; it is much more in the conservative style and culture of Ada, however, not to impose particular programming disciplines on users and to provide only minimal and *safe* capabilities out of which the user can fashion whatever discipline is appropriate.

How are portable implementations of `GENERIC_ELEMENTARY_FUNCTIONS` accommodated?

Two kinds of implementations of `GENERIC_ELEMENTARY_FUNCTIONS` are envisioned. On the one hand, vendors of Ada compilers for specific hardware may have an interest in producing tailored implementations of `GENERIC_ELEMENTARY_FUNCTIONS` for that hardware; for such implementations, portability is not a concern, and major portions of those implementations may not even be written in Ada. On the other hand, independent software producers may have an interest in developing a single implementation of `GENERIC_ELEMENTARY_FUNCTIONS` that is portable to a wide variety of different machines and Ada systems. While both kinds are allowed by the specifications, portable implementations—understandably the more challenging of the two—are affected by certain special considerations that do not apply to tailored ones (but they also benefit from some mild concessions, like the ones inherent in the treatment of prescribed results or range limits that are not safe numbers).

It is highly unlikely that an implementation of `GENERIC_ELEMENTARY_FUNCTIONS` can be rendered in Ada so as to be portable without qualification to all implementations of Ada. Typically, implementations intended to be portable are not absolutely portable, but are portable only with certain qualifications stemming from assumptions built into the code. Some examples of assumptions that simplify and circumscribe the design of an implementation of the elementary functions are the following:

- It might be assumed (i.e., required) that `FLOAT_TYPE'MACHINE_RADIX` is 2 or 16, and not some other number.
- It might be assumed that multiplication or division by a power of the machine radix is exact.
- It might be assumed that `SYSTEM.MAX_DIGITS` falls within some particular range of values.

Assumptions are always avoidable, or at least capable of being weakened, but the amount of effort required to do so may be more than the implementor can justify.

As long as “portable” implementations embody assumptions, there is a risk that they will fail in unpredictable ways when used in an unintended environment (where the assumptions may not hold). The proposed standard addresses this risk in three ways. First of all, it requires that the assumptions be clearly documented, enabling potential users of an implementation to evaluate its suitability in their environment. Secondly, it does not define the behavior of an implementation whose assumptions are violated, thereby giving implementors total freedom to opt either for speed (by ignoring the possibility of such violations) or for robustness (by detecting them and responding—predictably—in ways of their choice). (For example, an implementation might raise `PROGRAM_ERROR`, or some other exception, upon detecting a violation of its assumptions.) Finally, the proposed standard adopts the convention that implementations whose assumptions are violated in some environment are not in conformance with the standard in that environment. This last point is intended to discourage highly restricted (partial) implementations from absolving themselves in the guise of portability. It is not enough to be portable; an implementation must also be useful, and usefulness is measured by the number of environments in which it is conforming.

It is sufficient for a non-portable implementation (one intended for a single environment) to document its assumptions as "intended for use only in the . . . environment."

Conclusions; a look at the future

The proposed elementary functions standard satisfies one part of Work Item JTC1.22.10.02 ("Standardization of Ada Numeric Packages"), assigned to the WG9 Numerics Rapporteur Group. Though it will significantly improve the prospects of portability for engineering and scientific applications written in Ada, much more is needed; accordingly, other standards are being developed in response to the remaining parts of the work item. Reference has already been made to the need for, and development of, a standardized generic package of floating-point manipulation functions. In addition to that, the people who have worked together on the elementary functions standard are now working on proposals for complex arithmetic and complex elementary functions, for vector and matrix routines, and for random-number generators. These efforts, whose motivation comes partly from the consensus of support already exhibited for the proposed elementary functions standard, can be expected to reach fruition in the next few years.

The focus on Ada numerics associated with these ongoing standardization activities is also fostering several research endeavors. One line of research growing out of this effort involves the development of testing strategies and algorithms that can be used to validate the conformance of implementations of the proposed standard. Such methods are under development at several locations, including ANL [17], NAG, CWI [2], and Westinghouse [16]; the work at NAG [5] also encompasses methods for checking that the underlying floating-point arithmetic of the host system conforms to the Ada standard. Other research in progress is aimed at using automated program verification techniques to prove accuracy claims for Ada floating-point programs. These research activities, taken together with the standardization efforts, will enhance both the theoretical and the practical aspects of Ada numerics.

References

- [1] ACM SIGAda Numerics Working Group. Draft 0.81 of a Proposed Standard for a Generic Package of Primitive Functions for Ada, March 1989. Current draft.
- [2] M. Bergman. Testing of Elementary Functions in Ada. CWI Report, Centrum voor Wiskunde en Informatica, Amsterdam, 1989 (to appear).
- [3] W. S. Brown. A Simple but Realistic Model of Floating-Point Computation. *TOMS*, 7(4):445–480, December 1981.
- [4] K. W. Dritz. Influence of Language Features on the Specification of a Standardized Generic Package of Elementary Functions for Ada. In G. Pollock, editor, *Proc. 1988 Sandia Workshop on Ada in Real-Time and Scientific Environments*. Sandia National Laboratory, to appear. Available from author as ANL Preprint MCS-P16-1188.
- [5] M. J. Erl, P. P. Gardner, G. S. Hodgson, and M. W. Pont. Numerical Verification in Ada. NAG Report, Numerical Algorithms Group, Ltd., Oxford, 1989 (to appear).
- [6] R. Firth. Preliminary Draft Specification of a Basic Mathematical Library for the High Order Programming Language Ada. Royal Military College of Science, Shrivenham, Swindon, Wiltshire, England, March 1982.
- [7] B. Ford, J. Kok, and M. W. Rogers, editors. *Scientific Ada*. Cambridge University Press, 1986.
- [8] IEEE. IEEE Standard for Binary Floating-Point Numbers. ANSI/IEEE Std. 754-1985, IEEE, New York, 1985.
- [9] IEEE. IEEE Standard for Radix-Independent Floating-Point Arithmetic. ANSI/IEEE Std. 854-1987, IEEE, New York, 1987.
- [10] ISO-IEC/JTC1/SC22/WG9 (Ada) Ada Rapporteur Group. Ada-Comment Database. Maintained by the Ada Joint Program Office on AJPO.SEI.CMU.EDU.
- [11] ISO-IEC/JTC1/SC22/WG9 (Ada) Numerics Rapporteur Group. Proposed Standard for a Generic Package of Elementary Functions for Ada, March 1989.
- [12] W. Kahan. Why Must $0^0 = 1$? Preprint, University of California at Berkeley, August 1988.
- [13] J. Kok. Proposal for Standard Mathematical Packages in Ada. CWI Report NM 7718, Centrum voor Wiskunde en Informatica, Amsterdam, November 1987.
- [14] J. Kok and G. T. Symm. A Proposal for Standard Basic Functions in Ada. *Ada Letters*, IV(3):44–52, November/December 1984.
- [15] R. F. Mathis. Elementary Functions Packages for Ada. In *Proc. 1987 ACM SIGAda International Conference on the Ada Programming Language* (special issue of *Ada Letters*), pages 95–100, December 1987.
- [16] J. Squire. Verification of Reusable Numerical Software in Ada. Independent Research and Development Report, Westinghouse Electronic Systems Group, Baltimore, 1989 (to appear).
- [17] P. T. P. Tang. Accurate and Efficient Testing of the Exponential and Logarithm Functions in Ada. ANL Report ANL-88-24, Argonne National Laboratory, November 1988.
- [18] P. T. P. Tang. Portable Implementation of a Generic Exponential Function in Ada. ANL Report ANL-88-3, Argonne National Laboratory, February 1988.
- [19] P. T. P. Tang. Use of Language Features in the Implementation and Validation of a Standardized Generic Package of Elementary Functions in Ada. In G. Pollock, editor, *Proc. 1988 Sandia Workshop on Ada in Real-Time and Scientific Environments*. Sandia National Laboratory, to appear. Available from author as ANL Preprint MCS-P25-1188.
- [20] U. S. Department of Defense, Ada Joint Program Office. *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A. U. S. Government Printing Office, Washington, D. C., 1983. Also adopted by ISO as ISO/8652-1987, Programming Languages—Ada.

Distribution for ANL-89/2

Internal:

J. M. Beumer (3)
K. W. Dritz (128)
F. Y. Fradin
H. G. Kaper
A. B. Krisciunas
G. W. Pieper
D. P. Weber

ANL Patent Department
ANL Contract File
ANL Libraries
TIS Files (3)

External:

DOE-OSTI, for distribution per UC-405 (66)
Manager, Chicago Operations Office, DOE
V. Castor, DOD, The Pentagon, Washington, DC
C. Lillie, SDIO, The Pentagon, Washington, DC
W. Ritchie, AJPO, The Pentagon, Washington, DC
K. De Vriendt, Commission of the European Communities, Brussels

Mathematics and Computer Science Review Committee:

J. L. Bona (chair), Pennsylvania State University
T. L. Brown, University of Illinois, Urbana
P. Concus, Lawrence Berkeley Laboratory
S. Gerhart, Micro Electronics and Computer Technology Corp., Austin, TX
H. B. Keller, California Institute of Technology
J. A. Nohel, University of Wisconsin, Madison
M. J. O'Donnell, University of Chicago

ACM SIGAda Numerics Working Group:

W. J. Cody, ANL
P. M. Cohen, Martin Marietta, Denver
S. G. Cohen, Software Engineering Institute, Carnegie Mellon University, Pittsburgh
K. W. Dritz, ANL
B. Ford, Numerical Algorithms Group, Ltd., Oxford, UK
G. S. Hodgson, Numerical Algorithms Group, Ltd., Oxford, UK
J. Kok, Centrum voor Wiskunde en Informatica, Amsterdam
R. F. Mathis, Contel Technology Center, Fairfax, VA
T. G. Mauson, Quantitative Technology Corp., Beaverton, OR
G. B. Myers (chair), Naval Ocean Systems Center, San Diego
B. T. Smith, University of New Mexico, Albuquerque
J. S. Squire, Westinghouse Electric Corp., Baltimore
P. T. P. Tang, ANL
W. A. Whitaker, McLean, VA

ISO-IEC/JTC1/SC22/WG9 (Ada) Numerics Rapporteur Group:

J. G. P. Barnes, Alsys, Henley-on-Thames, Oxfordshire, UK
P. M. Cohen, Martin Marietta, Denver
L. M. Delves, University of Liverpool
K. W. Dritz, ANL
R. Firth, Software Engineering Institute, Carnegie Mellon University, Pittsburgh
B. Ford, Numerical Algorithms Group, Ltd., Oxford, UK
G. S. Hodgson, Numerical Algorithms Group, Ltd., Oxford, UK
J. Kok, Centrum voor Wiskunde en Informatica, Amsterdam
R. F. Mathis, Contel Technology Center, Fairfax, VA
G. B. Myers (chair), Naval Ocean Systems Center, San Diego
J. S. Squire, Westinghouse Electric Corp., Baltimore
R. P. Wehrum, Siemens AG, Munich
B. A. Wichmann, National Physical Laboratory, Teddington, Middlesex, UK
D. T. Winter, Centrum voor Wiskunde en Informatica, Amsterdam
M. Woodger, Alsys, Ashted, Surrey, UK