DRAFT

Compiler support for floating-point computation

Charles Farnum*
Computer Science Department
University of California, Berkeley
Berkeley, California 94720
farnum@renoir.berkeley.edu

August 21, 1987

1 Summary

Many compilers, even for languages like FORTRAN, make implementation decisions detrimental to high quality floating-point code. Suggestions are given to make it easier for both numerical analysts and naive programmers to obtain better object code with less programmer effort. Special attention is given to supporting the IEEE standard for binary floating-point arithmetic.

2 Introduction

The particular needs of floating-point computation have received relatively little attention in compiler writing research literature, texts, and courses. Language standards often leave the semantics of floating-point source code rather loosely defined, in order to avoid conflicts with the myriad hardware implementations. As a result, many compilers implement floating-point arithmetic in a haphazard fashion, making poor decisions about what machine constructs should result from common source code constructs, what optimizations are (not) worth doing, and sometimes even violating the language standard (and the user's best interests) in a misguided zeal for "better" code.

This paper is a compendium of important floating-point issues for compiler writers. It is organized into the following sections:

1. Notational conventions used in this article.
2. Common illegalities in current compilers.
3. Choices in the precision of subexpression evaluation.
4. Common optimizations that aren't worth doing and uncommon optimizations that are.
5. Support for library writers.
6. Gracefully making the functionality of the IEEE floating-point standards [1] [2] available to the programmer.

*Supported in part by an NSF Graduate Fellowship. Computer support for composing this document was sponsored by the Defense Advanced Research Projects Agency (DoD), Arpa Order No. 4871, monitored by Space and Naval Warfare Systems Command under Contract N00039-84-C-0089.

3 Notational conventions

The following conventions are used in this article:

- In general, computer oriented entities will be referred to with `typewriter text` while mathematical entities will be in *italics*.
- When discussing different precisions, the precision of programming variables will be indicated by the first letter of their name (`s` for single, `d` for double, and `e` for extended).
- A mathematical symbol with the same name as a programming variable denotes the mathematical value represented by the current bit-pattern of the programming value.
- The functions `sngl` and `dble` round their operand to single or double precision, respectively.
- The letter s, d, or e over an arithmetic operator indicates that the result of that operator is rounded to the given precision.

Some examples:

- After executing `s := sngl(d)`, $s = \mathrm{sngl}(d)$.
- We will later argue that `d := d + s*s` should store $d \overset{d}{+} (s \overset{d}{\times} s)$ in `d`.

4 Bugs and Features

Most compilers attempt to implement a superset of the standard language to meet the perceived needs of the intended users. Some extensions are certainly worthwhile (e.g., meeting the IEEE standard often requires some of them), but they should be well documented both externally and by (easily disabled) compile time warning messages. Although this is the party line in compiler texts and published articles, a surprising number of compilers don't obey it. Some common examples to avoid are listed in this section.

4.1 Differing precision in registers and memory

Faulty implementations of common subexpression elimination can lead to trouble if subexpressions are computed to a higher precision than variables (a common tendency in new floating-point chips, and also in certain older machines such as the GE Honeywell models). Consider the following sequence of statements:

```
s := dx/dy;
d := s - dx/dy;
```

The following problems can arise here:

- The compiler may notice, when computing `d`, that the value of `s` is still lying around in a register and can be used without loading. This is a valuable optimization if the value in `s` and the value in the register are, in fact, the same; but if the value in `s` is different (due to rounding from subexpression precision to storage precision), the substitution would be incorrect.
- Common subexpression elimination may notice that `x/y` is used twice, and thus store its value in a temporary location for reuse. If the temporary location has less precision than the register in which the subexpression was evaluated,[1] the wrong value will be used in the computation of `d`.

[1] This is primarily of interest on older machines; newer machines generally allow storage of the widest possible format.

4.2 Unwanted extra precision

Most languages supporting multiple precisions allow implementations to map more than one source language precision to the same machine precision. This can make certain codes malfunction and should only be done if the target machine is limited to one precision. In particular, promoting all variables from single to double is *not* doing the programmer a favor, even if the particular target machine "has plenty of memory". Saving space is not the only reason for using differing precisions; some programs depend on, e.g., `double` being twice the precision of `single`, by making use of the fact that the product of two `single` values can be represented exactly in a `double` variable. Other codes may require a difference to be either zero or large compared with roundoff error; a common way of achieving this is to store a value into a single precision variable and do subsequent computation upon it in double. Although few language standards insist that `double` must have greater precision than `single`, such has been the case on virtually all compilers and machines in the past; the capability should not be removed if the target machine easily supports it.

4.3 Algebraic transformations

Different source languages have different restrictions on when algebraic transformations can be utilized in evaluating expressions. The rules of the language must not be broken unless the transformed code has *exactly* the same execution semantics as the untransformed code on the target machine, including the signals generated. This is often not the case:

- On some machines, such trivial rules as `x*y = y*x` or `1.0*x = x` do not hold.
- Using associativity or distributivity can often change a result by changing the order in which rounding occurs; for example,

  `(10.0E-30 + 10.0E30) - 10.0E30`

  and

  `10.0E-30 + (10.0E30 - 10.0E30)`

  will yield quite different results.
- On all machines, replacing `(x*y) * z` with `x * (y*z)` will change the situations in which overflow occurs, too.

Know what the language allows and what algebraic transformations are valid for the target machine; then act accordingly.

Regardless of what the language allows, the ordering implied by parentheses should never be changed. Carefully written code often requires a specific execution order to ensure that, e.g., overflow does not occur; parentheses are a clear and concise way of showing the programmer's intent.

4.4 Decimal to binary conversion

Good decimal to binary conversion is very difficult; so difficult, in fact, that it is the only function in the IEEE standard where an implementation is allowed to make an error greater than 1/2 an ULP.[2] J. T. Coonen's thesis [3] provides efficient and accurate algorithms for these conversions.

In addition to having a good conversion routine, other more basic requirements must be met. Primary among these is that the same conversion routine be used throughout compilation and execution; some existing systems yield three different values for the same decimal constant, depending on whether the conversion is done by the compiler, the assem-

[2] "Unit in the Last Place" — the difference between two adjacent representable values.

bler, or the run time input libraries. Secondly, it is important to realize that decimal to binary conversion under the IEEE standard is an important operation in its own right, with the possibility of generating signals (such as inexact) and being affected by the current rounding mode. Thus, doing the conversion at compile time is not always a trivial task; in a system fully supporting dynamic rounding modes, two different versions of each inexact constant must be maintained, each with possibly different signals generated on use! It is currently a subject of debate whether this is worthwhile as a default behavior; it is suggested that compilers make it clear what their default behavior is, and allow (perhaps inefficiently) for the alternate behavior.

If the programmer does *not* want any signals generated at run time by the conversion, this can be handled easily in most languages by using the built in facility for defining constants as static objects (e.g., FORTRAN's `DATA` statement or Pascal's `CONST` declarations). Signals generated while converting these constants can be reported appropriately during compilation. The programmer also should be able to specify rounding modes for these static constants.

4.5 Type checking

Many new languages require strong type checking across separately compiled modules, supported by interfaces which provide the necessary type information. Compilers for such languages should enforce the strong type checking by including consistency checks to ensure that the same interfaces were used for separately compiled modules; a depressingly large number of current compilers do not do so, usually with the excuse that "the linker isn't smart enough". This is not a valid excuse; a special linker pre-pass can be written to do the consistency checks, or they can even be trivially implemented by generating code to do the checks at runtime (this only costs a few milliseconds at the beginning of execution).

Unfortunately, most older languages (notably, of course, FORTRAN) provide no means for type checking across separate modules. The difficulties this can cause are by no means limited to numeric code, but a particular instance occurs quite often due to a property of certain floating-point formats on the VAX[3] and IBM/370. On these machines, double precision values can be chopped to single precision values by discarding the lower half of the bit pattern. Thus, if a variable in memory is treated inconsistently as double precision in some places and single precision in others, the program will usually give plausible results. In FORTRAN, all parameters are passed by reference; thus, this will occur any time a double value is passed to a procedure expecting a single value. If such a program is executed on a machine whose formats do not have this property (e.g., any machine conforming to the IEEE binary floating-point standard), the program will malfunction horribly; these bugs are notoriously difficult to find.

Because of the insidious nature of these types of bugs, and their frequent occurrence in numerical code, we strongly recommend that as much type checking be done as possible, regardless of what the language requires/allows. Intentional violations of type checking (e.g., by library writers attempting to implement `COPYSIGN` by bit manipulations) should be forced to be made explicit in the code, either by mechanisms available in the language (e.g., `UNCHECKED_CONVERSION` in Ada[4]) or by extensions defined by the implementor. These force non-portable code to be visibly flagged as such, saving much grief if the software is ported in the future.

[3] VAX is a registered trademark of Digital Equipment Corporation.

[4] Ada is a registered trademark of the Ada Joint Program Office, U.S. Government.

5 Evaluation precision of subexpressions

In languages which support multiple floating-point precisions, the arithmetic functions (+, ×, cos, etc.) are typically overloaded, e.g., the symbol + might represent any of the functions $\overset{s}{+}$, $\overset{d}{+}$, or $\overset{e}{+}$ depending on the context.[5] Deciding which function to use is often partially left to the implementor; the choice should be considered carefully. Below we discuss some alternatives. A FORTRAN implementation using the strategy we recommend is described in a paper by Corbett [4].

5.1 Strict evaluation

Most languages require that the precision used to evaluate subexpressions with an overloaded operator be at least as wide as the widest operand. Strict evaluation uses the narrowest precision allowed by the language. For example, the statement `d := d + s*s` would store $d \overset{d}{+} (s \overset{s}{\times} s)$ in `d`. This strategy is easy to implement and the most efficient on machines such as the VAX whose machine instructions always yield results of the same precision as the operands.

Unfortunately, strict evaluation is almost never what the programmer desires when precisions are mixed within an expression. In the previous example, the extra precision in the variable `d` is entirely wasted, as it is swamped by the rounding error introduced by rounding $s \times s$ to single precision. Another example is the evaluation of `d := 7.0/3.0 * d`; if `7.0/3.0` is evaluated to single precision, the extra precision of `d` will be destroyed.

A common counter to these arguments is that a "careful programmer" (and, by implication, anyone worth supporting) will write

```
d := d + dble(s) * dble(s)
```

to ensure the necessary precision is carried. This certainly makes the code more portable, but at the expense of programmer effort and program legibility. The intent of overloaded operators is to eliminate clutter; the strict evaluation strategy forces such clutter back in. We counter the portability argument with the following observations:

- Given a compiler that makes well documented decisions on unspecified portions of the language, it is a worthwhile and relatively inexpensive task to write a source to source translator that removes most of the implementation specific code. For example, this tool could easily replace `d := d + s*s` with `d := d + dble(s)*dble(s)`.

- Using strict evaluation doesn't eliminate the portability problem, since some compilers use other strategies. Thus the same argument justifies insisting that "careful programmers should write `d := d + sngl(s*s)`"; this implication is usually ignored.

Don't use strict evaluation.

5.2 Widest available

An easily implemented alternative to strict evaluation uses the widest precision supported by hardware as the result precision of all overloaded operators. In a machine where the widest format supported is also the fastest (e.g., many of the new microprocessor floating-point chips) this strategy yields both the fastest speed and the most accurate results.

[5] Decimal to binary conversion can also be considered as an overloaded operator in that the precision of the constant should ideally depend on the context where it is used. Writing `0.3D0` instead of `0.3` is both tedious and easy to overlook; it has the further disadvantage that a program can not be upgraded from single to double precision simply by changing the variable declarations.

Although often ideal, using the widest available format is troublesome in the following cases:

- On most machines, higher precisions imply longer execution times. This extra cost comes both from the need to manipulate extra bits during the computation and from the need to convert values from one precision to another.

  Whether or not this execution speed cost is worth the benefit due to less rounding error depends significantly on the program code. Three simple examples:

  1. `d := d + s*s`
     Here, the higher cost of evaluating $d \overset{d}{+} (s \overset{d}{\times} s)$ instead of $d \overset{d}{+} (s \overset{s}{\times} s)$ is certainly worthwhile; otherwise, the extra precision of d is wasted.

  2. `s := s1 + s2 * (s3 + s4/s5)`
     Although

     $$\mathrm{sngl}(s_1 \overset{d}{+} s_2 \overset{d}{\times} (s_3 \overset{d}{+} s_4 \overset{d}{/} s_5))$$

     may be a marginally better value than

     $$s_1 \overset{s}{+} s_2 \overset{s}{\times} (s_3 \overset{s}{+} s_4 \overset{s}{/} s_5)$$

     it would be hard to justify the added cost.

     If the cost *is* justified, then the programmer should write

     ```
     d := s1 + s2 * (s3 + s4/s5)
     ```

     in order to take advantage of the extra precision. The widest needed strategy, which we suggest below, uses single if the value is to be stored in s and double if it is to be stored in d.

  3. `s := s1 + s2`
     On a reasonable machine,

     $$\mathrm{sngl}(s_1 \overset{d}{+} s_2) = s_1 \overset{s}{+} s_2;$$

     here the extra expense is simply wasted.

  Since most statements in typical code are of the simple form given in the third example, the use of the widest available precision is often wasted; if the widest precision is computationally expensive, then this strategy loses much of its appeal.

- In a system which supports infinitely many precisions, the "widest available" format is non-existent. Such systems which choose to support overloaded operators should use the "widest needed" strategy outlined below.

5.3 Widest needed

Strict evaluation eliminates much of the usefulness of overloaded operators by forcing the programmer to use explicit type conversions whenever a benefit is to be gained from varying precisions. Using the widest available precision wastes the extra computation in frequently arising situations. A natural suggestion is to use the widest precision that is "needed", i.e., that will be used in the local context.

More precisely, assigning precisions to an expression tree using the widest needed strategy can be described as follows:

1. Assign tentative precisions using the strict evaluation strategy, in a bottom up traversal of the tree.

2. Using a top down traversal of the tree, check each overloaded operator. Let the tentative precision of the operator be p_t,

and the precision expected by the parent[6] be p_e. Assign the wider of p_t and p_e as the precision of the operator.

This strategy is far superior to strict evaluation; if it is more efficient than the widest available strategy for a particular target machine, then it should be the default. The effects of the other strategies can be obtained by using explicit coercion functions in those uncommon cases where they provide better results. Implementing the widest needed strategy is more difficult than the other strategies, but not terribly so; a modification to a UNIX[7] FORTRAN compiler to implement the widest needed strategy required about 80 man hours [4] in the context of a compiler that was doing optimizations such as inline expansions. When this strategy is designed into a compiler instead of added after the fact, the time needed should decrease.

6 Improving efficiency

The ideal goal of an optimizing compiler is to increase the efficiency of a program without changing the output it produces. Unfortunately, the fact that floating-point arithmetic is normally viewed as an approximation to real or complex arithmetic has led some compiler writers to sacrifice semantics for the sake of speed because "one approximation is as good as another". is simply not so; it is possible to provide tight bounds on the error in many computations by careful coding that takes into account the differences between floating-point and real arithmetic. Therefore *do what the programmer says, not what you think he "wants".*

We believe this rule to be so important that we begin this section with two optimizations that are legal under most language standards, but can be painful for the numerical analyst:

- Reorganizing expressions to use fewer registers is a well understood and common technique. We have already pointed out that doing so can introduce anomalies since floating-point arithmetic fails to honor some common identities. Many languages allow arbitrary application of algebraic identities, regardless of their applicability to the machine arithmetic, so these optimizations do not fall under the heading of "avoid at all costs"; but parentheses, at the very least, should be respected. Experienced coders will not introduce parentheses without a good reason; inexperienced coders will not run their programs often enough for such optimizations to be worth the effort.

- Many optimization techniques involve moving computations, e.g., moving code out of loops, storing common subexpressions, or evaluating constant arithmetic at compile time. Floating-point arithmetic can often have side effects aside from computing a result, e.g., setting flags and/or trapping.[8] If code is to be moved, it is important to ensure both that no spurious side effects are introduced at the new location, and that the correct side effects occur at the old location.

[6] The result of the operator will be either used in a context expecting a fixed precision (e.g., assignment to a variable or as a parameter) or as an operand of an overloaded operator. In the former case, the expected precision is defined by the language (although discovering the expected precision may require more cross-file type checking than is required/facilitated by the language definition); in the latter, it is the precision assigned to the parent operator.

[7] UNIX is a registered trademark of Bell Laboratories.

[8] Remember that the decimal to binary conversion implicit in the appearance of a numeric literal is a full-fledged operation under the IEEE standard, and can generate signals and/or be affected by rounding modes.

We now list optimizations that are particularly important in floating-point code.

- A simple local improvement removes unnecessary coercions. If a cautious programmer has written

```
d := d + dble(s)*dble(s)
```

and the target machine supports a double precision product of two single precision numbers, don't penalize the programmer with the extra coercions. The FORTRAN standard considers this functionality important enough to provide a special function (DPROD) so that the programmer can perform this optimization at the source level; such a simple peephole optimization should be handled automatically by the compiler.

- Careful instruction scheduling to minimize pipeline conflicts is of particular importance in the longer pipelines typically used by high speed floating-point units.

- Machines with vector instructions cater to numeric processing and any compiler which fails to vectorize loops will fall into disuse. Unfortunately, many compilers refuse to vectorize any loop with an embedded IF statement. Many vector machines have a select vector operation which chooses one of two possible results based on a boolean condition; this operation is tailored to conditional expressions, but in a language without conditional expressions an IF statement must be used. This happens quite often when using functions with removable singularities; consider, e.g., evaluating $(\sin x)/x$ in a loop:

```
if (x[i] = 0) then
    y := 1
else
    y := sin(x[i]) / x[i]
endif
```

If it is considered too difficult to vectorize the above, then an extension to the language adding a conditional expression should be strongly considered.[9]

- Branch prediction is another area where great improvement can be gained. Much of the branching in floating-point codes exists solely for the purpose of handling exceptional cases; the jump only needs to be taken a very small percentage of the time. This situation is ideal for branch prediction optimizations. Profiling information appears to be the best way to decide which branch is more likely, but a simpler implementation can provide a notation for the programmer to make the prediction.

- Division is almost always much slower than multiplication. Replacing division with multiplication is done by many programmers as a matter of habit; it is certainly within the compiler's jurisdiction to replace division by a constant with multiplication provided the reciprocal is exact and the multiplication has identical effects.

[9] It is common practice in some circles to create conditional expressions, in languages that allow coercion of booleans to integers, by exploiting the fact that TRUE is often represented by 1 and FALSE by 0; the above example would be coded as `y := 1.0*(x[i] = 0) + sin(x[i])/x[i]*not(x[i] = 0)`. There exist compilers that recognize this convention as a special case and will skip over the divide by zero when x[i] is zero. Such programming practices should be discouraged, but implementors with strong commitments to portability should interpret this construct as a conditional expression and provide suitable warnings when booleans are coerced to real values.

7 Support for library writers

A good floating-point library for FORTRAN or an IEEE standard implementation is a job for specialists. Implementing a good library requires many capabilities that are not available in FORTRAN, nor in many more modern languages; e.g., a library that imposes the standard function call overhead for the use of a short generic function like ABS is not satisfactory. The necessary features are listed below, along with a specific example of how they might be used in implementing complex arithmetic.

7.1 Operator overloading

Adding a new data type, such as double precision complex, can be practically unbearable in a language like FORTRAN without compiler support. No one wants to go through the bother of writing

```
DOUBLE A(2),B(2),C(2),TEMP(2)
DCTIMES(TEMP,B,C)
DCPLUS(A,A,TEMP)
```

instead of the much clearer

```
COMPLEX_DOUBLE A,B,C
A = A + B * C .
```

The arithmetic operators in most languages are already overloaded with regard to integer and floating-point types, as well as differing precisions; the clear solution is to make this facility available to the library writer and, if feasible, to the general programmer. Examples of such facilities can be found in the operator functions of C++ or Ada. FORTRAN, of course, also needs the capability of defining new types.

Ideally, the implementation of precision overloading discussed earlier in the context of subexpression precisions should be orthogonal to the data type overloading; e.g., `d := d + s*s` should evaluate `s*s` to the same precision, regardless of whether `s` is real or complex.

7.2 Catching fatal errors

In some systems, the default behavior for a function receiving strange arguments is to halt the program and write an error diagnostic. If poorly implemented, this can lead to bizarre behavior when library functions call each other: consider the poor scientific programmer who is using only sines and cosines and receives the message "Invalid argument for REM". Some users of floating-point code have, quite legitimately, no idea what REM is.

The best solution here is to allow the caller some control over whether a function will terminate execution; the function which called REM could, if informed of the error, produce a more suitable message or perhaps use a different method of finding the desired result. If the implementation of this more extensive handling is considered too expensive, the terminating message should at the very least provide a traceback of the library calls that resulted in the error, e.g., "Invalid argument for REM; REM called by COS called by ...".

7.3 Inline functions

Replacing a function call with the body of the function is a rather simple transformation that often results in much more efficient code. Deciding when the transformation will be worthwhile can be a difficult task, but it is easy to give the programmer the ability to specify that a given function should be compiled inline.

We mention inlining here because the functions introduced by overloaded operators are often so short that it is clear they should always be expanded inline; thus, the library im-

plementor, at the very least, should have this capability.

7.4 Specifying structured constants

Given a new data type such as `complex` or `interval`, it is of course useful to provide some syntax for specifying constants of that type. The simplest alternative is to have certain function calls that can build constants from their primitive parts, e.g. a function `complex(r,i)` which returns the complex number (r,i). If the function call can be specified as inline, this facility provides the necessary capability without the need for special syntax.

7.5 Specifics for complex arithmetic

We now outline how the above functionalities can be combined to implement complex arithmetic. Adding interval arithmetic is similar.

A good library for complex arithmetic should provide:

1. The ability to declare complex variables, with the same choice of precisions enjoyed by real variables. The record facility provided by most languages is perfectly suitable; if the language has no records, they can be added with appropriate source to source translations.

2. Standard arithmetic functions for assorted mixtures of complex and real arguments, overloaded on the appropriate symbols. Not all combinations of real and complex values should be handled by promoting the real to a complex. For example, for `real r` and `complex c`, `r+c` should not be evaluated as `complex(r)+c`; doing so is less efficient and may be less accurate.[10] Thus, the overloading facility must be able to handle operands of different base types as well as different precisions. Some of these functions should be declared inline, depending on their simplicity and the relative speed of procedure calls on the target machine.

3. A function `complex(r,i)` which returns the value (r,i); this function can be used to designate complex constants, and should be declared inline.

4. Ideally, an `imaginary` date type for representing values whose real component is zero (this type is analogous to the type `real`) and a function for converting reals to imaginaries. The `imaginary` type would not normally be used to declare variables, but would rather be the result of expressions of the form `iota*r`, where `iota` is the constant $\imath = \sqrt{-1}$. The new type is simply the method by which this functionality can be added to the language without making explicit changes in the compiler.

8 Merging language and IEEE floating-point standards

If the IEEE floating-point standard has been implemented in a given system, then compilers for the system should naturally make the facilities of the standard available to high level language programmers. This section describes

[10] On an IEEE Standard machine, evaluating `(x,y) + z` as `(x,y) + (z,+0)` will destroy the sign of y if y is zero. The sign of zero can be of great importance in complex arithmetic; this is not the place to fiddle with it.

how the language standard and the IEEE standard can be merged in the least painful way.

Suggestions in this section referring to handling of NaNs apply equally well to systems that support similar entities (e.g., the indefinite operand of a CDC machine, or the reserved operands of a VAX).

8.1 Precisions

A compiler should provide source language data types for the precisions available on the target machine. Mapping source language types to the IEEE standard single, double, and extended should be done with the knowledge that single is the normal type used for input and output, while double and extended are normally used only to provide extra precision for sensitive calculations; thus the standard floating-point type in the language (e.g., `real` in Pascal or `FLOAT` in Ada) should map onto single. If a language lacks standard names for various precisions, `single`, `double`, and `extended` should be used for the sake of program portability.

8.2 Comparisons

Floating-point comparisons have typically been modeled by the function `compare(x,y)` which returned one of three values `<`, `>`, or `=`. The IEEE standard provides for a fourth value, `?` or *unordered*, which is the result if `x` or `y` is a NaN. This must be handled carefully:

- The comparisons $<$ and $\geq$ are *not* opposites; if `x` is a NaN, then both `x < y` and `x >= y` are false. Comparisons are often reversed during code generation; if this is done carelessly (perhaps at the source level or in an intermediate representation)and `x >= y` is substituted for `not(x < y)`, then improper code will result. This is only a problem in the sense that it provides an easy excuse for making a mistake; it should be easy to correctly negate comparisons on a machine which supports the standard.
- Many languages use the symbol `<>` to represent $\neq$. This can create confusion: `<>` is often read "less than or greater than", but in the standard, $((x < y) \vee (x > y))$ is not the same as $(x \neq y)$ when x or y is a NaN. Thus `<>` has two possible meanings. The choice should be left as a compiler option, i.e., the programmer should have the ability to specify that `<>` will mean either "less than or greater than" or "not equal", to help portability of existing code. The default meaning should be "less than or greater than" for two reasons:
  1. The comparison $x \neq y$ can be easily written `not(x=y)`, but
     $$(x < y) \vee (x > y)$$
     has no equally short form other than `x <> y`.
  2. The default is safe in the sense that users who use `x <> y` expecting it to be interpreted $x \neq y$ will be signaled with an invalid operation whenever it makes a difference (i.e., when x or y is a NaN).

A function `compare(x,y)` returning one of four values `<`, `>`, `=`, or `?` (without generating signals for NaNs) should be explicitly provided, particularly in languages with case or switch statements that can make particularly efficient use of such a function.

8.3 Minor incompatibilities in miscellaneous functions

The papers describing the IEEE standards [1] [2] contain many functions, some as part of

the standard and some only recommended, which are slightly different from assorted language standards. These incompatibilities vary in their severity, and should be handled in different ways:

1. Some difficulties arise because the IEEE standard introduces values that were unanticipated by the language standard. When the incompatibility only affects values outside the domain of the language standard, then the IEEE standard should be allowed to take precedence.

2. Some functionalities are provided slightly differently in the IEEE and language standards; e.g., most languages define a function `MOD(y,x)` such that

   $$0 \leq \mathrm{mod}(y,x)/x < 1,$$

   while the standard defines a similar function `REM` such that

   $$|\mathrm{rem}(y,x)| \leq |x/2|.$$

   In these cases, both the language and the IEEE standard functions should be supported. The library writing support outlined in the previous section is of great value here in moving the burden of supporting a plethora of slightly different functionalities from the compiler writer to a numerical library specialist.

3. Occasionally the language and IEEE standard stand in direct opposition. A compiler switch should be available to choose which standard will be obeyed at these points.

Some of the common differences are listed here, along with suggestions on how they should be handled.

1. `-x` is often defined to be equivalent to `0-x`, but this is inappropriate when signed zeros and signaling NaNs exist; `-x` should be the value of `x` with the sign bit reversed. The language standard can simply be overruled here.[11]

2. FORTRAN's `SIGN` function is identical to `COPYSIGN` except when the second argument is zero;

   ```
   SIGN(1.0,-0.0) = +1.0
   ```

   but

   ```
   COPYSIGN(1.0,-0.0) = -1.0 .
   ```

   Both functions should be supported.

3. Many languages specify that, when rounding x to an integer, $\lceil x \rceil$ is chosen if x is halfway between $\lfloor x \rfloor$ and $\lceil x \rceil$. In the IEEE standard, ties are resolved in favor of the even neighbor. Both functionalities should be supported.

4. In APL, `0/0` is defined to be 1, while the IEEE standard default value is a NaN. An APL compiler should have a switch to determine which value results.

5. Some languages define `I = X` to be equivalent to `I = INT(X)` for integer `I` and real `X`. In such a language, a compiler switch should allow for the IEEE option that the first may signal inexact while the latter does not.

6. A constant such as `MAXREAL` is often defined to be the largest real number representable by the floating-point system. This should not be set to infinity, but to the largest finite value; codes that use it usually treat it as a finite value whenever it might make a difference. The source level constant `INFINITY` should be available for indicating infinities.

[11] Unless it also deals with signed zeros, in which case both standards should be supported via a compiler switch.

7. There is often difficulty deciding what a function should do with a NaN. We will give a single example, the `MAX` function defined in many languages to return the maximum of two values. What should `MAX(5,NaN)` be? The answer depends on the application; sometimes the NaN should be ignored and 5 returned, while in other cases the NaN should propagate. Both functions should be supported.[12]

9 Acknowledgements

This paper is based on lectures given by W. Kahan at the University of California, Berkeley during the Fall of 1986; Professor Kahan has been of great help in revising the paper. Richard James' notes for the IEEE P854 working group [5] were useful as a numerical analyst's wish list of what compiler writers ought to know. David Hough provided helpful comments on an earlier version of the paper.

References

[1] IEEE standard for binary floating-point arithmetic. Reprinted in *SIGPLAN*, 22(2):9–25, February 1987, 1985.

[2] W. G. Cody et al. A proposed radix- and word-length-independent standard for floating-point arithmetic. *IEEE Micro*, 4(4):86–100, August 1984.

[3] Jerome T. Coonen. *Contributions to a Proposed Standard for Binary Floating-Point Arithmetic.* PhD thesis, University of California, Berkeley, 1984.

[4] Robert P. Corbett. Enhanced arithmetic for Fortran. *SIGPLAN*, 17(12):41–48, December 1982.

[5] Richard E. James. Dear compiler writer. July 1985. A list of compiler-related issues discussed by the P854 Working Group.

[12] Names for all of these functions are certainly a problem; work is being done trying to arrive at some naming conventions. In the meantime, the decision can be left to the library writer.