ANL--90/41

DE91 007076

ANL-90/41

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

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

by

Kenneth W. Dritz

Mathematics and Computer Science Division

December 1990

This work was supported by the Strategic Defense Initiative Organization, Office of the Secretary of Defense, under PMA 2300.

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

by

Kenneth W. Dritz

Abstract

This paper supplements the "Proposed Standard for a Generic Package of Primitive Functions for Ada," written by the ISO-IEC/JTC1/SC22/WG9 (Ada) Numerics Rapporteur Group. Based on recommendations made jointly by the ACM SIGAda Numerics Working Group and the Ada-Europe Numerics Working Group, the proposed primitive functions standard is the second of several anticipated secondary standards to address the interrelated issues of portability, efficiency, and robustness of numerical software written in Ada. Its purpose, features, and developmental history are outlined in this commentary.

At about the time that work on a proposed Ada standard for the elementary functions began in 1986, early efforts to implement the elementary functions—square root, logarithm, trigonometric functions, and the like—underscored the need to be able to perform certain steps in their computation with extreme accuracy. These functions are typically implemented by transforming the argument so that it lies within a reduced range, computing the desired function on the transformed argument by a polynomial or rational approximation (designed to be sufficiently accurate over the relatively narrow reduced argument range) to obtain an intermediate result, and then constructing the final result by appropriately transforming the intermediate result. Accuracy is controlled in the middle step by the choice of approximation method, which bounds the approximation error. However, the final result can be extremely sensitive to errors (such as roundoff errors) made in the argument reduction step. Unnecessary error can also enter in the final step if the transformation it represents is not carried out carefully.

Details of the transformations needed in the argument reduction and result construction steps depend, of course, on the function being implemented. In the case of the periodic functions, the essential requirement is to compute an accurate remainder when the argument is divided by the period, if specified; when the period is allowed to default to the irrational 2π, a technique other than a simple division is required to obtain a suitably accurate remainder. In other cases, especially `SQRT` and `LOG`, decomposition of the argument into its exponent and fraction parts is the starting point, with the fraction part (or a simple function of it) becoming the transformed argument; the result construction step in these cases usually involves a simple modification—often just a scaling—of the intermediate result by a simple function of the exponent part.

If one is interested in implementing the elementary functions in a portable fashion, how does one go about computing accurate floating-point remainders and decomposing floating-point numbers into their constituent parts portably? Two problems arise if one tries to do these things entirely in portable Ada: the result is inefficient, often involving loops that require many traversals; and it cannot be proven fully accurate with Ada's model of floating-point arithmetic, since the model caters to the weaknesses of the weakest conforming implementation of Ada. (On machines manifesting them, such weaknesses—for example, lack of a guard digit—can introduce errors in the argument reduction step that become amplified as the loops are traversed.) The efficiency and accuracy problems can be solved, of course, by judicious use of representation clauses or interface programming in assembler language or even machine language insertions, given knowledge of the host machine, but that obviously destroys portability.

Exact floating-point remainder and decomposition of a floating-point number into its constituent parts are two examples of *primitive functions*—low-level floating-point functions having the property that they cannot be coded in Ada so as to be simultaneously accurate, efficient, and portable. Since we know how to solve the accuracy and efficiency problems when details of the underlying machine are available (indeed, some of the primitive functions are directly available as hardware operations on specific machines), all that is really lacking is a standardized interface to the functions. That is what the proposed generic primitive functions standard [1] provides.

Portable implementations of the generic elementary functions standard will be the first beneficiary of the generic primitive functions standard; others will follow. However, the generic primitive functions standard will always have a specialized clientele: experts, probably highly trained numerical analysts, concerned with the development of high-quality, portable mathematical software. It is not for the average application programmer.

The proposed standard has been developed by the ACM SIGAda Numerics Working Group in collaboration with the Ada-Europe Numerics Working Group. The proposal has been adopted by the WG9 Numerics Rapporteur Group and is to be submitted to WG9 and its parents, leading ultimately to an ISO standard. 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.

Although work on the primitive and the elementary functions standards began at about the same time, the elementary functions standard was completed, except for some late refinements, about a year and a half earlier [9]. Earliest drafts of the primitive functions standard drew heavily from recommendations made many years earlier in [3]; other works influencing the Ada primitive functions at an early date were [6, 11, 15, 14]. Later versions of the primitive functions were influenced by the IEEE floating-point standards [7, 8] and by the proposed Language Compatible Arithmetic Standard (LCAS) [12, 13]. One reason for the delay in completing the primitive functions, relative to the elementary functions, was a series of late additions to the proposed primitive functions standard as the result of evolving implementation experience with the elementary functions. Another was the recognition that software intending to exploit IEEE arithmetic had to pay particular attention to some of its more subtle features, such as denormalized numbers and signed zeros. It took considerable effort to describe the primitive functions so that they could be implemented in either IEEE or non-IEEE environments. This issue also had ramifications for the elementary functions standard, resulting in a recent revision of it [10] and in the updating of its rationale document [5].

The proposed standard for the primitive functions defines the specification of a generic package called `GENERIC_PRIMITIVE_FUNCTIONS`. It is a package because that is the accepted way to collect together several related subprograms; it is generic, with generic formal parameters for the two types used for the arguments and results of the subprograms, in view of the rules for parameter associations and the inability to anticipate the types used in applications. The generic formal parameter `FLOAT_TYPE` gives the type to be used for the floating-point arguments and results of subprograms in `GENERIC_PRIMITIVE_FUNCTIONS`, while the generic formal parameter `EXPONENT_TYPE` gives the type to be used for the few integer arguments and results that, with one exception,[1] deal with exponents of the canonical machine representation. When an instantiation of `GENERIC_PRIMITIVE_FUNCTIONS` is used in an implementation of the elementary functions (e.g., in the body of `GENERIC_ELEMENTARY_FUNCTIONS`), the `FLOAT_TYPE` of the latter should be passed through to the former, and a sufficiently wide integer type should be associated with `EXPONENT_TYPE`. The predefined type `INTEGER` probably suffices for the latter, but if one is worried about sufficient range, then an integer type whose range covers `SYSTEM.MIN_INT .. SYSTEM.MAX_INT` can be defined and used instead.

Like the elementary functions standard, the primitive functions standard permits implementations to impose a restriction that the generic actual type associated with `FLOAT_TYPE` in an instantiation must not contain a range constraint that reduces the range of allowable values. Implementations choosing not to impose the restriction must be designed to be immune from the avoidable effects of such range constraints; in general, this means that variables of type `FLOAT_TYPE` cannot safely be used for intermediate results within an implementation of `GENERIC_PRIMITIVE_FUNCTIONS`. Those imposing the restriction must document it; they can safely use such variables, but they must behave in one of several stated ways (i.e., predictably) if the restriction is violated. (For a detailed discussion of the genesis of this optional restriction and its implications, see the latest revision of [5]. The freedom of an implementation to impose the restriction will be revoked in the future if and when Ada—for example, as part of the Ada 9X revision process—acquires additional functionality that allows the declaration of variables having the precision, but not the range, of a generic formal floating-point type.) Incidentally, implementations are not allowed to impose a similar restriction on the generic actual types that can be associated with `EXPONENT_TYPE` during instantiation; it is not difficult to implement `GENERIC_PRIMITIVE_FUNCTIONS` to be efficient while limiting the consequences of insufficient range in that generic actual type to the unavoidable raising of `CONSTRAINT_ERROR` during a subprogram call or return.

[1] One of the subprograms takes an argument that is a nonzero count of the number of digits to be retained in a particular computation; the predefined integer subtype `POSITIVE` is used for the corresponding parameter.

Perhaps the most significant difference between the two standards, other than subject area, is their respective handling of accuracy requirements. The elementary functions standard allowed implementations to approximate the exact mathematical result but constrained the approximation error by requiring implementations to satisfy "maximum relative error" bounds. In contrast, the primitive functions standard requires implementations to deliver the exact mathematical result defined for each function, whenever that result is representable; approximations are permitted only when the mathematical result is not representable and is smaller in magnitude than the smallest normalized positive floating-point number; and even then, the result is constrained to be one of the adjacent representable numbers. This level of accuracy is an essential aspect of the definition of the functions as operations on machine numbers yielding related machine numbers, without which their utility in argument reduction, etc., would be compromised. Achieving the required accuracy is *not* something that can be accomplished portably in Ada, at least not without making assumptions about the performance of the hardware that go well beyond the requirements imposed by the Ada model of floating-point arithmetic. On the other hand, the required accuracy can be easily and efficiently achieved by targeting implementations for specific environments and by utilizing knowledge of the machine representations in conjunction with appropriate operations (often integer or bit operations), accessed if necessary through low-level interfaces. A precedent for the accuracy required of the primitive functions can be found in the Ada attribute `T'BASE'LAST` for a floating-point type `T`: by definition, it has full machine-number accuracy, which, in general, exceeds model-number and safe-number accuracy.

Because the primitive functions transform machine numbers into other well-defined machine numbers, the standard includes a discussion of exactly what is meant by "floating-point machine number" within the context of the subprograms' definitions. What numbers are in the set of machine numbers? Does that set include the extra-precise numbers that some Ada implementations generate as a consequence of using extended registers for intermediate results? The answer to the latter question must be no, for otherwise the precise mathematical formulas used to specify the results of some of the functions would imply that the output from a function must be extra-precise if its input is, and yet the programmer has no means to ensure that that will be the case. Thus, it is clearly stated that the "machine numbers" referred to throughout the standard are the *storable* machine numbers—the ones that can be (a) stored; (b) propagated by assignment, parameter association, and function returns; and (c) characterized by the representation attributes `FLOAT_TYPE'MACHINE_MANTISSA`, `FLOAT_TYPE'BASE'FIRST`, and `FLOAT_TYPE'BASE'LAST`. Implementations of the primitive functions are entitled to assume that only storable machine numbers will be seen as arguments, and implementations of Ada must uphold that assumption (by forcing storage, if necessary, before calling a primitive function) in order for implementations of the primitive functions to have any hope of conforming to the standard.

Furthermore, because some hardware (e.g., that implementing IEEE arithmetic) has the capability of representing denormalized numbers—those with the minimum exponent and an unnormalized fraction part—one must also be precise about whether the set of machine numbers includes them. The standard says that it does if the hardware has the capability of representing them and the Ada implementation uses the hardware in such a way that it actually generates them; otherwise, it does not. This is especially significant when talking about "adjacent machine numbers," since the machine number adjacent to the smallest positive normalized number, in the direction toward zero, will be a denormalized number if the hardware and Ada implementation recognize denormalized numbers, and zero otherwise. It is also germane to the approximations that are permitted when a defined result falls in the denormalized range and is not exactly representable.

Some hardware (again, typically hardware conforming to IEEE arithmetic) has the capability of representing both positive and negative zeros (i.e., the sign of zero is relevant in some contexts). Like the elementary functions standard, the primitive functions standard allows signed zeros to be exploited if they are present in the hardware, but does not require them to be exploited. And like the elementary functions standard, the primitive functions standard does not give the required sign of each zero result (when signed zeros are being exploited), but leaves that to other standards or interpretations.[2] The behavior of one of the primitive functions, `COPY_SIGN`, does depend on the sign of a zero argument (when signed zeros are being exploited), as is also true of `ARCTAN` and `ARCCOT` in the elementary functions. The standard also clarifies that plus and minus zero are *not* to be considered "adjacent" (and therefore different) machine numbers, in any context where adjacency is relevant; thus, the "neighbors" of zero do not depend on the sign of zero.

[2]There are four exceptions, however. The required signs of zero results from `ADJACENT`, `SUCCESSOR`, `PREDECESSOR`, and `COPY_SIGN` *are* spelled out in the standard because those functions are intimately concerned with representations.

Early versions of the proposed primitive functions standard did not permit *any* approximations: when the exact mathematical result was not representable, they called for the raising of an exception to signal that fact. Indeed, this applied not just to underflow situations,[3] but to overflow as well. An exception called `REPRESENTATION_ERROR` was reserved for that purpose. Commenting on an early version of the proposal, an observer convinced the committee that it would be better to signal overflow in the usual way (i.e., by raising the predefined exception provided by Ada for that contingency) and that it would also be better to provide a result conforming to the Ada standard in cases of underflow (including flushing to zero, if nothing better could be done) instead of raising an exception. An overflow or underflow in the result of a primitive function is most likely to occur when the primitive function is used for scaling purposes in the final step of some other computation, such as that of an elementary function. In such a case, the elementary function would overflow or underflow as well, and it would be undesirable to force the latter to intercept a `REPRESENTATION_ERROR` exception arising in the former just so that it could substitute some other behavior. As the primitive functions standard is now written, an overflow or underflow occurring in the result of a primitive function called to perform scaling in the final step of the computation of an elementary function can simply be propagated from the primitive function through the elementary function to the latter's caller, which will thus satisfy the requirements of the elementary functions standard in a most efficient way.

With underflows reported by approximations and overflows signaled by the appropriate predefined exception, there was no longer any need for the `REPRESENTATION_ERROR` exception, which was accordingly eliminated. No exceptions are declared by `GENERIC_PRIMITIVE_FUNCTIONS`. Only predefined exceptions may be raised by implementations of the primitive functions, and even those are restricted (as they were in the elementary functions standard) to specific cases where they are unavoidable.

The subprograms (fourteen functions and one procedure) in `GENERIC_PRIMITIVE_FUNCTIONS` can be organized into four groups for presentation purposes. In the discussions that follow, arguments and results are of the floating-point type `FLOAT_TYPE` except where noted, and β stands for the value of `FLOAT_TYPE'MACHINE_RADIX`.

The first group comprises basic decomposition, composition, and scaling subprograms for floating-point numbers. These are the `EXPONENT`, `FRACTION`, `COMPOSE`, and `SCALE` functions and the `DECOMPOSE` procedure.

`EXPONENT` is primarily useful in argument reduction steps, where it gives a coarse indication of the magnitude of its argument. Except when `X` $= 0.0$, the function `EXPONENT(X)` delivers—as a value of the integer type `EXPONENT_TYPE`—the unique integer k such that $\beta^{k-1} \leq |X| < \beta^k$. This definition is entirely mathematical and not related to the representation of `X` on the machine. Thus, as a positive `X` decreases through the normalized range and into the denormalized range, `EXPONENT(X)` continues to decrease, even though the exponent part of the machine representation of `X` stops decreasing when the smallest normalized number is reached. In fact, on the assumption that `FLOAT_TYPE'MACHINE_EMIN` is the value of that minimum exponent,[4] `EXPONENT(X)` can return a value that is less than `FLOAT_TYPE'MACHINE_EMIN` (e.g., when `X` is denormalized). Finally, `EXPONENT(0.0)` is defined in this standard to deliver `0`.

The `EXPONENT` function can be computed on typical hardware by extracting and unbiasing the exponent field of the representation, with a special case for `X` $=$ `0.0` and with additional steps required when `X` is denormalized. `EXPONENT` corresponds closely to the IEEE recommended function `logb`, which is usually available in hardware, except that its result is of an integer type instead of a floating-point type.

Some observers contended that `EXPONENT(0.0)` should not be `0`; the most mathematically sensible alternative, $-\infty$, which can be represented on IEEE hardware at least, is ruled out by the integer-type result of `EXPONENT`. The committee staunchly preferred to stick with an integer type for the representation of the integer values delivered by this function, especially when it concluded that a result of zero for a zero argument is often a "don't care" case anyway (in the sense that the potential caller of `EXPONENT` will avoid the call and take a different path, when `X` $=$ `0.0`), and is probably harmless when not. Another alternative, raising an exception to signal an illegal argument when `X` $=$ `0.0`, was ruled out because it is unnecessarily harsh when a zero result is harmless.

The companion function `FRACTION` is also useful in argument reduction steps. For nonzero `X`, `FRACTION(X)` is defined to yield $\texttt{X} \cdot \beta^{-k}$, where k is as defined above for `EXPONENT`; `FRACTION(0.0)` is `0.0`. Thus, `FRACTION(X)` is the fraction part of the canonical form of the floating-point number `X` (normalized, however, when `X` is denormalized). This function can be computed on typical hardware by extracting the fraction field of the representation, with a special case for `X` $=$ `0.0` and with additional steps required when `X` is denormalized.

[3] For simplicity, this is understood to mean either actual underflow or merely denormalization, which is also known as "gradual underflow."

[4] This is a reasonable assumption, without which some numbers expressible in the canonical form would not be representable. It requires, however, that the definition of canonical form be relaxed to allow unnormalized fraction parts.

Often, both the exponent part and the fraction part of a floating-point number are needed in argument reduction. For such occasions, the procedure DECOMPOSE, which computes and delivers both simultaneously through a pair of arguments of mode "out," is provided.

The function COMPOSE is essentially the inverse of DECOMPOSE; it constructs a floating-point value from a given fraction and exponent part. Except when FRACTION = 0.0, COMPOSE(FRACTION, EXPONENT)—for arguments of the floating-point type FLOAT_TYPE and the integer type EXPONENT_TYPE, respectively—delivers the value FRACTION $\cdot \beta^{\text{EXPONENT}-k}$ (if it is representable), where k is the unique integer such that $\beta^{k-1} \leq |\text{FRACTION}| < \beta^k$; COMPOSE(0.0, EXPONENT) delivers 0.0 for any EXPONENT. If the defined result is not representable, then the appropriate predefined exception is raised in overflow situations, and one of the adjacent representable numbers is delivered in underflow situations. Note that the FRACTION argument is not required to be a pure fraction, with a zero exponent part (as if it had been obtained from the FRACTION function previously); rather, the fraction part of FRACTION is extracted and used to construct the result. It should be obvious that this function can be computed, on typical hardware, by appropriate manipulations of the fraction and exponent parts of floating-point quantities, as for the previous functions. COMPOSE finds representative uses in the result construction step of mathematical functions.

The remaining function of the first group, SCALE, is similar to COMPOSE; it has uses both in result construction steps and in argument conditioning (for Euclidean norms, complex arithmetic, and some matrix computations, for example). It takes arguments X and EXPONENT and returns X $\cdot \beta^{\text{EXPONENT}}$ (with the same provisions for dealing with overflow and underflow as exhibited by COMPOSE). SCALE is analogous to the IEEE recommended function scalb. When implemented by directly manipulating the exponent part of a floating-point number, it is potentially more efficient than multiplying or dividing by a power of the hardware radix, and by definition it retains full accuracy (multiplication and division, even by a power of the hardware radix, can lose accuracy on systems lacking guard digits for these operations). The function is sometimes available as a hardware operation.

The functions of the first group are not all independent. In theory, it is sufficient to have just EXPONENT and SCALE, or alternatively EXPONENT and COMPOSE; the others can be obtained in terms of these two. For greater efficiency, however, implementations should code each independently using the most direct interface to low-level representations and operations available.

The second group of subprograms comprises directed rounding functions (ROUND, TRUNCATE, FLOOR, and CEILING, all of which yield an integer value in the floating-point type FLOAT_TYPE) and an exact remainder function (REMAINDER).

ROUND, of course, delivers the value of its argument, rounded to the nearest integer, with ties being broken by choosing the even integer; this corresponds to IEEE unbiased rounding. Ada already has something comparable in its predefined conversion between floating-point and integer types. The ROUND function differs in having a floating-point result type and in specifying that ties will be broken by choosing the even integer. ROUND and the other directed rounding functions are supposed to produce their floating-point results without going through an intermediate conversion to an integer type, which could raise an exception (often the higher-precision floating-point types can accommodate larger integer values than can be represented in the available integer type of widest range). TRUNCATE simply discards the fractional part, thereby rounding in the direction of zero. FLOOR and CEILING round in the negative and positive directions, respectively. All of these functions can be programmed efficiently at a low level and might even exist as hardware operations.

The REMAINDER function delivers the exact remainder upon dividing its first floating-point argument by its second floating-point argument. More precisely, REMAINDER(X, Y) finds an integer quotient q and a remainder r such that $r = \text{X} - \text{Y} \cdot q$; it delivers r. Algorithms exist for computing the result exactly, and reasonably efficiently, regardless of the relative magnitudes of the dividend and divisor; the operation is available as a hardware instruction on some machines.

There are two customary ways of defining the quotient q, which determines the corresponding remainder r. One way defines q as the integer obtained by rounding the exact value of X/Y towards zero. This gives r the sign of the dividend and a magnitude less than that of the divisor; it is the definition used by Ada for its predefined "rem" operator on integer-type operands, yielding an integer-type result. The other way defines q as the integer *nearest* the exact value of X/Y, with ties broken by choosing the even integer. This, in turn, gives r a magnitude not greater than *half* that of the divisor and a sign that may be either positive or negative. Because the latter definition, corresponding to the IEEE rem operation, is slightly preferred by numerical analysts for such purposes as argument reduction in the arbitrary-cycle versions of the trigonometric functions, it is what has been adopted for the primitive functions standard. It is tempting to offer this function in the form of an overloading of the predefined

"rem" operator, and indeed an earlier version of the proposed standard did so. However, the two overloaded "rem" operators would have distinctly different numerical behavior (e.g., 43 rem 5 yields 3, whereas 43.0 rem 5.0 would yield -2.0), so to avoid confusion the functional form REMAINDER(X, Y) was preferred for the floating-point remainder operation in GENERIC_PRIMITIVE_FUNCTIONS.[5]

The third group of subprograms contains the PREDECESSOR, SUCCESSOR, and ADJACENT functions, which allow a floating-point machine number to be perturbed by the smallest possible amount to obtain the next larger or smaller machine number. The principal use for these functions is in testing mathematical software, where very fine control over test arguments is sometimes needed. As defined, they are also useful for generating the machine numbers (denormalized, if the hardware has that capability) adjacent to zero.

PREDECESSOR and SUCCESSOR are one-argument functions that deliver the machine number adjacent to their argument in the direction inferred from the name, whereas ADJACENT is a two-argument function that returns the machine number adjacent to its first argument in the direction of the second argument. The latter function is provided for applications in which the direction of motion is not known in advance and needs to be determined dynamically; it is identical to the IEEE recommended function nextafter. There is another difference between ADJACENT and the other two functions: PREDECESSOR and SUCCESSOR raise the predefined exception signaling overflow upon an attempt to move beyond the first or last floating-point machine number, while ADJACENT never raises an exception (it is not possible to move beyond the range of machine numbers with it). The committee debated whether it was extravagant to have both sets and found itself split into two camps, neither of which wanted to give up its preferred choice. It was argued that one could not be assured of obtaining the other set if only one set were provided, because of a well-known weakness in the Ada model of floating-point arithmetic that makes the comparison of nearby floating-point numbers indeterminate.

The final group of subprograms contains miscellaneous functions—namely, COPY_SIGN and LEADING_PART.

The COPY_SIGN function, often found in other languages and represented in LCAS by sign and in the IEEE recommended functions by copysign, delivers the value obtained by transferring the sign of its second argument to the first (but otherwise retaining all the precision of the first argument). This function is often useful in giving the final result of some computation the appropriate sign (without resorting to an if-then-else test) after having stripped the sign away in the argument reduction step, perhaps by using the very same function to set it positive there. In highly accurate and portable code, this function is preferable to negation and the abs operator because those can lose low-order digits on hardware lacking a guard digit for subtraction. On hardware distinguishing the sign of zero (such as IEEE hardware), and where the implementation of GENERIC_PRIMITIVE_FUNCTIONS chooses to exploit the capability of signed zeros, COPY_SIGN is required to distinguish between plus zero and minus zero for its second argument; thus, it confers the sign of its second argument on the result even when the second argument is zero. COPY_SIGN was a late addition to GENERIC_PRIMITIVE_FUNCTIONS.

LEADING_PART, another late addition, was motivated by the LCAS trunc operation. It delivers the value of its first argument with only some of the leading radix-digits retained (the number of them given by the value of the second argument, which is of the predefined type POSITIVE), and with the remaining radix-digits—the low-order digits—replaced by zeros. This function plays a leading role in sophisticated strategies for simulating higher precision, where a floating-point number needs to be decomposed into a major portion of limited precision and an additive residue. The leading part is usually used as a factor in a subsequent multiplication by a small integer, such that the result has a sufficiently small number of radix-digits to be represented *exactly* within the model of floating-point arithmetic. The residue can be accurately obtained by subtraction, assuming the starting value has no more precision than that of safe numbers.

Two functions in the earliest versions of the proposed standard, both taken from [3], were dropped along the way. These were RECIPROCAL_REL_SPACING(X) and ABS_SPACING(X), which give information about the spacing of machine numbers in the neighborhood of X. Although they are useful for fine control over the termination of an iterative algorithm, or for measuring and reporting error, committee members did not find them important enough to retain; when the committee was unable to justify their inclusion to the satisfaction of some observers, it decided to omit them.

The relationship between functions in GENERIC_PRIMITIVE_FUNCTIONS and certain required operations or recommended functions of the IEEE floating-point standards has been mentioned repeatedly in this rationale. It is

[5]The preceding discussion only scratches the surface of the long and involved history of this operation. Many other alternatives, some of which made their way into earlier drafts of the standard, were considered at one time or another.

anticipated that relevant functions in GENERIC_PRIMITIVE_FUNCTIONS will serve as the realization of some of the functionality of the proposed IEEE binding for Ada [4].

The relationship between functions in GENERIC_PRIMITIVE_FUNCTIONS and some of the features of LCAS has also been discussed. The fair degree of overlap between the two has prompted the suggestion that everything in LCAS that is not already built into Ada should be available in GENERIC_PRIMITIVE_FUNCTIONS in a compatibly defined way. The obvious benefit of following that suggestion to the letter would be the ability of an LCAS binding for Ada to point to GENERIC_PRIMITIVE_FUNCTIONS as the embodiment of that part of its functionality not built into Ada. Unfortunately, this goal was not expounded early enough in the development of GENERIC_PRIMITIVE_FUNCTIONS, and a few differences remain.

Several partial implementations of GENERIC_PRIMITIVE_FUNCTIONS, varying in the degree to which they exploit knowledge of the underlying machine, exist. Some of them have tried to be relatively general, that is, adaptable to different architectures by suitable choice of parameters; none have yet tried to be as efficient as possible.

Through a report [16] to the Ada 9X Requirements Team, the SIGAda Numerics Working Group has had an influence on Ada 9X as a result of the work it did in developing the proposed primitive and elementary functions standards. The report contains a discussion of the problems of writing high-quality, portable mathematical software; it included a number of Ada 9X revision requests aimed at solving some of these problems. One of the recommendations was to include the functionality of GENERIC_PRIMITIVE_FUNCTIONS in the Ada language, in the form of attributes (of the function kind). Several of the issues discussed in the report have been accepted by the Ada 9X Requirements Team as requirements for Ada 9X [2], including the incorporation of both elementary functions and primitive functions in optional annexes in Ada 9X.

References

[1] ACM SIGAda Numerics Working Group. Proposed Standard for a Generic Package of Primitive Functions for Ada, December 1990. Draft 1.0.

[2] Ada 9X Project. Ada 9X Requirements. Office of the Under Secretary of Defense for Acquisition, Washington, December 1990.

[3] W. S. Brown and S. I. Feldman. Environment Parameters and Basic Functions for Floating-Point Computation. *TOMS*, 6(4):510–523, December 1980.

[4] R. B. K. Dewar. Proposed Ada Interface for the IEEE Standard for Binary Floating-Point Arithmetic, August 1989. Version 4.

[5] K. W. Dritz. Rationale for the Proposed Standard for a Generic Package of Elementary Functions for Ada. ANL Report ANL-89/2 Rev. 1, Argonne National Laboratory, Argonne, Illinois, October 1989. A later (December 1990) revision is available from the author.

[6] B. Ford. Parameterization of the Environment for Transportable Mathematical Software. *TOMS*, 4(2):100–103, June 1978.

[7] IEEE. IEEE Standard for Binary Floating-Point Arithmetic. ANSI/IEEE Std. 754-1985, IEEE, New York, 1985.

[8] IEEE. IEEE Standard for Radix-Independent Floating-Point Arithmetic. ANSI/IEEE Std. 854-1987, IEEE, New York, 1987.

[9] ISO-IEC/JTC1/SC22/WG9 (Ada) Numerics Rapporteur Group. Proposed Standard for a Generic Package of Elementary Functions for Ada, March 1989. Draft 1.0.

[10] ISO-IEC/JTC1/SC22/WG9 (Ada) Numerics Rapporteur Group. Proposed Standard for a Generic Package of Elementary Functions for Ada, December 1990. Draft 1.2.

[11] P. Naur. Machine Dependent Programming in Common Languages. *BIT*, 7:123–131, 1967.

[12] M. Payne, C. Schaffert, and B. Wichmann. Proposal for a Language Compatible Arithmetic Standard. *SIGPLAN Notices*, 25(1):59–86, January 1990.

[13] M. Payne, C. Schaffert, and B. Wichmann. Proposal for a Language Compatible Arithmetic Standard. *SIGNUM Newsletter*, 25(1):2–43, January 1990.

[14] K. A. Redish and W. Ward. Environmental Enquiries for Numerical Analysis. *SIGNUM Newsletter*, 6(1):10–15, 1971.

[15] J. Reid. Functions for Manipulating Floating-Point Numbers. *SIGNUM Newsletter*, 14(4):11–13, December 1979.

[16] J. Squire. Special Study Report on Ada Numerical Issues. Unpublished material, October 1989.