The Design of Floating-Point Data Types

DAVID GOLDBERG
Xerox Palo Alto Research Center

The issues involved in designing the floating-point part of a programming language are discussed. Looking at the language specifications for most existing languages might suggest that this design involves only trivial issues, such as whether to have one or two types of REALs or how to name the functions that convert from INTEGER to REAL. It is shown that there are more significant semantic issues involved. After discussing the trade-offs for the major design decisions, they are illustrated by presenting the design of the floating-point part of the Modula-3 language.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs—*data types and structures*; G.1.0 [**Numerical Analysis**]: General—*computer arithmetic*; *error analysis*

General Terms: Design, Languages

Additional Key Words and Phrases: Ada, backward error analysis, error analysis, exceptions, floating point, floating-point standard, FORTRAN 90, guard digit, Modula-3, precision, rounding, ulp

1. INTRODUCTION

There has been surprisingly little discussion of the issues surrounding the semantics of floating-point data types. For example, the past issues of the Principles of Programming Languages (POPL) proceedings do not contain any papers on language design for floating point. This paper presents some of the interesting semantical questions involved with floating point. Although there are a few papers on the interaction between floating point and programming languages (e.g., [8] and [10]), they mainly consider how to deal with languages that already exist. This paper presents some of the issues that arise when designing any new language that allows the manipulation of floating-point types.

One goal of high-level programming languages is to provide a way to express an algorithm so that it runs correctly on a wide range of machines (ideally, on any machine for which a compiler exists), without losing too much efficiency. This presents a problem for numerical programs (which in this

Author's address: Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304.

paper means programs that use floating point). To quote Hoare [12]:

leaving certain aspects of a language *undefined*, for example, range of integers, accuracy of floating-point, and choice of overflow technique... is absolutely essential for standardization purposes, since otherwise the language will be impossible to implement efficiently on differing hardware designs.

Languages that follow Hoare's philosophy cannot express algorithms that depend on the detailed behavior of floating point. In this paper we argue that (1) there are practical algorithms that depend on detailed floating-point behavior; (2) thus, very general semantics (such as those in Ada [2]) are not precise enough; and (3) however, there are semantics that support algorithms sensitive to detailed floating-point behavior without losing too much efficiency.

The paper is organized as follows: Section 2, after first giving some background on numerical error analysis, identifies the key issue for the semantics of arithmetic operations, namely, whether a language semantics will support invariants involving forward error bounds or backward error bounds.

Section 3 presents three patterns that a language design might follow: Brown's model, a forward model, and an exact model. Brown's model, which is used in Ada, supports backward but not forward error bounds. The forward model is more precise, supporting both types of bounds. The exact model has recently become practical because of the widespread adoption of the IEEE floating-point standard and is the most precise of all.

Support of error analyses is not the only thing that must be considered in the design of floating-point data types. Section 4 discusses three other issues: specifying precision, handling exceptions, and the precise definition of rounding. Finally, Section 5 illustrates the design choices that have been presented by describing the design of the floating-point system in Modula-3 [15].

2. WHAT SHOULD FLOATING-POINT SEMANTICS BE?

There is a fundamental difference between integer and symbolic programs on the one hand, and floating-point programs on the other. In the first category, the programs have discrete results and are intended to produce identical results on all machines. In the second case, the exact same answer on all machines is not necessarily expected, because of rounding errors.

One way to make sense of what floating-point semantics should be is to determine what kind of invariants a floating-point program should satisfy. In general, these invariants do not involve equalities. For example, a zero-finding program would not be expected to find the exact zero of a function (since the zero might in fact not even be a rational number); rather, some bounds on the answer are desired.

2.1 Background on Error Analysis

An error analysis needs a unit of measure for expressing rounding error. This paper uses *ulps*, or *u*nits in the *l*ast *p*lace. To illustrate, on a decimal machine with 5 digits of precision, the numbers within 1 ulp of 234.56 are 234.55 and 234.57. A way to distinguish between exact mathematical functions and the computed value of a function is also needed. An ideal function is represented by $f(x)$, whereas $\mathbf{F(x)}$ represents the computation of the function executed on a real computer.

Numerical analysts have identified two types of error bounds, *forward* and *backward*. A forward bound is the simplest to understand: It tells you how close the real value is to the computed one. For example, by analyzing the code for **F** and knowing the details of arithmetic on machine X, you might be able to show that there is a forward bound $|f(x) - \mathbf{F(x)}| < 5$ ulps, which is always satisfied when **F** is computed on X. Then, if $\mathbf{F(x)}$ is computed as 234.56, the true answer $f(x)$ must be between 234.61 and 234.51.

Unfortunately, many important algorithms do not satisfy a small forward bound. Consider the simple formula $x^2 - y^2$. Even if machine X has the best possible arithmetic, that is, an arithmetic where each arithmetic operation produces a result as close as possible to the exact result, no small forward error bound is possible. The reason is that the computation of x^2 and y^2 generally has a small amount of rounding error, and when $x \approx y$, the subtraction $x^2 - y^2$ cancels most of the digits, leaving behind mainly the rounding errors from the squaring operations. Thus, the final result could be in error by a very large amount.

For algorithms where there is no small forward bound, it may still be possible to produce a small backward bound. A backward bound is a bound on the *input*. For example, a backward bound of 5 ulps for $f(x)$ on machine X means that, for every input value x, there is an x' with $|x' - x| < 5$ ulps so that $f(x') = \mathbf{F(x)}$.

To clarify, here are a few more examples of formulas and their error bounds. The formula $x^2 - y^2$ has a cancellation when $x \approx y$ that cannot be avoided, so the best that can be done is a backward bound. But a slightly different algorithm, $(x - y)(x + y)$, does satisfy a small forward bound (on most machines). For some problems, there are no practical algorithms that have a small forward bound (but that do satisfy a small backward bound). Gaussian elimination is one example.[1] Another is the quadratic formula $r = (-b \pm \sqrt{b^2 - 4ac})/2a$ when the roots are almost equal, that is, $b^2 \approx 4ac$. In this case there is no small forward bound possible, although a small backward bound can be found.

2.2 Forward versus Backward

If you want to guarantee that an algorithm written in some language is correct (i.e., provide an error bound for it) and if that language does not spell out the semantics of floating point, you need to do the following for every machine X: First, find out how the compiler for X converts floating-point constructs into machine instructions. Then look in the hardware architecture

[1]Actually, there is a twist to Gaussian elimination, in that the backward bound depends on a number (easily) computed during Gaussian elimination. In practice, that computed number is always very small. See [9] or [11] for an account of this.

manual to find the details of floating-point arithmetic on X. Finally, do an error analysis for your algorithm using the results of these steps.

In contrast, if your algorithm is written in a language that provides floating-point semantics, then you only need to do the analysis once. The algorithm will be correct on all machines, as long as your error analysis used only facts guaranteed by the language semantics.

With this background, a key design decision for floating-point semantics can be identified. The semantics should be general enough so that they can be efficiently implemented on most machines, but strict enough so that they can be used to prove algorithms correct, that is, so that they satisfy an error bound. The key question is therefore whether floating-point semantics should provide enough information to prove forward bounds or backward bounds.

Most languages that provide semantics for floating-point types support only backward bounds (e.g., Ada). The arguments for backward bounds are as follows:

—Backward bounds are less sensitive to details of machine arithmetic, so it is easier to provide portable semantics that enable backward bounds than it is for forward bounds.

—When the input to a numerical algorithm is based on physical measurements, that input will have some experimental error; thus, backward analyses are more appropriate.

—Even if the input is exact when expressed in decimal, many decimal numbers cannot be exactly expressed in binary, so the input will be perturbed by the decimal-to-binary conversion process.

However, on closer analysis, there are many cases where backward bounds are insufficient. Some of the types of problems that require tighter control on arithmetic are as follows:

—*Binary–decimal conversion*. In general, these conversions have rounding error. But good conversions should be sufficiently accurate that converting a number from binary to decimal and back will recover the original number.

—*Monotonicity*. Although you do not expect mathematical functions to be exact, you do expect familiar properties to be maintained. The preservation of monotonicity is common: If $x < y$, you would like, for example, the computed values of EXP to satisfy $\mathrm{EXP}(x) \leq \mathrm{EXP}(y)$.

—*Multiple-precision arithmetic*. Multiple-precision arithmetic is often implemented in assembly language in a highly machine-specific way, but there is a portable method of doubled precision based on storing each double-precision quantity as a pair of floating-point numbers (x_h, x_l) and using ordinary floating-point operations on each of x_h and x_l [16]. The double-precision quantity this pair represents is the number that is the exact sum of x_h and x_l. When operating on such pairs, if x_h has the high-order bits, then the least significant bits of x_h are very important, because they represent bits midway in the representation of the double-precision quantity.

—*Interval arithmetic*. Sometimes you would like to have an exact (but possibly pessimistic) bound for the result, that is, to be guaranteed that it lies in an interval. This requires some precise control of arithmetic. An important special case of this is zero finding: You might like to have an interval within which a zero is guaranteed to lie.

The arguments for forward error analysis are not merely theoretical. For example, Tang is producing efficient algorithms for the basic mathematical functions (e.g., EXP, LOG) that satisfy provable forward error bounds ([18] is the first in the series). His EXP routine is provably accurate to within 0.77 ulps. The proofs are easy to follow and take only a few pages. Clinger [6] has a decimal-to-binary conversion that recovers the original input. It uses floating point in part and is therefore more efficient than algorithms that run entirely in multiple-precision integer (e.g., "bignum") arithmetic. Like Tang's algorithm, its proof of correctness is based on forward error analysis.

3. THREE STYLES OF FLOATING-POINT SEMANTICS

In order to support error bounds, a language must specify the accuracy of the four basic arithmetic operations ($+, -, \times, /$). There are three styles for doing this: Brown's model, a forward model, and an exact model.

3.1 Brown's Model

The first style is typified by the work of Brown [3], which provides a parameterized model of floating-point arithmetic. Virtually all hardware floating-point arithmetics can be modeled by plugging the right parameters into Brown's system. Thus, a language can provide floating-point semantics with inquiry functions that report the parameters for the machine it is running on. Brown's model has been implemented in the Ada programming language [2], so to be concrete, Ada is discussed rather than the abstract Brown model.

A floating-point number is a real number that can be exactly represented in the computer's floating-point format. Given floating-point numbers x and y, the Ada model specifies constraints that must be satisfied by the computed value of x *op* y. It does this using *model numbers*, which are a subset of a machine's floating-point numbers. In the special case when the floating-point numbers x and y happen to be model numbers, x *op* y must be computed to be between the two model numbers on either side of the exact answer.[2] In general, x and y are each enclosed in the smallest possible *model interval*, that is, an interval whose endpoints are model numbers. The result of x *op* y must be contained in the smallest model interval that contains all of the x' *op* y', where x' ranges over the model interval for x, and similarly for y'. Because the Ada model does not let one pin down the exact value of the

[2] If the exact value x *op* y is a model number, then the operation must return that model number.

inputs, it cannot support forward error bounds.[3] Besides the fact that they do not support forward error bounds, these semantics have a bizarre consequence. Since the result of a floating-point operation is not exactly specified, but merely has to lie in a certain range, the Ada standard allows a floating-point operation such as 1.0/10.0 to give one answer in one place in a program and a different answer in another place (or time) in that same program. Thus, Ada does not even satisfy the basic requirement that, when a deterministic program is run twice, it produces the same answer each time.[4]

Ada provides a hook that allows forward error bounds, namely, the attribute **MACHINE_ROUNDS**, which is true if arithmetic always rounds (as IEEE and VAX arithmetic do). However, this is a very crude hook. For example, many algorithms satisfy small forward bounds when run on an IBM/370, but this arithmetic does not round, so this case is not detected by the **MACHINE_ROUNDS** attribute. Furthermore, this attribute seems somewhat out of place in Ada, dwarfed as it is by the model number mechanism.

3.2 A Forward Model

Because there are many interesting problems for which backward error bounds are too weak, it is worth considering a model like Ada's, but one that supports forward analyses rather than backward ones. What model numbers effectively do is allow you to ignore the low-order bits of the *input*. For example, if there are 48 bits of significand, but the low-order 4 bits are unreliable, then model numbers would be declared to be those whose low-order 4 bits are always zero.

What about ignoring the low-order bits of the *output*? Imagine an alternate definition of model numbers, where the result of an operation was required to lie in the model interval that contained the exact answer.

To see why Brown did not adopt this idea for his model, *guard digits* need to be explained. For example, to compute $1.01 \times 10^1 - 9.97$, the number 9.97 is shifted to make the decimal points align, thus changing the calculation to $(1.01 - .997) \times 10^1$. However, now .997 has an extra third digit to the right of the decimal point. This can be either saved in a *guard digit* or dropped. If there is no guard digit, then the calculation becomes $(1.01 - .99) \times 10^1 = 2.00 \times 10^{-1}$, compared to the correct answer of 1.30×10^{-1}, which is quite wrong, in error by 70 ulps. In contrast, a machine with a guard digit will never make an error larger than 1 ulp.

[3]For example, if x and y are nearby numbers, then in Ada you cannot provide a forward bound for $x - y$, because perturbing x and y around their model intervals can make dramatic changes to the value of $x - y$.

[4]To make this argument concrete, consider a compiler for Ada on a machine with rounding modes. Since Ada semantics are satisfied in any rounding mode, an optimizing compiler might not reset the modes at the start of execution. Thus, the outcome of an Ada program would be nondeterministic, depending on what state the rounding modes were in when the program was executed.

In Brown's model, the idea is to have a parameterized system, where as arithmetic gradually gets "worse," one slowly cranks up a parameter (in this case, the number of bits ignored by model numbers). Until recently, a common way to weaken arithmetic was to leave out the guard digit. In a floating-point format using base 2 and p digits of fraction, subtraction without a guard digit can have results in error by almost 2^{p-1} ulps. Thus, when using the forward version of model numbers, the hardware change of deleting one guard digit results in model numbers going from a spacing of 1 ulp to a spacing of 2^{p-1} ulps. In other words, forward model numbers cannot detect the difference between an arithmetic whose only flaw is the lack of a guard digit, from one that always makes large errors in subtraction.

It is very rare for modern machines to lack a guard digit (the Cray family is the only major machine without them). Thus, if we were willing to have semantics that did not work well for Cray machines, the forward version of model numbers proposed above would be a feasible alternative to Brown's model. As examples, for IEEE arithmetic and VAX arithmetic, the forward model numbers would be all floating-point numbers. For IBM/370 arithmetic, the forward model numbers would have one digit less than the hardware floating-point numbers (see [17] for an analysis of IBM/370 arithmetic).

3.3 An Exact Model

A third way to provide semantics is to specify precisely the results of each arithmetic operation. That is, a precise semantics would specify an algorithm for each of the four basic operations and require that the result of each operation match the results of this algorithm bit for bit.

There is an obvious difficulty with this approach, as noted by the Hoare quotation in the introduction. Until recently, not only did every computer manufacturer provide a different style of computer arithmetic, but some manufacturers even had different arithmetics on different models. It was traditional for numerical analysis texts to provide a table (e.g., [7, p. 8]) listing floating-point parameters for a sampling of computers. With such a proliferation of arithmetics, it was impractical to have them all well documented and to expect users to understand all that documentation. On the other hand, if one universal algorithm were provided, then compilers for machines whose hardware did not provide results matching this algorithm would have to generate a very inefficient emulation.

However, the situation has changed recently. Every major new architecture introduced in the past 10 years has used IEEE arithmetic, and currently the majority of existing machines use IEEE arithmetic [14].[5]

[5]The IEEE floating-point standard falls roughly into two parts. First, there are specifications for the formats for floating-point numbers and for the results of $+$, $-$, $\times$, and $/$. Second, there are rules about accessing and modifying rounding modes, exception status bits and trap enable bits, and rules for other operations like square root and binary-to-decimal conversion. When vendors say they support the IEEE standard, they are usually referring only to the first of these parts.

This widespread use of one arithmetic suggests that an exact model is now practical. Either IEEE arithmetic could be chosen as the universal standard (the recently announced DEC Alpha machines will offer hardware IEEE emulation), or a small set of well-documented arithmetics could be chosen. For example, one approach would be to provide an inquiry function returning one of, say, **IEEE**, **VAX**, **370**, specifying which arithmetic was in use. Precise specifications of each of these arithmetics are widely available, and the vast majority of computers implement one of them. For those computers to which it applies, this model provides much more precise information than the parameterized models of Sections 3.1 and 3.2. The few machines that are not on the list will mostly be those with poor arithmetic (e.g., Crays). Parameterizing precise semantics via an inquiry function returning one of **IEEE**, **VAX**, **370** is not all that different from using model numbers, which also requires an inquiry function, namely, one that returns the number of digits in the model numbers.

The problem with this approach to precise semantics is deciding what machines to support. The majority of non-IEEE machines use either VAX arithmetic or IBM/370 arithmetic. One sign that these are the three prevalent arithmetics is the existence of floating-point chips (such as the AMD Am29C327) that support exactly these three types of arithmetic. So the solution above, **IEEE**, **VAX**, **370**, is natural. If the list were to be expanded, the next obvious architecture to add would be the Cray. Unfortunately, the Cray family does not have a rigorously defined architecture, and different models of Cray have (slightly) different arithmetics. A real program that behaves significantly differently on the Cray XMP compared to the Cray 2 is discussed by Carter [5].

4. OTHER ASPECTS OF FLOATING-POINT ARITHMETIC

So far the discussion has concentrated on the behavior of the four basic floating-point operations. We have argued that models based on backward analyses like the one in Ada do not give precise enough information to prove error bounds for algorithms, and that either a forward model or an exact hardware-based model is more appropriate.

This is only one of the issues that a floating-point semantics needs to cover. Other important ones are the following:

(1) How should floating-point precision be specified?
(2) How are underflow, overflow, and other exceptions handled?
(3) What is the precise definition of rounding?

In the following subsections, alternatives for each of these issues are discussed. In the final section, the particular set of decisions made for Modula-3 is presented.

4.1 Specifying Precision

There is nothing more frustrating to a programmer than to be given a computer with some important and useful functionality that cannot be accessed from software. One example of this is extended precision (i.e., a floating-point type with more precision than double precision). Until recently, most languages provided (at most) two floating-point types, and when using those languages on machines with extended-precision hardware, there was simply no practical way to access extended precision.

An obvious way to solve this problem is to introduce a third floating-point type. Although this is simple, some people find it inelegant. Why three types? In addition, languages that do not support generic procedures have to replicate floating-point functions and operators three times, once for each type.

An alternative approach is to provide a whole family of types. In both Ada and FORTRAN 90 [1], this family is parameterized by the number of decimal digits of precision. Although this approach is more elegant in that it does not involve the special number 3, it is also awkward. First of all, it is not a good match to computer hardware, virtually all of which has at most three fixed types. Second, there is the problem of determining the type of a floating-point constant (i.e., literal). FORTRAN 90 solves this by introducing a symbolic name (called a *kind*) for each type and then by postfixing constants with _**name**, where **name** is a named integer constant specifying the type.

In more detail, if you know that you want some fixed amount of precision (say, 12 decimal digits), in Ada you can declare a variable to be **REAL is digits 12**. Then you may want to make an inquiry to find out how many digits you really got, since Ada can give you anything that has at least 12 digits. FORTRAN 90 is similar, except that there is an enforced level of indirection. The kind associated with each real type is an integer, with a kind of 0 corresponding to the type **REAL**. The declaration **REAL (KIND = k)** declares variables to be of the real type corresponding to **k**.

The effective functionality of all of these systems is similar. In each case, one is not guaranteed to get a type of the specified precision, but instead only a type (if one exists!) that has at least as much precision as specified. And, in each case, one cannot compute a precision n at run time and then directly ask for a type with precision n. The argument to **digits** in Ada and the **KIND** argument in a FORTRAN declaration must both be compile-time constants.[6]

4.2 Exceptions

Traditionally, programs abort computation when an overflow or division by zero occurs. However, it is not hard to think of situations where this is inconvenient. One obvious example is an environmental query: trying to discover the largest representable number by computing ever larger numbers until overflow occurs.[7] Another example (discussed in [10]) involves functions that take functions as parameters, such as a zero finder. If the zero finder accidentally probes outside the domain of the function, it would be nice if the computation did not abort. Currently, systems avoid this problem by requir-

[6]There have been proposals for languages with dynamic precision control, for example, [13].

[7]Although a well-designed language offers the user a way to discover the largest number more easily.

ing the user to specify a valid domain for the function, but this can be inconvenient for a function with many single-point singularities.

By default, the IEEE standard calls for exceptional operations to return *special values*, such as NaN or Infinity, and to continue the computation. This has the advantage of being language independent. Thus, a language might deal with exceptions by simply supporting the IEEE special values.

However, it is convenient for languages to map exceptional floating-point operations to a language exception mechanism (if there is one) for two reasons. First, this provides support to machines that do not have IEEE arithmetic and whose only alternative to aborting the whole computation is to raise an exception. Second, the IEEE standard strongly recommends (but does not require) that IEEE implementations allow the installation of trap handlers. Probably the most important application of trap handlers is to improve performance. For example, suppose a computation like sin x/x is in an inner loop, and the case $x = 0$ is very rare. Rather than test for $x = 0$ each time through the loop, it will be faster to eliminate the test and to deal with the case $x = 0$ in the trap handler.

A common language mechanism for handling exceptions is to allow users to declare a handler surrounding a block of code. Then, if an exception is raised in that block, execution in the block is terminated, and control transfers to the handler (this is the model in Ada and Modula-3). Mapping floating-point exceptions into language exceptions of this kind is useful and natural. However, note that this does not correspond exactly to the recommended functionality of the standard. The IEEE standard recommends that, when a trap handler is enabled, the return value of the trap handler be used as the result of the operation that trapped, which would require that, after executing the handler, control return to the block where the exception occurred.[8]

4.3 The Definition of Rounding

Rounding occurs in programming languages in the conversion of one numeric type to another (e.g., when converting from a float type to an integer, or from double precision to single precision). Language definitions are frequently vague on the precise definition of rounding. There are two common ambiguities. One has to do with halfway cases: Do they round up or down? Most language definitions do not say. There are a few algorithms that depend on the definition of rounding [10], and although so far they are rather specialized, with increasing use of the IEEE standard (which strictly prescribes rounding), more may be discovered.

The other ambiguity has to do with rounding modes. The IEEE standard provides four rounding modes (infinity, zero, minus infinity, and nearest). The manipulation of rounding modes can be useful for exact algorithms (such as computing a correctly rounded square root), as well as for interval

[8]It is not unreasonable for a language to fail to support the IEEE-recommended behavior. Resumable exceptions are usually very difficult to implement on heavily pipelined machines, which may not save enough state to determine exactly where the interrupt occurred.

arithmetic. In most languages, it is ambiguous how the built-in conversion functions behave on IEEE machines: Do they always round to nearest, or do they follow the current rounding mode?

5. AN EXAMPLE: MODULA-3

The language Modula-3 is used to illustrate the preceding discussion. The early language specification [4] was typical of most language reports in that floating point was very cursorily defined. The latest version [15] has floating-point semantics defined along the lines suggested by this paper.

In brief, Modula-3 supports forward error analyses using the exact semantics of Section 3.3, it has three fixed floating-point types, it maps floating-point exceptions to Modula-3 exceptions, and it precisely defines the meaning of all rounding operations.

5.1 Rounding

Floating-point semantics are captured in a series of required interfaces. The roundoff error in the basic floating-point operations is controlled by **RoundDefault** (a constant of type **RoundingMode** in one of the required interfaces), which controls which of {IEEE, Vax, IBM/370, Other} rounding rules are followed. In the case of IEEE arithmetic, the type **RoundingMode** is also used to describe the current rounding mode. The declaration is

TYPE **RoundingMode** = {MinusInfinity, PlusInfinity, Zero, Nearest, Vax,
IBM370, Other}.

The built-in ROUND function (which converts from floating-point types to integers) always rounds to nearest, with the functions FLOOR, CEILING, and TRUNC providing the other three rounding modes. Thus, on machines with IEEE arithmetic, there is no built-in function that converts from float to integer according to the current rounding mode. However, it is easy to write a procedure that does this. For conversion between floating types, the FLOAT function obeys rounding modes.[9]

Discussion. The restriction to {IEEE, Vax, IBM/370, Other} seems reasonable, given that the primary target for Modula-3 is workstations. The portable public domain Modula-3 compiler from the DEC Systems Research Lab (SRC) has been ported to many machines, all of which have either IEEE or VAX arithmetic.

Although it is unfortunate that the styles of float-to-integer conversions and conversions between float types are not consistent, it is straightforward to perform all possible types of conversions.

5.2 Specifying Precision

Modula-3 has three floating-point types rather than a family. To express the constant 1.0 as a literal, one of **1.0e0**, **1.0d0**, or **1.0x0** is used, depending on

[9]Actually, [15] defines FLOAT to always round to nearest, but this appears to be an error that will probably be changed in the next edition.

whether the constant is REAL, LONGREAL, or EXTENDED, respectively. When no suffix is provided, **e0** is assumed.

The language has some built-in generic functions such as FLOAT, which converts between floating-point types, and ABS, which takes the absolute value of a number. However, there is no overloading, and so for user-defined functions (which includes the math library of EXP, SIN, etc.), there must be a different function for each type. Thus, the math library will come in three different flavors.[10]

Discussion. Modula-3 is strongly typed and only allows implicit conversions between types that are *subtypes* of one another. The language spells out exactly which types have the subtype relation, but in general, T1 is a subtype of T2 if every member of T1 is a member of T2, and if T1 and T2 are intended to be represented in the same way. This means that, even if the assignment **x** := **y** involves an implicit type conversion, it can still be implemented by copying. This representation restriction explains why CARDINAL is a subtype of INTEGER, but there is no subtype relation between REAL and LONGREAL.

The above facts mean that there are no implicit conversions between floating-point types, and thus, the introduction of a family of floating-point types would require introducing families of floating-point constants and also families of functions taking floating-point parameters. This explains why Modula-3 uses the three-type model rather than the family-of-types model.

The biggest problem with this solution to the precision problem concerns literals. It is awkward to write a generic implementation of a numeric procedure, because if it requires literals, the literals must be written differently for the REAL, LONGREAL, and EXTENDED versions.

5.3 Threads

Modula-3 has multiple threads of control (sometimes called lightweight processes) that share an address space. Routines that change the floating-point state (such as **SetRounding** and **SetFlags**) do so on a per-thread basis.

Discussion. Thread scheduling in Modula-3 may be preemptive; that is, the processor may asynchronously switch between threads. This explains why changes to the IEEE state are made on a per-thread basis. If an asynchronous thread switch occurs in the middle of a computation and if the other thread changes the floating-point state, the floating-point state of the original thread must be restored when it is resumed. If the IEEE state is not maintained on a per-thread basis, then all floating-point computations have to be executed as critical sections, which is not only impractical, but ruins any chances of concurrency on multiprocessors.

[10] There is a simple generic facility, so even though there must be three math libraries, they can all be instantiations of a single generic library.

5.4 IEEE Support

There are required interfaces that supply routines for reading and writing the rounding modes, the exception status flags, and the trap enable flags. On non-IEEE machines, setting modes that do not exist raises the Modula-3 exception **Failure.**[11]

To handle exception traps and flags on both IEEE and non-IEEE machines, there is a **Behavior** type, which has the values {Trap, SetFlag, Ignore}. Each exception has a behavior. For example, on machines that do not detect underflow, but silently flush to zero, GetBehavior(Flag.Underflow) = Ignore. On machines with IEEE arithmetic, GetBehavior is initially SetFlag for all exceptions. If a machine supports a trap handler for overflow (whether it has IEEE arithmetic or not), then SetBehavior(Flag.Overflow, Flag.Trap) will cause overflow to trap, raising the Modula-3 **Trap** exception. If the machine does not support a trap handler for overflow, then SetBehavior will raise the **Failure** exception.

This has a nice application to integer arithmetic. The **Flag** type in Modula-3 has a value for integer overflow, so the behavior mechanism detects whether integer arithmetic is modulo 2^{32} or signals overflow (as well as gives a way to change that behavior, when possible).

Discussion. It was a goal of the Modula-3 design to support IEEE arithmetic well, while still being applicable to non-IEEE machines. One approach would be to define an IEEE interface that would only appear on IEEE implementations. The solution above seems preferable for several reasons. First, having an interface that only appears on IEEE machines means that portable code would have to come in two versions. With the Modula-3 approach, a single version can handle both cases.[12] Detecting non-IEEE machines can be done by testing the **FloatMode.IEEE** constant or with a handler for the **Failure** exception. Second, since many IEEE functions make sense on non-IEEE machines (e.g., **IsNaN** and **GetBehavior**), it seems preferable to have one version of these functions, rather than separate ones for IEEE and non-IEEE machines.

6. SUMMARY

Some numerical algorithms are relatively insensitive to floating-point semantics (e.g., Gaussian elimination), whereas others, such as the examples listed in Section 2.2, are quite sensitive. Implementing reliable, portable algorithms to solve sensitive problems is difficult in most programming languages, because the languages are either silent about the semantics of floating-point arithmetic or specify a semantics that is too general to support forward error analyses.

A floating-point semantics that is suitable for a range of numerical programs should address not only the behavior of the basic arithmetic opera-

[11]According to [15], **Failure** is in the raises clause of **SetBehavior** and **SetRounding**, but not **SetFlags**. Its omission in **SetFlags** appears to be an error and will probably be changed in the next edition.

[12]A language with a preprocessor (e.g., C's #ifdef) would allow a single source file to handle both versions, but the Modula-3 language does not define a preprocessor.

tions, but also things like the exact definition of rounding and exceptional conditions such as overflow. For each of these areas, we have discussed some practical ways to provide a semantics. To illustrate the trade-offs between these choices, we have given a detailed look at the floating-point semantics of Modula-3, providing a rationale for the decisions taken in that language.

ACKNOWLEDGMENTS

A number of people read drafts of this article and provide useful feedback that helped me to improve it, especially Alan Demers, Jim Demmel, John Gilbert, and the referees.

REFERENCES

1. ANSI. FORTRAN. S8(X3.9-198x), version 112, American National Standards Institute, New York, June 1989.
2. ANSI. Reference manual for the Ada programming language. ANSI/MIL-STD 1815A, American National Standards Institute, New York, 1983.
3. BROWN, W. S. A simple but realistic model of floating-point computation. *ACM Trans. Math. Softw. 7*, 4 (Dec. 1981), 445–480.
4. CARDELLI, L., DONAHUE, D., GLASSMAN, L., JORDAN, M., KALSOW, B., AND NELSON, G. Modula-3 report (revised). Tech. Rep. 52, Digital Equipment Corp. System Research Center, Palo Alto, Calif., Nov. 1989.
5. CARTER, R. L. Electronic posting to the *Numeric Interest* mailing list. June 1991.
6. CLINGER, W. D. How to read floating point numbers accurately. In *Proceedings of the ACM SIGPLAN 90 Conference on Programming Languages and Systems. ACM SIGPLAN Not. 25*, 6 (June 1990), 92–101.
7. CONTE, S. D., AND DE BOOR, C. *Elementary Numerical Analysis: An Algorithmic Approach.* 3rd ed. McGraw-Hill, New York, 1980.
8. FARNUM, C. Compiler support for floating-point computation. *Softw. Pract. Exper. 18*, 7 (July 1988), 701–709.
9. FORSYTHE, G., AND MOLER, C. *Computer Solution of Linear Algebraic Systems.* Prentice-Hall, Englewood Cliffs, N.J., 1967.
10. GOLDBERG, D. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv. 23*, 1 (Mar. 1991), 5–48.
11. GOLUB, G. H., AND VAN LOAN, C. *Matrix Computations.* 2nd ed. Johns Hopkins University Press, Baltimore, Md., 1989.
12. HOARE, C. A. R. An axiomatic basis for computer programming. *Commun. ACM 12*, 10 (Oct. 1969), 576–583.
13. HULL, T. E., AND COHEN, M. S. Toward an ideal computer arithmetic. In *Proceedings of the 8th Symposium on Computer Arithmetic* (Como, Italy, May 19–21). IEEE Computer Society, Los Alamitos, Calif., 1987, pp. 131–138.
14. IEEE. IEEE Standard 754-1985 for binary floating-point arithmetic. *ACM SIGPLAN Not. 22*, 2 (1985), 9–25.
15. NELSON, G., ED. *Systems Programming with Modula-3.* Prentice-Hall, Englewood Cliffs, N.J., 1991.
16. PRIEST, D. M. Algorithms for arbitrary precision floating point arithmetic. In *Proceedings of the 10th Symposium on Computer Arithmetic* (Grenoble, France, June 26–28, 1991). IEEE Computer Society, Los Alamitos, Calif., 1991, pp. 132–143.
17. STERBENZ, P. H. *Floating-Point Computation.* Prentice-Hall, Englewood Cliffs, N.J., 1974.
18. TANG, P. T. P. Table-driven implementation of the exponential function in IEEE floating-point arithmetic. *ACM Trans. Math. Softw. 15*, 2 (June 1989), 144–157.

Received October 1991; revised July 1992; accepted August 1992