A Demonstration of Presubstitution for ∞/∞

What Is It ?

Presubstitution supplies a value or a simple procedure to cope with floating-point exceptions
Invalid Operation, Overflow, Division-by-Zero, Underflow and Inexact
in a way better than aborting execution of a program when an exception occurs. The IEEE 754 (1985) standard specifies *default* presubstitutions when a programmer specifies no others:

Exception	IEEE 754 Default Presubstitution
Invalid Operation	NaN
Overflow	$\pm\infty$ with the appropriate sign
Division-by-Zero	$\pm\infty$
Underflow	*Gradual,* to a *Subnormal* number or zero
Inexact	Rounded (or infinite Overflowed) Result

“Division-by-Zero” really means “Infinite Result Exactly from Finite Operands”. An *Invalid Operation* creates a new *NaN* (*N*ot-*a*-*N*umber) just when any other finite or infinite result would most likely be more confusing; examples are $0/0$, ∞/∞ . $0\cdot\infty$, $\infty - \infty$, real $\sqrt{\ldots < 0}$, … among many others. *Gradual Underflow* ensures that the gap between adjacent floating-point numbers is a monotone non-decreasing function of their magnitudes instead of jumping by too many orders of magnitude as the numbers pass through 0 , which is what happens if underflows flush to **0.**0 . All but a few esoteric programs, most concerned with exact integers, ignore *Inexact* exceptions.

The default presubstitutions specified by IEEE 754 were chosen carefully to maximize the likelihood that programmers could either ignore the exceptions or postpone their detection and handling to convenient places in their programs. To this end, each exception *raises* its own *sticky* flag that can be *reset* (put down) only by an explicit command in the program, which may also test, save or restore a flag. All that should be a long story for another day.

Alternatives

What alternative to an IEEE 754 default might a programmer choose? One alternative aborts or at least pauses when a subprogram being debugged generates an unanticipated exception. Aborting should never be the choice for a subprogram released to users lest control be wrested from a user’s main program and sent into *Limbo*. (This is where innocent but unbaptized souls languish forever between Heaven and Hell.) Another alternative filters out exceptions by testing operands in advance and branching to special fix-up code whenever necessary. This can degrade performance when the exception would occur so rarely that the branch would almost never be taken but the test levies a tax upon performance all the time. More important, tests-and-branches complicate a program’s structure and enlarge its capture-cross-section for mistakes.

Ideal Language

Ideally a subprogram’s text should depart minimally from the simplest subprogram that handles all the normal cases well. Ideally, rare exceptions generated within a subprogram’s inner loop should be handled by a footnote (like a flag-test-and-branch) appended after the loop or by a preface (like an invocation of a non-default presubstitution) prefixed before the loop.

In a block-structured programming language, an invocation of a non-default presubstitution acts like a declaration of local variables at the beginning of a block. These variables are irrelevant outside the block’s scope, as is the non-default presubstitution. Depending upon a language’s conventions, local variables may or may not be inherited by nested sub-blocks; the same goes for presubstitution. In a programming language whose variables are all global, so is presubstitution.

Designers and implementors of programming languages tend to regard presubstitution as a nuisance, at best a form of “syntactic sugar” for other programming locutions, at worst an invitation to engage in the arcana of floating-point hardware traps. Actually, presubstitution is the *only portable* way found so far by which applications programmers may state how they wish to handle certain kinds of transgressions, corner cases and other exceptions generated sometimes by extreme data and often by accidents incidental to an otherwise auspicious algorithm.

“… the *only portable* way ” ?

Readers who balk at this extravagant assertion are invited to test their programming prowess on an example whereon non-default presubstitution works better than almost everything else.

Example: The Holy Grail

Let us consider inner loops from a program designed to compute a few hundred eigenvalues and eigenvectors of a real symmetric n-by-n matrix of huge dimensions $n > 20000$ on a distributed computer system with many processors working in parallel. Such computations arise in physics, chemistry and structural mechanics, among other areas. This program succeeds in establishing the required orthogonality among eigenvectors computed on different processors without slowing them down with intercommunication that would consume rather more time than the arithmetic takes. This program has been called “The Holy Grail” because it has been sought for decades. Currently it is the only program that can guarantee orthogonal eigenvectors for matrices too many of whose eigenvalues are clustered tightly, as occurs for structures with many (near-)symmetries.

Details can be found in papers by B.N. Parlett and others in *Acta Numerica* (1995), *Linear Algebra and Applications* (1997 and 2000), and *SIAM J. on Matrix Analysis and Applications* (2004), with more to come. But those details won’t matter here.

Very little about the loops’ context need be known to appreciate how they benefit from presubstitution, and that little will be supplied forthwith.

Input data are a positive integer index m , a real x and two real arrays D(0:n) and LLD(0:n) of finite nonzero numbers except D(0) := LLD(0) := LLD(n) := 0 . Otherwise $LLD(j)\cdot D(j) > 0$. The arrays determine the matrix’s eigenvalues $z_1 < z_2 < z_3 < \dots < z_{n-1} < z_n$ as the n zeros of two rational functions $f(x)$ and ß(x) generated by the loops. The loop running forward through the arrays generates $f(x)$; running backwards generates ß(x) ; both have the same zeros z_j .

Unfortunately each rational function has n poles too, one between every pair of adjacent zeros z_j , and another pole at $x = \infty$. The n–1 finite poles of f usually differ at least slightly from those of ß . These slight differences will make all the difference when an eigenvector is to be computed after its eigenvalue z_m has been found, but the eigenvector is a detail that won't matter here. What will matter here is the computation of z_m by a zero-finding process that generates a sequence of values of x intended to converge to z_m . In principle, z_m is a value of x at which $f(x)$ and ß(x) reverse sign by descending through zero.

In practice z_m can lie arbitrarily close to a neighbor $z_{m\pm1}$ and, worse, so close to a pole of $f(x)$ and/or ß(x) that their graphs, plotted at discrete floating-point numbers, may fail to reveal z_m by crossing the x-axis there. In such a case z_m will be revealed as a discontinuity of an integer index k(x) computed in the same loops as compute $f(x)$ and ß(x) . This index has the property that $z_{k(x)} < x \le z_{k(x)+1}$ provided we adopt conventions $z_0 := -\infty$ and $z_{n+1} := +\infty$. Then z_m is the value of x at which k(x) jumps from m–1 to m ; this jump can be found easily by binary chop because, despite roundoff, the computed k(x) is a nondecreasing function as expected. (This is hard to prove even after the observation that, when finite, $\partial f/\partial x \le -1$ and $\partial ß/\partial x \le -1$.)

Thus has the context of the loops been established: By means external to the loops, a sequence of real samples x will be generated converging to a value z_m at which k(x) jumps up from m–1 to m and, ideally, $f(x)$ and ß(x) decrease through zero as x increases.

Here are the loops:

Forward loop:
```
  k := 0 ;    y := –x ;
{  Presubstitute  1.0  for  ∞/∞ ;
   For  j = 1  up to  n  do
    {  ƒ := D(j) + y ;
        y := (y/ƒ)·LLD(j) – x ;
        k := k + SignBit(ƒ) ;
     };}…  Now  ƒ = ƒ(x)  and  k = k(x) .
```

Backward loop:
```
  k := 0 ;    y := D(n) – x ;
{  Presubstitute  1.0  for  ∞/∞ ;
   For  j = n–1  down to  0  do
    {  ß := LLD(j) + y ;
        y := (y/ß)·D(j) – x ;
        k := k + SignBit(ß) ;
     };}…  Now  ß = ß(x)  and  k = k(x) .
```

Function SignBit(f) returns a copy 1 or 0 of its argument's sign bit even if the argument is ±0 . The statement "Presubstitute 1.0 for ∞/∞ " affects only the division (y/f) or (y/ß) and only when it would be (∞/∞) , in which case 1.0 is substituted for a quotient that would otherwise create a NaN , raise the Invalid Operation flag, and destroy the validity of subsequent passes around the loop. Before discussing alternatives to the "Presubstitute …" statement we must introduce one more complication: The variables x, k, y, f and ß need not be simply scalars.

Most computers nowadays divide too slowly. Each pass through the loop is spent mostly waiting for its division to finish even if it overlaps the signbit(…) function and anticipatory fetches of elements from the arrays D(…) and LLD(…) into registers. Many computers pipeline divisions so that they can be overlapped. These computers can interleave a loop's computations for several samples of x perhaps all estimates of one eigenvalue z_m or else simultaneous estimates for, say, z_{m-1} , z_m and z_{m+1} . Consequently the variables x, k, y, f and ß will be arrays upon which arithmetic is performed elementwise to obtain results for several samples of x simultaneously, thus speeding up computation by exploiting more fully a processor's capacity for concurrency and, incidentally, reducing cache misses during accesses to huge arrays D(…) and LLD(…) .

Presubstitution in the Loops

How does their non-default presubstitution work? It will take effect if (an element of array) ƒ or ß vanishes during some pass other than the last through the loop. Then y becomes ±∞ , as does ƒ or ß in the next pass, wherein (y/ƒ) or (y/ß) is changed from (∞/∞) to the correct limit 1 that would have been obtained if the earlier ƒ or ß had merely become infinitesimal instead of vanishing. And the same non-default presubstitution 1 works if, instead of vanishing, ƒ or ß becomes so tiny that y overflows to ±∞ in some pass other than the last (where it won't matter).

Now let us consider alternatives to non-default presubstitution. How must the loops change if "Presubstitute 1.0 for ∞/∞ " is unavailable and thus deleted from the foregoing loops' texts?

An Obvious Alternative

An obvious alternative leaves the loop bodies unchanged and lets ∞/∞ create a NaN that will propagate to a final value of ƒ or ß . Then tests appended after the loop can branch to recompute the spoiled value(s) of ƒ or ß in a slower loop devised to prevent division-by-zero or overflow from creating ∞ . This obvious alternative works too slowly on hardware that traps whenever it encounters a NaN and then takes an order of magnitude longer than a division to propagate the NaN through arithmetic operations. The slowdown is intolerable if one processor in a distributed system creates a NaN during an early pass through the loop, forcing other processors to wait while the unlucky one copes with its NaNs. Such NaNs are very improbable if the given data is chosen at random, but occur often if the given huge matrix has many tight clusters of eigenvalues, which is the the situation that motivated the Holy Grail's development. Therefore this obvious alternative is unacceptable to a programmer who intends his program to be *portable* (not much slower than competing programs) to hardware that handles NaNs too slowly.

An obvious fix-up for the obvious alternative is to turn the trap on NaN into a *break* out of the loop, abandoning whatever it has computed and beginning anew with a slower loop devised to prevent division-by-zero or overflow from creating ∞ . For hardware that does not trap, the break has to be inserted into the loop as a test-isNaN(y)-and-branch. Only an unusually clever optimizing compiler will plant the test somewhere overlapped by a division logically subsequent to the test; otherwise the test-and-branch will slow down the loop when no NaN occurs. This fix-up also increases the likelihood that all the elements of arrays k(x) and ƒ(x) or ß(x) will be computed in the slower loop though only one element would have been contaminated by NaNs. So, this obvious fix-up is unacceptable to a programmer who intends his program to be *portable* (not much slower than competing programs) regardless of how slowly hardware handles NaNs.

A less obvious way to fix up the obvious alternative partitions each loop's execution into $\lfloor n/N \rfloor$ batches for some suitable positive integer N rather less than n . Each batch runs through the original loop at most N times and then tests isNaN(y) to determine whether the batch should be repeated with the loop body modified to preclude NaNs, albeit slowly, starting from the data saved at the end of the previous successful batch. The batched forward loop might look like this:

The Batched Forward Loop:

```
k := 0 ;    y := –x ;
For  i = 0  up to  floor(n/N)  do  …  a batch of loops:
 {  ko := k ;   yo := y ;
    For  j = 1 + i·N  up to  min(n, (1+i)·N)  do  … the faster loop:
        {  ƒ := D(j) + y ;
           y := (y/ƒ)·LLD(j) – x ;
           k := k + SignBit(ƒ) ;
         } ;
    If  any(isNaN(y))  then  do   ... a batch of slower loops:
        {  k := ko ;  y := yo ;
           For  j = 1 + i·N  up to  min(n, (i+1)·N)  do
             {    ƒ := D(j) + y ;
                  q := if  isInfinite(y)  then  1.0  else  y/ƒ ;
                  y := q·LLD(j) – x ;
                  k := k + SignBit(ƒ) ;
              };};}…  Now  ƒ = ƒ(x)  and  k = k(x) .
```

The batched loop appears twice-nested; actually it is thrice-nested because x, y, k and ƒ are generally arrays upon which arithmetic is performed elementwise. Predicate any(isNaN(y)) turns true when any one element of y becomes NaN , in which case the slower loop is executed. An alternative that may run faster on big arrays y is to test whether the Invalid Operation Flag has been raised, and then lower it and execute the slower loop. Therein the predicate isInfinite(y) acts elementwise to put either 1.0 or the appropriate element of the elementwise quotient-array y/ƒ into a temporary array q . Ideally, any unused quotient in the array y/ƒ should not be computed; if it is, the NaN created thereby will also raise an Invalid Operation Flag that should be lowered lest it distract the program(mer) later. Perhaps this ideal is too much to ask.

Omitted from the text of **The Batched Forward Loop** is any indication of how N was chosen. If too small, too much time will be wasted saving arrays ko := k and yo := y and either scanning y for any(isNaN(y)) or else testing a flag to which access is typically slow.

If N is too big, too much time will be wasted when an element of ƒ vanishes, spawning an ∞ which spawns a NaN in the next pass through the loop, which NaN then retards subsequent passes through the faster loop. This can happen only when an element of x is so unlucky as to fall upon one of fewer than $n^2/2$ arguments; these are all the poles of the loop's sequence of values ƒ . When the array of values x all approximate eigenvalues in a tight cluster (the Holy Grail's *raison d'être*) more than one of x's elements may be unlucky, most likely in different batches. So, if N is too big and x too unlucky, too much time may be wasted more than once.

An optimal choice for N is roughly $\sqrt{2s \cdot n/p}$ in which p is the probability of an unlucky x and small factor s is very roughly the ratio of two times, first the time to store arrays k and y , and second the time taken to pass once through the faster loop when y is NaN . The second time varies by orders of magnitude over diverse machines. How shall a conscientious programmer weigh these imponderables when designing a single program to be portable (competitive in speed *etc*. with other programs) to diverse machines some of them not yet built?

The batched fix-up for the obvious alternative to non-default presubstitution imposes, upon our conscientious programmers, burdens that many of them are perversely proud to bear, but also imposes something worse upon the rest of us: An elegant and short loop has been turned into a monster with a bloated capture-cross-section for unreliable and unpredictable performance.

An Unobvious Alternative

An unobvious alternative to non-default presubstitution modifies the loops to prevent division-by-zero or overflow from creating ∞ . One modification follows the statement " f := D(j) + y ; " in the forward loop by an insertion " f := f + µ ; " wherein µ is smaller than uncertainty inherited by f from roundoff, and yet big enough that the next expression " (y/f)·LLD(j) " cannot overflow. A suitable constant µ , if one exists, must satisfy inequalities like

$$æ\cdot\min_j\{|D(j)|\}/4 \ge \mu \ge 2\cdot\max_j\{|D(j)\cdot LLD(j)|\}/\Omega$$

wherein

$$æ := 1.0 - \text{NextAfter}(1.0,\ 0.0) \quad \text{is a rounding error threshold and}$$
$$\Omega := \text{NextAfter}(+\infty,\ 0.0) \quad \text{is the biggest finite floating-point number.}$$

For IEEE 754 Single Precision $æ = 1/2^{24} \approx 6/10^8$ and $\Omega \approx 3.4\cdot10^{38}$. For IEEE 754 Double Precision $æ = 1/2^{53} \approx 1.1/10^{16}$ and $\Omega \approx 1.8\cdot10^{308}$.

In the backward loop the analogous modification follows the statement " ß := LLD(j) + y ; " by " ß := ß + µ ; " where

$$æ\cdot\min_j\{|LLD(j)|\}/4 \ge \mu \ge 2\cdot\max_j\{|LLD(j)\cdot D(j)|\}/\Omega\ .$$

These inequalities that constrain µ also restrict the range of data D(…) and LLD(…) that the program can accept to less than half the exponent range that an unmodified program can accept. This range restriction, clearly unacceptable in single precision, is probably acceptable in double precision though it puts the modified program at a competitive disadvantage when displayed in the program's documentation. The range restriction could be relaxed if µ were recomputed in every pass through the loops, but only at the cost of further slowing down the loops on every machine as a result of catering to those machines bogged down by infinities and NaNs.

Presubstitution as "Syntactic Sugar"

A clever optimizing compiler could translate the loops' command "Presubstitute 1.0 for ∞/∞ " into a replacement of the assignment " y := (y/f)·LLD(j) – x " in the forward loop by

```
" q := if (isInfinite(y) & isInfinite(f)) then 1.0 else y/f ;
  y := q·LLD(j) – x ".
```

An analogous modification could be compiled for the backward loop. The predicate isInfinite(f) is redundant, but a compiler is unlikely to be clever enough to know that. The time spent upon the two predicates isInfinite(…) might be overlapped by the division y/f executed speculatively and superseded by a conditional move of 1.0 when the unwanted quotient would be NaN. If raised, the Invalid Operation flag would have to be adjusted too. And if a compiler can insert the two assignments that de-sugar the presubstitution command, why can't the programmer do it?

Both tests in " q := if … " are redundant since the hardware performs them in the course of every division. If the hardware can trap on an Invalid Operation to a lightweight handler that can set the quotient to 1.0 quickly, the programmer will prefer this handling provided it runs faster than the tests when no trap occurs. Then the programmer will wish the compiler was clever enough to translate his two assignments " q := …; y := …; " back to the original single assignment plus a command that enables the trap-handler and tells it outside the loop what to do inside.

Alas, no compilers are that clever today. Moreover older hardware cannot speculate; and hardware that can speculate does so by invading resources that could otherwise be devoted to concurrency. Concurrency is needed to cope with arrays x, k, y, ƒ, ß and now q speedily.

Is there a limit to the size of these arrays compatible with
speculative execution without performance degradation?

If anybody can answer this question it is the compiler, not the applications programmer trying to write a program portable to as wide as possible a range of today's and tomorrow's computers. Here "portable" means at least competitive with other software in speed, robustness, range, accuracy and ease of use. And the program should be written once albeit tested everywhere.

The programmer's intent is expressed more transparently by a " Presubstitute … " statement than by assignments like " q := … ; y := … " which introduce a new array variable q and raise that question about array sizes. The arrays entail either implicit or (in most languages today) explicit looping on two indices, one the loops' explicit index j and another running over the arrays. Which loop should be innermost? Are the arrays x, k, y, ƒ, ß best broken into smaller blocks? How small? These are the questions a programmer wishes the optimizing compiler to answer perhaps differently on different hardware. Substituting the two assignments " q := … ; y := … " for " Presubstitute … " cannot make the questions easier for the compiler. This is why the compiler, not the programmer, should de-sugar non-default presubstitution if anyone must.

A Very Unobvious Alternative

Non-default presubstitution is unavailable in today's programming languages. Maybe next year's? In the meantime, what should a conscientious programmer do to implement the Holy Grail as efficiently as possible in just one portable code? Here are some suggestions:

Forward loop:
```
  k := 0 ;    y := –x ;
{ Presubst.  ±Ω  for Overflow and Div-by-0 ;
   For  j = 1  up to  n  do
    {  ƒ := D(j) + y ;
        y := (y/ƒ)·LLD(j) – x ;
        k := k + SignBit(ƒ) ;
     };}…  Now  ƒ = ƒ(x)  and  k = k(x) .
```

Backward loop:
```
  k := 0 ;    y := D(n) – x ;
{ Presubst.  ±Ω  for Overflow and Div-by-0 ;
   For  j = n–1  down to  0  do
    {  ß := LLD(j) + y ;
        y := (y/ß)·D(j) – x ;
        k := k + SignBit(ß) ;
     };}…  Now  ß = ß(x)  and  k = k(x) .
```

Oh dear, non-default presubstitution again! But this one is easy for a programmer to de-sugar: Remove the " Presubst. … " statement and insert the conditional assignment

" If isInfinite(y) then y := CopySign(Ω, y) ; "

at the loops' beginning just ahead of " ƒ := D(j) + y " and " ß := LLD(j) + y ". Doing so replaces any infinite y by the biggest finite floating-point number ±Ω with the same sign. It also reduces slightly the range of inputs acceptable to the program since the suggested presubstitution works only if every $|D(j)| < æ·\Omega$ and every $|LLD(j)| < æ·\Omega$. (Recall that æ is the roundoff threshold.) If the computer takes too long to decide the predicate " isInfinite(y) " replace it by "($|y| > \Omega$) ". Verifying the validity of the foregoing suggestions is left to the diligent reader.

There is one more detail to consider. The foregoing suggestions perform well only if Overflow or Division-by-Zero can generate an infinite y at most a few times in either loop for any single unlucky scalar x that approximates an eigenvalue z_m . There are good reasons to believe that such is the case, but no proof. If our belief is wrong then machines that generate and/or handle infinities too slowly may also run these loops too slowly on rare occasions.

How Presubstitution Should Work

Except for C99 and Fortran 2003, and despite helpful capabilities latent for over two decades in hardware conforming to IEEE 754, programming languages offer little help to programmers trying to handle floating-point exceptions conscientiously. The subject can hardly be discussed without arguments at cross-purposes. For example take the word "exception":

To programming languages like C, C++ and Java, an exception is the trap or transfer of control precipitated by some unusual or untoward event discovered by the program or signalled to it from without. The exception is the response to the event; and a jump is the only response allowed.

To IEEE Standard 754 a floating-point exception is an event, a transgression that need not be unusual nor undesired though transgressions demanding a program's attention may well be both unusual and undesired. The word "exception" is used sometimes for a single event, sometimes for a class of similar events. The standard's default response to every exception is presubstitution and a side-effect (the raising of a flag if it has not already been raised), not a jump.

Of course, a programmer may have a good reason to override IEEE 754's default by requesting a jump in response to an exception, especially while debugging. But computing environments that superimpose jumps as their default responses to certain exceptions — typically the Invalid Operations, Overflows and Divides-by-Zero declared to be Errors in older computer systems — violate the standard and expose software users to serious hazards. Here are two examples:

In June 1996 the *Ariane V* rocket turned cartwheels and blew up half a billion dollars worth of instruments intended for European science in space. The proximate cause was the programming language ADA's policy of aborting computation when an *Arithmetic Error,* in this case an irrelevant Floating-Point → Integer Overflow, occurred. See `http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html` and p. 22 of `http://www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf` .

In Sept. 1997 the *Aegis* missile-cruiser *Yorktown* spent almost three hours adrift off Cape Charles VA, its software-controlled propulsion and steering disabled, waiting for Microsoft *Windows NT* 4.0 to be rebooted after a division-by-zero unexpectedly trapped into it from a data-base program that had interpreted an accidentally blank field as zero. See `http://www.gcn.com/archives/gcn/1998/july13/cov2.htm` .

Exceptions become Errors only when mishandled.

Results incorrect because of ignored exceptions are hazardous too. Computing environments can offset this hazard by attaching annoying little warnings to results displayed while a flag still raised signifies that some serious exception remains *unrequited*. Protests against such warnings will come from the spiritual descendants of people who tied down noisy pressure-relief valves on steam boilers, disabled smoke alarms in kitchens, or disdained seat-belts while driving. How to redirect their protests tactfully towards a programmer who neglected a duty to reset the flag or else augment dubious results with an intelligible explanation is a story for another day.

No flag need be raised by a non-default presubstitution's treatment of an exception since this treatment reflects a programmer's intentions no differently than do non-exceptional assignments. When applicable, presubstitution takes effect without cluttering a program's source-text with tests-and-branches; this feature alone should rally language aesthetes to its support.

We can model the effect of presubstitution by imagining a table built into the hardware. Let's call it the *P-Table*. Its entries are indexed by the kind of exception served and, if relevant, the direction of rounding. One entry is for divide-by-zero; four are for overflow, one for each of the four rounding modes; one is for ∞/∞, another for 0/0, and so on. All entries contain a numerical value or NaN, a bit to enable or disable flag raising, and a bit that enables or inhibits copying of a sign bit. For example, the entry for ∞/∞ is a NaN by default but may be replaced by a number like 1.0, and a bit set to inhibit copying of its sign from the exclusive OR of the operands' signs. The entries for division-by-zero and for overflow rounded to nearest (the default) are ∞ by default with a bit set to enable copying of its sign from a penultimate result not yet delivered to its ultimate destination. Each ∞ can be replaced by any other number including Ω, which is the default entry for Overflow rounded towards zero. And so on. The P-Table's entries for Underflow are complicated; see `<http://www.cs.berkeley.edu/ARITH_17U.pdf>`.

A presubstitution command behaves as if it wrote its operand(s) over one or more entries in the P-table. If this table exists in the hardware, the operand(s) enter(s) the floating-point pipeline like any other floating-point operation's operand(s) but the destination is the P-Table. If this table exists in memory to be accessed by a trap-handler or a math. library subprogram, the compiler has to treat the command like a floating-point instruction not to be "optimized" by motion past other floating-point instructions, and may have to allow the floating-point pipeline to empty before continuing execution. This implies that presubstitution commands are best used sparingly. The system must also provide facilities to save and restore the P-table when entering and leaving the scope of a non-default presubstitution.

To treat a non-default presubstitution command as if the P-Table were in hardware when it isn't, the compiler will have to de-sugar the command by planting tests and branches in the program. These will slow the program down, but not much more than if the programmer had to plant such tests and branches in the source-code's text. Still, the programmer should limit the scope of non-default presubstitution as narrowly as possible to avoid unnecessary degradation of performance.

Language support for non-default presubstitution, where it is applicable, does more than allow the programmer to express intentions transparently. This support enhances portability by hiding from the programmer architectural vagaries handled better by a cleverly optimizing compiler. Moreover, supplying this linguistic support offers also an opportunity for hardware designers to simplify floating-point traps and provide only the few capabilities actually found worthwhile for floating-point exception-handling, instead of catering to all conceivable generalities, like jumps anywhere and accesses to variables everywhere, that can never be used in portable codes.

Conclusion

Floating-point exceptions resemble migratory songbirds; they are seen only rarely in most places but, when seen, they are seen in flocks. Whatever brings one usually brings others. This is the flaw in arguments that would justify excessively slow handling of exceptions by their rarity.

Diverse approaches to floating-point exception handling are necessitated by the diversity of mathematical singularities; these have defied classification. Programmers will always have to choose from among preemption (testing to filter out exceptions), non-default presubstitution, checking after every potentially dubious result, or testing a summary flag afterwards. Languages inhospitable to modes like non-default presubstitution, and to side-effects like flags, must let programmers choose other ways to handle exceptions, typically ways that divert control and thus invite more mistakes. Examples include ADA's drop-through to a handler (if one exists) for an "Arithmetic Error", and BASIC's "On *ERROR* go to …" and "On *ERROR* gosub …". Imposing a discipline upon such jumps restrictive enough to enable necessary compiler optimizations but not so restrictive as to vitiate the jumps' usefulness raises questions about scope, visibility and inheritance similar to questions raised by modes and flags. These questions cannot be addressed adequately solely by language designers and implementors with today's advanced Computer Science degrees but with inadequate exposure to the range of numerically exceptional situations.

Each of the aforementioned exception-handling modalities has a natural rôle in various numerical algorithms even if one modality can be simulated by another for a price. When linguistic lacunae in a programming environment deny a programmer the natural choice, the robustness of the programmers's software is jeopardized. And then it becomes our software.

More important, linguistic support for apt locutions like presubstitution enhances the productivity of conscientious applications programmers, the ones who worry about exceptions, with less risk of burning them out.

CAVEAT: THIS DOCUMENT IS A WORK IN PROGRESS CONTINUALLY SUBJECT TO CHANGES IN RESPONSE TO CONSTRUCTIVE SUGGESTIONS FROM READERS.