Real Inferno

Eric Grosse
Bell Laboratories
Murray Hill NJ 07974 USA
ehg@bell-labs.com

19 August 1996

Abstract

Inferno is an operating system well suited to applications that need to be portable, graphical, and networked. This paper describes the fundamental floating point facilities of the system, including: tight rules on expression evaluation, binary/decimal conversion, exceptions and rounding, and the elementary function library.

Although the focus of Inferno is interactive media, its portability across hardware and operating platforms, its relative simplicity, and its strength in distributed computing make it attractive for advanced scientific computing as well. Since the appearance of a new operating system is a relatively uncommon event, this is a special opportunity for numerical analysts to voice their opinion about what fundamental facilities they need. The purpose of this short paper is to describe numerical aspects of the initial release of Inferno, and to invite comment before the tyranny of backward compatibility makes changes impossible.

Overviews can be found at http://inferno.bell-labs.com/inferno/, but for our immediate purposes it may suffice to say that Inferno plays the role of a traditional operating system (with compilers, process control, networking, graphics, and so on) but can run either on bare hardware or on top of another operating system like Windows95 or Unix. Programs for *Inferno* are written in the language *Limbo* and compiled to machine-independent object files for the *Dis* virtual machine, which is then implemented with runtime compilation for best performance. Files are accessible over networks using the *Styx* protocol; together with the presentation of most system resources as files and the manipulation of file namespaces, this permits integration of a collection of machines into a team. Limbo looks somewhat like a mixture of C and Pascal, augmented by modules (to cope with the namespace and dynamic loading needs of large programs) and by a channel facility for convenient (coarse-grain) parallel programing. Array references are bounds-checked and memory is garbage collected.

The rest of this paper covers the fundamental floating point environment provided by the Limbo compiler and *math* module, the "elementary functions," and finally some comments on why particular definitions were chosen or why certain facilities were included or excluded. This discussion assumes the reader is familiar with scientific computing in general and the IEEE floating point standard in particular.

1 Floating point

In Limbo, arithmetic on literal and named constants is evaluated at compile time with all exceptions ignored. Arithmetic on variables is left by the compiler to runtime, even if data path analysis shows the value to be a compile time constant. This implies that tools generating Limbo source must do their own simplification, and not expect the compiler to change x/x into 1, or $-(y-x)$ into $x-y$, or even $x-0$ into x. Negation $-x$ changes the sign of x; note that this not the same as $0-x$ if $x=0$.

The compiler may perform subexpression elimination and other forms of code motion, but not across calls to the mode and status functions. It respects parentheses. The evaluation order of $a+b+c$ follows the parse tree and is therefore the same as for $(a+b)+c$. These are the same rules as for Fortran and C.

Contracted multiply-add instructions (with a single rounding) are not generated by the compiler, though they may be used in the native BLAS libraries. All arithmetic follows the IEEE floating point standard [6], except that denormalized numbers may not be supported; see the discussion in section 3.

The most important numerical development at the language level recently has been accurate binary/decimal conversion [1][4][12]. Thus printing a real using *%g* and reading it on a different machine guarantees recovering identical bits. (Limbo uses the familiar *printf* syntax of C, but checks argument types against the format string at compile time, in keeping with its attempt to help the programmer by stringent type checking.) A good *strtod/dtoa* is, unfortunately, 1700 lines of source (15kB compiled), though with modest average runtime penalty. This code must be used in the compiler so that coefficients are accurately transferred to bytecodes. Smaller, faster, but sloppier, runtimes will also be provided when mandated by limited memory and specialized use. However, programmers may assume the features described in this paper are present in all Inferno systems intended for general computing.

Each thread has a floating point control word (governing rounding mode and whether a particular floating point exception causes a trap) and a floating point status word (storing accumulated exception flags). Functions *FPcontrol* and *FPstatus* copy bits to the control or status word, in positions specified by a mask, returning previous values of the bits. *getFPcontrol* and *getFPstatus* return the words unchanged.

The constants *INVAL, ZDIV, OVFL, UNFL, INEX* are non-overlapping single-bit masks used to compose arguments or return values. They stand for the five IEEE exceptions:

- "invalid operation" ($0/0$,$0+NaN$,$\infty-\infty$,$\sqrt{-1}$)

- "division by zero" $(1/0)$,
- "overflow" $(1.8e308)$
- "underflow" $(1.1e-308)$
- "inexact" $(.3*.3)$.

The constants *RND_NR, RND_NINF, RND_PINF, RND_Z* are distinct bit patterns for "round to nearest even", "round toward $-\infty$", "round toward $+\infty$", "round toward 0", any of which can be set or extracted from the floating point control word using *RND_MASK.* For example,

- to arrange for the program to tolerate underflow, *FPcontrol(0,UNFL).*
- to check and clear the inexact flag, *FPstatus(0,INEX).*
- to set directed rounding, *FPcontrol(RND_PINF,RND_MASK).*

By default, *INEX* is quiet and *OVFL, UNFL, ZDIV,* and *INVAL* are fatal. By default, rounding is to nearest even, and library functions are entitled to assume this. Functions that wish to use quiet overflow, underflow, or zero-divide should either set and restore the control register themselves or clearly document that the caller must do so. The "default" mentioned here is what a Limbo program gets if started in a fresh environment. Threads inherit floating point control and status from their parent at the time of spawning and therefore one can spawn a "round toward 0" shell and re-run a program to effortlessly look for rounding instabilities in a program.

2 Elementary functions

The constants *Infinity, NaN, MachEps, Pi, Degree* are defined. Since Inferno has thorough support of Unicode, it was tempting to name these ∞, ε, π, and ^{o}, but people (or rather, their favorite text editing tools) may not be ready yet for non-ASCII source text. *Infinity* and *NaN* are the positive infinity and quiet not-a-number of the IEEE standard, double precision. *MachEps* is 2^{-52}, the unit in the last place of the mantissa 1.0. The value of *Pi* is the nearest machine number to the mathematical value π. *Degree* is $Pi/180$.

Three useful functions *fdim, fmax, fmin* are adopted from the Numerical C extensions [13]. The unusual one of these, often denoted $(x-y)_+$, is defined by $\mathrm{fdim}(x,y) = x-y$ if $x>y$, else 0. The compiler may turn these into efficient machine instruction sequences, possibly even branch-free, rather than function calls. There are two almost redundant mod functions: *remainder(x,y)* is as defined by the IEEE standard (with result $|r| \le y/2$); *fmod(x,y)* is $x \bmod y$, computed in exact arithmetic with $0 \le r < y$. Limbo has a "tuple" type, which is the natural return value in the call $(i,f) = \mathrm{modf}(x)$ to break x into integer and fractional parts. The function *rint* rounds to an integer, following the rounding mode specified in the floating point control word.

For a good-quality, freely-available elementary function library, *math* uses the IEEE subset of *fdlibm* [11]. Of course, a conforming implementation may use entirely different source, but must take care with accuracy and with special arguments. There are the customary power, trigonometric, Bessel, and erf functions, and specialized versions $\mathrm{expm1}(x) = e^x - 1$, $\mathrm{log1p}(x) = \log(1 + x)$. An additional function $\mathrm{pow10}(n) = 10^n$ is defined; in the default implementation this is just fdlibm's $\mathrm{pow}(10., n)$ but it is provided so that separate trade-offs of accuracy and simplicity can be made [10]. *fdlibm* uses extra precise argument reduction, so the computed $\sin(n*Pi)$ is small but nonzero. If demands warrant, degree versions of the trigonometric functions will be added, but for now the style $\sin(45 * Degree)$ is used. The library also provides IEEE functions *ilogb, scalbn, copysign, finite, isnan,* and *nextafter*.

The functions *dot, norm1, norm2, iamax, gemm* are adopted from the BLAS [3] to get tuned linear algebra kernels for each architecture, possibly using extra-precise accumulators. These are defined by $\Sigma x_i y_i$, $\Sigma|x_i|$, $\sqrt{\Sigma x_i^2}$, i such that $|x_i| = \max$, and $C = \alpha AB + \beta C$ with optional transposes on A and B. Since Limbo has only one floating-point type, there is no need here for a precision prefix. Limbo array slices permit the calling sequences to be more readable than in Fortran77 or C, though restricted to unit stride. This encourages better cache performance anyway. The matrix multiply function *gemm* remains general stride (and is the foundation for other operations [7]).

Limbo is like C in providing singly-subscripted arrays with indexing starting at 0. Although Limbo offers arrays of arrays, as in C, for scientific work a better choice is to adopt the style of linearizing subscripts using Fortran storage order. This promotes easier exchange of data with other applications and reuses effort in organizing loops to achieve good locality. In previous language work [5], we implemented a C preprocessor that allowed the programmer to choose a convenient origin (such as 1) and have it compiled into 0 for the base language; because we passed arrays as dope vectors, we were even able to allow different origins for the same array in calling and called functions. The main lesson we learned from that experience, however, was that permutations become a nightmare when there is anything but dogmatic adherence to a single origin. So for an m by n matrix A, the programmer should use loops with $0 \le i < m$ and $0 \le j < n$ and access $A[i + m * j]$.

For interoperability with foreign file formats and for saving main memory in selected applications, functions are provided for copying bits between and reals and 32-bit or 64-bit IEEE-format values.

Finally, *math* provides a tuned quicksort function *sort(x,p)* where x is a real array and p is an int array representing a 0-origin permutation. This function leaves the contents of x untouched and rearranges p so that $x[p_i] \le x[p_{i+1}]$. This is usually what one wants to do: sort an array of abstract data types based on some key, but without the need to actually swap large chunks of memory.

3 Rationale

This section discusses why certain numerical features were included or not.

3.1 Rounding modes and accumulated exceptions

Directed rounding is only needed in a very few places in scientific computing, but in those places it is indispensable. Accumulated floating point exceptions are even more useful. User trap handling is a harder problem, and may be worth leaving for later, possibly with a default "retrospective diagnostics" log [8].

Note that the exception masks must be architecture independent, since they reside in the Limbo bytecodes, and therefore the implementation involves a small amount of bit fiddling. Still, it is efficient enough to encourage use. Ports to the Alpha, ARM, and Hal processors will be a severe challenge, since their architects chose to put rounding modes statically in instruction opcodes rather than providing the dynamic model specified in section 2 of the IEEE standard. Perhaps on these machines, rounding modes will simply be ignored.

3.2 Sudden underflow

Some processor vendors make supporting gradual underflow just too hard. (One must struggle upon the system trap to reconstruct exactly which instruction was executing and what the state of the registers was. On the MIPS, it is said to be 30 pages of assembler.) So Inferno supports denormalized numbers only if the hardware makes this easy. Providing underflow that is correct but very slow, as some systems do, is not necessarily doing the user a favor.

To determine portably if a particular system offers gradual underflow, mask off UNFL and do trial arithmetic.

3.3 Speed

Computers with slow (software) gradual underflow usually provide a fast flush-to-0 alternative. This often suffices, though there are important examples where it forces an uglier and slower coding style. A worse situation is if the hardware uses system traps for Infinity and NaN arithmetic. The resulting slowdown will make otherwise excellent and natural algorithms run slowly [2].

We considered providing syntax to declare a certain program scope within which precise IEEE behavior was required, and relaxing the rules outside such scopes. (The numerical C extensions [13] use pragma for this purpose.) These scope declarations would need to be in the bytecodes, since significant optimization may be attempted by the runtime compiler. After some discussion, and with some trepidation, it was agreed that instead all compilers would be required to preserve the same result and status as for an unoptimized version.

3.4 Comparison

The standard C operators == != < <= > >= are the only comparisons provided, and they behave exactly like the "math" part of Table 4 of the IEEE standard. Programs interested in handling NaN data should test explicitly. This seems to be the way most people program and leads to code more understandable to nonexperts. It is true that with more operators one can correctly write code that propagates NaNs to a successful conclusion—but that support has been left for later. NaN("tag") should be added at that same time.

3.5 Precision

All implementations run exclusively in IEEE double precision. If the hardware has extra-precise accumulators, the round-to-double mode is set automatically and not changeable, in keeping with Limbo's design to have only one floating point type. Extended precision hardware, if available, may be used by the built-in elementary function and BLAS libraries. Also, we contemplate adding a dotsharp function that would use a very long accumulator for very precise inner products, independent of the order of vector elements[9]. But reference implementations that use only double precision, avoid contracted multiply-add, and evaluate in the order 1 up to n will always be available for strict portability.

At the time the decision was made to restrict the system to 64-bit floating point, Limbo integers were almost exclusively 32-bit and the consistency argument to have a single real type was compelling. Now that Limbo has more integer types the decision might be reconsidered. But so many engineers needlessly struggle with programs run in short precision, that offering it may do as much harm as good. On most modern computers used for general purpose scientific computing, 64-bit floating point arithmetic is as fast as 32-bit, except for the memory traffic. In cases where the shorter precision would suffice and memory is a crucial concern, the programmer should consider carefully scaled fixed point or specialized compression. To efficiently interoperate with data files that use the short format, programmers may use the provided realbits32 function. While there are surely appropriate uses for a first-class 32-bit real type, for now we follow Kahan's sarcastic motto "why use lead when gold will do?"

3.6 BLAS

The few BLAS in the core library were chosen for readability and, in case of gemm, for optimization beyond what a reasonable compiler would attempt. We expect that compilers will (soon) be good enough that the difference between compiling $y+ = a * x$ and calling daxpy is small. Also, as mentioned above, dot and gemm might reasonably use combined multiply-add or a long accumulator in some optional implementations.

3.7 $\Gamma(x)$

To avoid confusion with the C math library, which defined *gamma* as $\ln \Gamma$, we offer only *lgamma* for now. This function and *modf* return an (int,real) tuple rather than assigning through an integer pointer, in keeping with Limbo's design. The opportunity has been taken to drop some obsolete functions like *frexp*. Other functions are unchanged from the C math library.

3.8 Future

A prototype preprocessor has been written to allow the scientific programmer to write $A[i, j]$ for an A that was created as a $Matrix(m, n)$ and to have the subscript linearization done automatically. Here $Matrix$ is an Limbo abstract data type containing a real array and integers m, n, and column stride lda used as in typical Fortran calling sequences.

The Limbo compiler is soon expected to implement the type *complex*.

Higher level numerical libraries will also be provided, and although that topic is beyond the scope of this paper, opinions about what should come first would be welcome.

Distributed computing has not been mentioned here because it involves relatively few considerations specific to floating point computation. However, it may be worth noting that in the default environment (with underflow trapped, so that presence or absence of denormalized numbers is not significant) programs run independently on heterogeneous machines nevertheless get precisely identical results, even with respect to thread scheduling. This implies that certain communication steps can be avoided, and that regression testing is considerably simplified.

Please direct comments on these numerical aspects of Inferno to Eric Grosse. More general technical comments can be directed to the principal developers of Inferno: Sean Dorward, Rob Pike, Dave Presotto, Howard Trickey, and Phil Winterbottom. I am grateful to David Gay, Bell Labs, to David Hook, University of Melbourne, and to participants of the IFIP WG2.5 Working Conference on Quality of Numerical Software for insightful comments on a first draft of this paper. Inferno, Limbo, and Dis are trademarks of Lucent Technologies Inc. Unix is a trademark of Unix Systems Laboratories. Windows95 is a trademark of Microsoft.

References

[1] W. D. Clinger. How to read floating point numbers accurately. In *Procedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 92–101, 1990.

[2] James W. Demmel and Xiaoye Li. Faster numerical algorithms via exception handling. In Jr. Earl Swartzlander, Mary Jane Irwin, and Graham

Jullien, editors, *Proceedings: 11th Symposium on Computer Arithmetic*. IEEE Computer Society Press, 1993.

[3] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. Algorithm 656: An extended set of Basic Linear Algebra Subprograms. *ACM Trans. on Mathematical Software*, 14(1):18–32, March 1988.

[4] D. M. Gay. Correctly rounded binary-decimal and decimal-binary conversions. Numerical Analysis Manuscript No. 90-10, AT&T Bell Laboratories, Murray Hill, NJ, 1990. freely redistributable, available at `http://netlib.bell-labs.com/netlib/fp/`.

[5] E. H. Grosse and W. M. Coughran, Jr. The pine programming language. Numerical Analysis Manuscript 83-4, ATT Bell Laboratories, 1983. `ftp://cm.bell-labs.com/cm/cs/doc/92/pine.ps.Z`.

[6] IEEE. Standard for binary floating-point arithmetic. Technical Report Std 754-1985, ANSI, 1985.

[7] Bo Kagstrom, Per Ling, and Charles Van Loan. Portable high performance GEMM-based Level 3 BLAS. In R. F. Sincovec et al., editor, *Parallel Processing for Scientific Computing*, pages 339–346. SIAM Publications, 1993. `/netlib/blas/`.

[8] W. Kahan. Lecture notes on the status of IEEE Standard 754 for binary floating-point arithmetic. Technical report, Univ. Calif. Berkeley, May 23 1995. Work in Progress.

[9] U. Kulisch and W.L. Miranker. *Computer arithmetic in theory and practice.* Academic Press, 1980.

[10] M. D. McIlroy. Mass produced software components. In Peter Naur and Brian Randell, editors, *Software Engineering*, pages 138–155, 1969. Garmisch, Germany, October 1968.

[11] Kwok C. Ng. `fdlibm`: C math library for machines that support ieee 754 floating-point. freely redistributable; available at `http://netlib.bell-labs.com/netlib/fdlibm/`, March 1995.

[12] G. L. Steele and J. L. White. How to print floating point numbers accurately. In *Procedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 112–126, 1990.

[13] X3J11.1. Chapter 5, floating-point C extensions. Technical report, ANSI, March 29 1995.