Computer System Support for Scientific and Engineering Computation

Lecture 17 - June 28, 1988 (notes revised June 14, 1990)

1 Multiple-Precision Floating Point

In the last lecture, we stated that there is rarely any need for going beyond double precision. For example, in the late 1950's, people thought that triple precision would be needed to compute eigenvalues, but then the QR algorithm was discovered. We have also seen that accumulation of sums can be done using compensated summation, rather than going to a higher precision. In general, a clever algorithm can almost always obviate the need for precision higher than double. When computer hardware was more expensive than computer software, it made sense to pay for programmers to develop clever algorithms that would save on compute time. That is less true today. Thus we may see an increasing demand for multiple-precision floating-point packages. However at present, multiple-precision packages are not widespread. For example, the widely used IMSL library does not contain one. As background to introducing programming language extensions for supporting multiple precision, we need to discuss expression evaluation and "prototypes".

1.1 Expression Evaluation

In lecture 16, we discussed three reasonable methods of expression evaluation:

Fundamentalist FORTRAN The FORTRAN standard says that an operation must be evaluated in *at least* the maximum of the precision of its operands. In this method of expression evaluation, an operation is evaluated in exactly that precision.

Widest Available This method evaluates each operation to the widest precision available in the language (and, presumably, in the hardware).

Scan for Widest This scheme evaluates each operation to the widest precision that occurs in the expression.

The advantage of "Scan for Widest" over "Widest Available" on some machines is that it will go faster if everything is in single precision. The advantage of "Scan for Widest" over "Fundamentalist FORTRAN" occurs in expressions like `S = S + A*B + 3.0/7.0`, where `S`, `A`, and `B` are double precision variables. The programmer almost certainly wants `3.0/7.0` to be evaluated to double precision, but in the fundamentalist method it will be evaluated only to single precision. This situation can arise if `S` was originally declared in single

precision: it's awkward to modify every constant expression whenever you change the type of a variable.

Languages that can not count on having "Scan for Widest" often have to introduce special syntax. For example, FORTRAN-SC uses # to coerce expressions to dot-precision, rather than simply computing an expression to dot-precision whenever an element of the expression is of dot-precision type.

The disadvantage of "Scan for Widest" is that it can require the compiler to generate more instructions. On orthogonal instruction sets, explicit type conversion instructions may be needed. On architectures that always evaluate to the widest precision (usually extended), instructions may be needed to explicitly change the rounding to narrower precisions.

1.2 Prototypes

A common error when programming in C or FORTRAN is to define a function whose formal parameters are `REAL` and then pass it an argument of type `DOUBLE` (or vice-versa). On a machine where single and double precision have different exponent fields, the error is usually discovered immediately. However, if the program is running on a machine where single and double precision have the same size exponent field (IBM 370, or VAX using the F and D formats), then the program will probably not exhibit catastrophic behavior, but will lose some significant figures and will be subtly incorrect.

When the function and its reference are in the same file, it is possible for the compiler to check for a match between the types of the actual arguments and the formal parameters. If they are in different files then only the loader can check for a match, assuming the compiler has deposited enough information into the object files. However, just because the compiler and loader could check for matches, it doesn't mean that the language defines this as what should happen. There are a few programming languages that have defined the semantics of matching the types of arguments and parameters. Modula-2 (inspired by the Mesa language developed at Xerox PARC) requires that any functions that will be referenced in another file (in fact, in another module) be declared in a *definition module*. When a module references a function from another module, it must explicitly *import* it. When a module is compiled, the linker will check the types of function arguments against the types declared in the definition module named in the import statement.

The proposed ANSI standard for C has a slightly weaker form of inter-module type checking. It uses *prototypes*, which are a list of the argument types of a function. A function must be declared with prototypes before any reference, and the function definition must also include a prototype. The programmer will presumably use the `#include` macro facility to help ensure that all the prototypes match: a friendly loader could enforce it. We will use the term *prototypes* to refer to checking of procedure argument types, independent of the details of its definition.

1.3 Language Support for Multiple Precision

There are a number of multiple precision floating point packages available today. One is the MP package developed by Richard Brent around 1976-78; versions are published in various ACM journals. Another is the Numerical Turing language developed by Tom Hull in the 1970's. Still another is ACRITH, although it only provides working precision and dot-precision, rather than a whole range of multiple precisions. Multiple precision is distinct from BIGNUM integer/rational arithmetic present in lisp, in that BIGNUM

arithmetic is exact, and never performs any rounding. Multiple precision rounds like normal precision, but has more significant digits. Several of the symbolic mathematics packages support multiple precision floating point in addition to BIGNUM arithmetic. For example, BIGFLOAT was implemented in MACSYMA in the 1970's by Richard Fateman.

The problem with these packages is that they are inconvenient for the casual user. If you have an algorithm coded up in FORTRAN, and decide (perhaps at runtime) that it needs more precision, you have to recode the algorithm using one of these packages. The proposal we are about to give is not a package, but rather a few language extensions that build on "Scan for Widest" expression evaluation and prototypes. Hull's Numerical Turing has similar attributes; we shall not dwell here upon how his differs from ours.

Almost every language has two distinct floating point types. Multiple precision requires a third type. The declaration `EXTENDED*10 X;` declares `X` to be an extended variable stored in 10 bytes, which corresponds to the 80 bit extended precision used on the Intel 8087. The quadruple precision available on VAX, IBM 370 and Precision Architecture would correspond to `EXTENDED*16`. In a language with dynamic memory allocation, the size of the precision can be a variable. The code for a subroutine might look like this:

```
PROC(X, ....)
        EXTENDED*16 X
        REAL  Y, Z

        P = ....
        CALL PRECASSG(P, X)
        X = X + Y*Z
```

The `EXTENDED` declaration just allocates space for `X`. The actual precision of `X` is set with a call to `PRECASSG`. It is an error to set the precision greater than that declared in the `EXTENDED` declaration. Since expressions are evaluated using "scan for widest", the product `Y*Z` will be evaluated in precision `P`. The use of prototypes matches the precision of function arguments and parameters. To implement extended precision, there must be a descriptor associated with each extended precision variable that keeps track of its current precision. When compiling an expression, the compiler must generate a prologue that computes the maximum of the precisions of all the variables, in order to know in what precision to compute the expression.

1.4 Using Multiple Precision

A problem for the programmer is to determine what precision to use. The programming model we have in mind is to first compute an expression using working precision. If the accuracy is poor, then recompute in a higher precision. Since computation time (for all but inordinately high precision) goes up roughly as the square of the precision, the time spent in the previous computation is negligible compared to the final computation time. So the problem becomes, how to determine the accuracy of a computation? There are three main techniques. The first is to perform an error analysis before writing the program, so that the precision requirement is known in advance. For instance, when computing the zeros of a quadratic equation, the error analysis shows that you need carry at most twice as much precision as you want in the final answer. The second method is to perform a running error analysis as you compute the result. The third method is to use interval analysis.

The cost of an interval calculation can be up to four times the cost of a normal calculation, since computing the product of two intervals $[\underline{x}, \overline{x}]$ and $[\underline{y}, \overline{y}]$ can require computing all four products $\underline{x}\underline{y}$, $\underline{x}\overline{y}$, $\overline{x}\underline{y}$, and $\overline{x}\overline{y}$. However, interval analysis is the simplest method for the programmer. When used without multiple precision arithmetic, interval analysis is an unconstructive critic: it tells you your answer is bad, without offering any way to make it better. When combined with multiple precision arithmetic, interval analysis is quite useful. If the interval size of the final answer is k times larger than what you want, simply redo the calculation using more precision. In most cases, $\log_2 k$ extra bits are all that is needed, although some computations will require more.

One thing that can go wrong is that the inital calculation in working precision can overflow. We saw an example of that in lecture 12, with the expression [1]

$$\sum_{n=0}^{N}(-x)^n/n! = 1 - x(1 - \frac{x}{2}(1 - \frac{x}{3}(\cdots - \frac{x}{N}))\cdots)),$$

When that happens, simply try the calculation again in higher precision. If the point where the overflow occurs has moved, the overflow was probably due to insufficient precision. If it is in the same spot, the overflow is probably deserved, that is, the function being evaluated probably has a singularity.

1.5 Algorithms for Multiple Precision

Most multiple precision packages require that the numbers be stored in a special format. Is it possible to do multiple precision arithmetic with no special formats, and without any "bit-twiddling", that is, using only normal arithmetic operations? The answer is yes. The key is the following result. To state it, we will use the notation $[a+b]$ to mean $a+b$ rounded to working precision.

Theorem 1 *If $|p| \geq |q|$ and $z = [p+q]$, then $p+q = z+\zeta$ exactly, where $\zeta = [[p-z]+q]$.*

This holds for any machine that has a guard digit and either has a binary or rounds by chopping. Thus it holds for the IBM/370, because although it is hexadecimal rather than binary, it chops. It does not hold for a decimal calculator that rounds. The example below is a counterexample for that case, assuming arithmetic rounded to 5 significant decimals.

$$\begin{aligned} q &= 0.99997 \\ p &= 0.99998 \\ p+q &= 1.99995 \\ z = [p+q] &= 2.0000 \\ p-z &= -1.00002 \\ [p-z] &= -1.0000 \\ \zeta &= -0.00003 \end{aligned}$$

Thus $z + \zeta = 1.99997 \neq p + q$. A proof of the theorem in the case when the radix is 2 is given in Knuth's *Seminumerical Algorithms* (2nd edition) as Theorem C in section 4.2.2. Appendix A gives a variant of this formula, extracted from *A Survey of Error Analysis*, that works on all known machines except the Cyber 205.

[1] In single IEEE precision, this particular expression gets very large but doesn't quite overflow unless x is so big that e^{-x} underflows. But other examples exist which will overflow.

To do multiple precision arithmetic without any special formats, we will represent a multiple precision number x as a sequence of numbers in working precision $(x_1, \cdots, x_n)$. You can recover x by summing the x_i exactly, that is $x = \sum x_i$. When two such multiple precision numbers are added, we need a way to collapse the sum. This process is called *distillation*. Distillation replaces a sum $x_1+x_2+\cdots+x_n$ with a shorter sum $x'_1+x'_2+\cdots+x'_{n'}$ that has exactly the same value, using the formula from the theorem as a basic step. The distillation algorithm is given in the appendix. The proof that the algorithm works is difficult: see the article by Pichat in *Numerische Mathematik*, or the discussion of the Bohlender algorithm in the book by Kulisch and Miranker.

Appendix A

Distillation algorithm for computing

```
      S = 0.
      DO 9 J = 1,N
9     S = S + X(J,...)
```

Distillation Algorithm :

```
      S = 0.
      C = 0.
      DO 9 J = 1,N
        Y = C + X(J,...)
        T = S + Y
        F = 0.
        IF (SIGN(1.,Y) .EQ. SIGN(1.,S)) F = (0.46 * T - T) + T
        C = ((S - F) - (T - F)) + Y
9       S = T
      SUM = S + C                        (slightly better than S)
```