Symbolic Execution and NaNs: Diagnostic Tools for Tracking Scientific Computation

Richard J. Fateman

Electrical Engineering and Computer Sciences Dept.
University of California, Berkeley, 94720-1776, USA

fateman@cs.berkeley.edu
http://http.cs.berkeley.edu/~fateman

Abstract

When unexpected results appear as output from your computation, there may be bugs in the program, the input data, or your expectation. In grappling with difficult diagnostic tasks one direction is to seek tools to analyze the symbolic mapping from the input to the output. We are generally ahead in this game if we can understand complex system behavior without necessarily studying detailed program source code. This is true especially if the source is in an unfamiliar language, difficult to understand, or simply unavailable. We explain two tools: time-honored "symbolic execution" which requires some kind of computer algebra system, and a novel modification, NaN-tracking. This is a simplified version of symbolic execution that is more easily implemented in a conventional language like Fortran or C. The principal requirement of this second approach is a competent implementation of a compiler and run-time system. In particular, the language system must provide access to features of the IEEE-754 binary floating-point arithmetic standard [7]. While our own experiments are based in part on an implementation in Lisp, the mechanisms we use should be accessible from languages in nearly every C-based UNIX workstation used for scientific computing.

1 Introduction

Given a program to debug one generally compares its apparent functioning to its specifications. Deducing the function computed by some elaborate or perhaps just hidden (black-box) routine `f` and observing that $f(3) = 9$ and $f(4) = 16$ might lead one to guess that $f(x) := x^2$. This may be false, of course, since there are infinitely many polynomial functions that fit the data so far and the restriction to only polynomials is certainly gratuitous and unjustified *a priori*. If we have a symbolic system, or more likely a special version of a numeric language like Fortran "extended" to allow symbolic input, output, and computation, we could try using a symbolic input like p. If, computing $f(p)$ we find it comes out as p^2, we have fairly strong confirmation that we have guessed correctly. If it comes out as

$$1.5p^2 - 3.5p + 6$$

or perhaps the more haphazard

$$(p-3)(p-4) * (\cos(p)/2 - 1) + 16(p-3) - 9(p-4),$$

both consistent with $f(3) = 9$ and $f(4) = 16$, we would also know exactly the shape of the function. By testing with a symbolic p we have gotten infinitely more information than by testing and finding that $f(2)$ is not $2^2 = 4$ but 5, in the first case, or $\cos(2)$ in the second.

Unfortunately, our test using p will be problematical if f is some elaborate function whose internal operation and side effects depended on whether its input was, say, greater than 7, or even, odd, or prime. We could claim that the input p might have any, or none of those properties. Consequently any program-path decisions must be somehow encoded in the result or computed with the aid of additional information[1]. Such problems with decisions make the implementation of an effective "extension to allow symbolic execution of Fortran" or similar conventional languages challenging [5]. Nevertheless, the information that can be obtained by symbolic execution is quite valuable, and persons engaged in software engineering or program proofs under high-reliability constraints, or with elaborate semantics such as interacting tasks, may indeed include symbolic execution as one tool [1] [4].

Let us just give a few examples of what has been attempted with symbolic execution for program understanding or proof.

There are proofs of the algebraic correctness of an asymptotically fast matrix multiplication routine. One can provide confirmation of subroutines in nuclear reactor safety code [5]. We will show a demonstration of numerical-style programs in which (say) a matrix that is nearly all conventionally numerical has added to one element a symbolic ϵ. One can observe how this perturbation propagates through calculations.

Beyond examples that compute "oblivious" algebraic functions (making no tests on the input), one can bite the bullet and actually implement multi-way branching. Loop-free programs can be exercised through the collection of different choices. Proofs of termination and correctness of invariants can sometimes be derived. Code for critical control systems (e.g. power-plant shutdown tests) have apparently been beneficiaries of this kind of analysis.

[1] perhaps "oracular" from asking a nearby human "is $p > 7$?"

Finally, before going on to the next stage, we should make one other minor observation: Traditionally a black-box program is a computation module whose detailed internal structure is not available for examination. Sometimes a program whose internal structure and source code is entirely available for inspection *may as well be hidden* if its operation is, *de facto* mysterious, and its size and structure presents a barrier to understanding. The 100,000 line "dusty deck" programs written by now-retired programmers may fit in this category.

Unfortunately, for any but the most determined practitioner, and except for particular "nice" contexts, symbolic execution is difficult. Here's why.

2 Difficulties with Symbolic Execution

Processes that depend on iteration and which halt on converging to a numerical value do not generally yield to this kind of symbolic simulation. It is not usually possible to test an expression of symbolic parameters for "smallness".

In fact, quite the opposite of getting smaller, even trivial processing steps on symbolic expressions result in what is known in the computer algebra business as "intermediate expression swell".

That is, the size of expressions involving symbolic parameters generally grows quite rapidly. Here is an example: three symbolic iterations of Newton's method

$$x_k \;:=\; \frac{x_{k-1}^2 + v}{2\,x_{k-1}}$$

to find the square root of v.

If we let $x_0 = r$, then we obtain an expression

$$x_3 \;:=\; \frac{r^8 + 28\,v\,r^6 + 70\,v^2\,r^4 + 28\,v^3\,r^2 + v^4}{8\,r^7 + 56\,v\,r^5 + 56\,v^2\,r^3 + 8\,v^3\,r}$$

which certainly "looks large," yet is an expression closer to $\sqrt{v}$. In fact, setting $v = 2$, and choosing r anywhere between 0.5 and 4, then x_3 agrees to $\sqrt{2}$, $1.41421356\cdots$ to three or more decimals. (Plotting this expression gives a kind of step function: negative values of r converge to the negative square root).

By contrast with this expression growth, in numerical computation, we know that if every element in an n by m matrix is a double-precision floating-point number our computer program needs only nm chunks of 64-bit storage to accomodate it. If any of the elements were symbolic, then more, perhaps much more, storage may be needed. In the symbolic mathematical computing arena, resources may be strained even representing a dense matrix that is only 25 by 25 entries if each element is a high-degree dense multivariate polynomial.

If we examine the expense of computing, "efficient" symbolic execution is a relative measure: such a programming system typically represents the combination of an interpreter for some traditional numerical language's semantics (say Fortran or Ada) plus some symbolic computation system. Thus symbolic execution may be hundreds of times slower than normal compiled execution. Wading through a long execution to find an error can be painful.

Even in the best of circumstances, looping and branching programs cause some difficulties, since the decision-points may require the evaluation of predicates whose truth is difficult to compute, or where vital information is simply unknown until "full-evaluation run time". Following alternate paths at each decision may work for a short program: the program execution becomes a tree rather than a single path. Ideally, for longer programs, variant paths can be merged so that the processing proceeding from a single point can continue regardless of how the program reached that point.

These difficulties have made symbolic execution unpopular.

3 A Simpler Alternative

As we have seen, symbolic execution can carry nearly all the arithmetical operations in full detail (although it doesn't ordinarily account for roundoff, overflow, etc.). If we are willing to forego much of this information we can abbreviate the symbolic computation in various ways, preserving only some aspects of the result. In particular, there is a convenient technique to reduce intermediate expression swell and simultaneously preserve a level of performance close to that of ordinarily floating-point arithmetic.

Instead of computing entirely with symbols, we use standard (hardware-supported) floating-point numbers entirely. We inject special floating-point quantities, the reserved "not a number" or NaN [7] operands $\{K_1,\ K_2, \cdots\}$ in some locations. The NaNs are distinguished in hardware (or software) from normal floating-point numbers by having the "largest" exponent in the representation. (128 in single, 1024 in double). We actually have a large number of distinct special symbols available: some 2^{22} in single precision, 2^{51} in double precision. In case we use only a few of them, we might give them names like red, green, blue.

In this simplified alternative model, instead of computing exact symbolic results, we are only allowed to say of a result:

1. It is an ordinary floating-point number, computed in the ordinary way.

2. It is not an ordinary number, and in particular it may depend on some particular K_i among the injected symbols.

3. If it depends on more than one of those symbols, we will only know one of them. We can arrange for it to be the symbol of lowest index.

In principle, the programming language can be Fortran, C, Ada, Lisp or nearly any other programming language, and the compilation need not change at all. That is, the optimizers, the special libraries, the use of parallel code on multiple processors, and the other technologies used in high-speed scientific computation need not be re-implemented. The ordinary number results are computed at full speed. The propagation of these injected symbols, NaNs is done at hardware speed through IEEE floating-point compliant operations, and (depending on the implementation of the treatment of reserved operations) may run without slowdown, or perhaps may suffer some slowdown by being deflected for treatment in the system's trap-handling software.

In fact, many people, including those who implement language processors, view NaNs from an unfortunate perspective, perhaps influenced by history, confusing them with "undefined" objects. With this in mind software compiler vendors generally neglect to implement the needed trap routines, and some even use the traps for other, conflicting, signalling, so it cannot be assumed that the programs in some languages will work in all implementations of that language.

The NaNs are not undefined any more than C's "null pointer" is undefined. They are specifically defined, and can

undergo certain restricted but for the most part completely well-defined operations.

((Some history. Far less useful "indefinite" values were available on CCD6000 series and old Cray machines; reserved operands were created by DEC for their VAX; these would trap whenever touched. Absent reliable semantics for comparisons, these were fairly useless. Cray put these in because of the inconveneinece in catching pipelined results when produced; the results could be caught later: an attempt to compute with them causing a fatal error. DEC VAX reserved operand traps could not be disabled; even a move would trap, and produce a very slow result (VAX 8600 could handle only 40,000 such traps a second.)

Recounting the history of NaNs themselves, W. Kahan says that absent an agreement on how to deal with over/underflow, a compromise was reached. (Regarding overflow, the debate was: should it become infinite, or "larger than any representable number but finite"; and regarding underflow, should it be zero, or "smaller than any non-zero number but not zero"?) In any case, the compromise led to signalling and quiet NaNs. In quiet NaNs, the leading significant bit in the fraction part is 0; for signalling NaNs it is 1 (Though some implementors may have incorrectly reversed this convention). Signalling NaNs trap unless you mask off this trap; if you provide no trap handler, the system raises an invalid flag and generates a quiet Nan, and continues. (caution: if an implementor has the signalling/quiet bit reversed, converting a 1 to a 0 might make the fraction part all 0s, which may make it look like infinity.)

We hope that in many circumstances it may be possible to implement these routines once for all scientific programming languages on a given platform (hardware and software support system), so that routines for C, Fortran, and related styles of languages can all be handled at the same time.

The current Java [2] specification, in its quest for portability appears to make it impossible to use NaN in this way. The only exception-handling mechanism forces the completion of all expressions, statements, methods, constructor invocations, or initializations that have begun but not completed in the current thread. This unwinds until a handler is found, but by that time the opportunity to return a modified NaN as a result has long since passed.

4 Is this simplification acceptable?

It is natural to wonder how this could be useful, with so much loss of information. Let us look at some examples where this loss seems acceptable.

4.1 Reading uninitialized data

All arrays that are initially undefined may be filled with NaNs. An array `A` can be filled with all elements set to K_1, in which case any output that has K_1 in it has been computed using an uninitialized element of `A`. Since we have so many of these NaNs, we can consider using a different NaN for each different array, or more expansively, we can use a different NaN *for each element in each array.* This would allow us to attribute faults to particular array entries. (If we have single-precision floats, we may run out of different NaNs. It is unlikely we would do so for doubles.)

4.2 Uncertain data

In experimental data sets, it may be useful to put in estimated numbers for missing or uncertain data and run some analysis. One then may wonder which outputs are independent of that uncertain data. Replace the data with tagged NaNs and re-run the computation. You can now see which results depended on one or more of the uncertain values. This might be appropriate for data arriving from a remote weather station with unreliable communication or other problems.

4.3 Isolating regions

In some data sets and with certain kind of processing, it may be appropriate to tag periodic elements, borders, rows or columns with NaNs. This may be especially helpful in debugging: it will be possible to see if an algorithm has accessed some forbidden zone in a mesh.

Consider the recording paper tape or drum used in certain instruments such as seismographs, or atmospheric pressure recording stations. Some of these instruments inject a timing signal such as a blip every 60 seconds, to confirm that the mechanism is working, and to place time-frames around the real data being recorded. The broadcast television signal has similar characteristics: there is a non-picture gap in transmission during the re-scan phase that can be used to synchronize receivers, send data, etc. Such "out-of-range" data can be implemented in masses of floating-point data more generally by NaNs.

Here is another, more elaborate, 2-D example. Consider a kind of 3-dimensional conformal mapping of ϕ from one surface $z = f(x, y)$, defined as a collection of points x, y, z to a similar space, $w = g(u, v)$. It may be useful in understanding ϕ to see how it maps various axes, grids or perhaps the unit circle, from one space to another. To do so, we inject into the definition of z selected NaNs along these lines so that (for example) if $x = 0$, z is the NaN Red. That is, the surface is the same as before, but $f(0, y)$ is Red. The result map colors the image of the y-axis Red. We can use other colors for other axes.

Another example that is perhaps more concrete: imagine a bit-map of the United States, where each data point is a floating-point number representing the altitude above sea level of the corresponding geographical location. Replace all points on (or near) state boundaries by Red NaNs. Compute a new map, which displays the average daily exposure to radiation of a person at a given location on the map. This might be a function of altitude, distance from the equator, proximity to Three-Mile-Island or Hanford, etc. This new map would still have the state borders in Red.

Take the same original data and draw a map of the country under different map projections (mercator, polar). You have the altitude map and state borders.

Of course not all data has this kind of separable component quality. If you were to try to find the average altitude over areas marked off by latitude and longitude, then you would either compute arithmetically with the border data, propagating NaNs any time you touched a border, or have to make special arrangements in your code to to detect and avoid the NaNs. While this latter approach could be very useful, it is our intention to promote the use of these techniques for dealing with black-box programs. In such cases one does not have the freedom to insert checks for NaNs. Under the circumstances, some or all of the output data would be smeared over with Red.

4.4 Running programs backward

We do not propose that this be can be done with any generality: it is clearly impossible to find the inverse of an arbi-

trary program. We we imagine as a basis for a program some "pure functional" computational scheme, with a repeatable "stateless" execution, we can consider developing an interactive system based on the isolating region idea above. First one isolates of the inputs $(x, y, \cdots)$ affecting a given output $f(x, y, \cdots)$. Then by holding all but one of the inputs, say x constant, we could graph or otherwise deduce the dependency of $f(x, y, \cdots)$ on x.

4.5 Locating failure-inducing code retrospectively

For the technique proposed here we generally need to use some extra storage, but perhaps not much. In return, the value of the information provided may be substantial.

Suppose that your program divides 0.0 by 0.0 in the course of some (presumably erroneous) computation. In an interactive system one can halt the program, look at the program counter, check the operands, and probe the environment. If one cannot afford such an interaction, which in general requires either halting the process being probed or else preservation of substantial amount of information to examine offline in some simulated environment. As an inexpensive alternative (mostly in hardware), one can provide a NaN as the result. Given a sufficiently flexible implementation of exception handling, it may be the case that a software trap will be exercised in which the NaN generated can encode the program counter, or other appropriate information sufficient to identify the instruction or statement and procedure in which the violation occurred. It may be that this information is too bulky to store immediately in the fraction part of the NaN, especially for a single-precision datum. In that case we can store this information elsewhere, most plausibly in a table indexed by the fraction. Indeed, one could store substantial information in this way for all errant operations so that very little data would be lost. For example, if the exception be prompted by the combination of one or more previously-generated NaNs, the encoded data could include information like "NaN4 is the double-precision sum of NaN3 and 47 which were added on source-code line 12 of function f." In the limit, one could implement a virtual computer algebra system that would store whole computation sequences pertaining to the lineage of each NaN result datum. (Like a progressive nursery rhyme it may take some work to figure out what really happened... "This is the dog that chased the cat that caught the rat that ate the cheese that lived in the house that Jack built.")

5 A Simple-minded Implementation

To illustrate these concepts, we first show how to use a computer algebra system, Mathematica[9] to implement NaN-like operations by software. The principle is this: we make sure that any time an expression `K[i]` is combined with something else arithmetically, the expression becomes that `K[i]`. These 4 lines provide the facility for the four operators `+, *, ^`.

```
K[i_]+_^:=K[i]
K[i_]_^:=K[i]
K[i_]^_^:=K[i]
_^K[i_]^:=K[i]
```

To be more complete, we would have to add Mathematica rules for all of the other thousands of operators, `Sin`, etc., as well as patch a few boundary conditions like $K_i{}^0 = 1$. If we wish to assert that in combining a function of several NaNs, say K_i and K_j the result is $K_{\min(i,j)}$ then this would add yet more coding. As might be expected, this suprasoftware approach is orders of magnitude slower than the sub-software it is simulating.

In the Mathematica model one can easy store more information by injecting symbols with more informative indexes. That is, instead of `K[437]` one can use `K["uninit Array A"]` or `K["Div by 0 in Main"]` One could also use rules that produce new `K[i]`, say by incrementing a global counter each time one is produced, and associate with each index a table entry that describes the activity that produced it. Thus

```
Sin[K[i_]]^:=
(gcount++;
 nantab[gcount]:=StringJoin["Sin[",
                            ToString[K[i]],
                                "]"];
            K[gcount])
```

causes the generation of a new NaN, say `K[10]` when `Sin[K[4]]` is encountered. It also sets `nantab[10]` to the string `Sin[K[4]]` in case you wish to look at it later.

Experimenting with linear algebra and simple routines shows the kind of propagation expected. Squaring a square matrix

$$\begin{array}{cccccc} 2 & 1 & 1 & 1 & 1 & 1 \\ 1 & 3 & 1 & 1 & 1 & 1 \\ 1 & 1 & 4 & 1 & 1 & 1 \\ 1 & 1 & K(4) & 5 & 1 & 1 \\ 1 & 1 & 1 & 1 & 6 & 1 \\ 1 & 1 & 1 & 1 & 1 & 7 \end{array}$$

with a `K[i]` in location (x, y) generally wipes out the corresponding result column and row with that NaN. In this case it results in

$$\begin{array}{cccccc} 9 & 9 & K(4) & 11 & 12 & 13 \\ 9 & 14 & K(4) & 12 & 13 & 14 \\ 10 & 11 & K(4) & 13 & 14 & 15 \\ K(4) & K(4) & K(4) & K(4) & K(4) & K(4) \\ 12 & 13 & K(4) & 15 & 41 & 17 \\ 13 & 14 & K(4) & 16 & 17 & 54 \end{array}$$

Squaring it again wipes out *all* the definite value elements. Those computations requiring global information to produce each element of the result (e.g. matrix inverse, Fourier transform)generally wipe out all elements immediately.

6 Interpreting NaNs

Seeing the results in a matrix above is probably an unsuitable display if there are more than a few entries of more than a few NaNs. A typical Fortran formatted printout should provide sufficient room to note the presence of a NaN, but might lose any detailed information about its contents. To be useful, the output routines of such extended programs should be made aware of the presence of this information. As is shown in the example to follow, either interactive querying of the environment is a reasonable way out, but in a pure batch situation, one must plan for the consequences of additional information on output. This is a weakness in that it requires adaptation of programs that would otherwise produce the rigidly formatted results typical of Fortran tables of numerical data.

Often a well-designed application will have computational and input/output components well-separated. In these cases the output routines may need reformulation to be flexible

enough to display NaNs and related diagnostics, but the computational component need not be changed.

How might one alter the output to make good use of the information of NaNs? In the Mathematica example above, the `nantab` array was used to show how different NaNs were produced by, for example, the `Sin` program. In the case of *interactive* data analysis, the presence of NaNs that refer back to programs or common data areas can be traced through `nantab`-like "symbol table" information to relate the results back, not only to procedure names (like `Sin`), but perhaps statement numbers, etc. from the originating computation. In the case of batch or "offline" computing, or even interactive computing that plows on ahead and produces massive results, we expect that the retrospective diagnostic methods would need extra saved data in addition to the traditional result of the computation.

The kind of symbolic diagnosis where ordinary, but symbolic data is treated as exceptional and is therefore represented by `nantab` entries is appropriate where there are only a few results. The example of the last section illustrates this. By contrast, how can we deal with applications where scientific visualization uses plotting of large-scale data. How can one show the NaNs?

This requires some planning, given the need to compress information to fit into a plot. For example all NaNs could be treated using a distinct color or set of colors, or other visual clues. One way might be to tie their apparent values to particular normally-unachievable values so they would appear as special contour lines in a display.

If we are willing to write special analysis programs to use NaNs creatively, we can do yet other tasks in a novel way. To continue with our earlier example, consider experimentation with a digitized version of a weather station tape where we have barometric readings every 10 seconds or so, and a clocking blip every 5 minutes that goes up to 100 and down to 0 in one reading. Perhaps there is also a special synchronization pulse every 24 hours or 7 days. We may have unreliable instruments or transmissions so that we miss some readings. We can write a program that periodically resynchronizes if necessary, inserts guesses for missing data and perhaps even inserts guesses for missing synchronization signals. This could be based on the NaNs introduced, but would have to take explicit action for them.

Another interesting application[2], is an interactive statistical package which allows the use of missing data. Only if a summary statistic of some sort requires the use of missing data is it brought to the user's attention. At that point it is possible to insert the data if it has since become available, or if some special value or average can be used instead. The `nantable` in this application must store what amounts to a procedure to obtain the result, once the missing data is supplied.

7 Fitting into Systems?

In some languages (e.g. our prototype system in Mathematica, or even in compiled Lisp), it is plausible to substitute slightly different operations for arithmetic, as indicated in a subsequent section on a sample implementation.

This substitution is inefficient, and it is simply unacceptable to replace all multiplications in a Fortran program with subroutine calls or even in-line code that checks to see if the result is a NaN, and if so, calls some subroutine. It must be possible to run the Fortran compiled program unchanged and unimpeded in terms of efficiency for normal operations. Therefore if we are to have an instrumented NaN-based facility, the code that we have inserted in-line must be removed, and all or nearly all of our code moved into trapping software. This can indeed be done on many platforms, although we have encountered Fortran systems that seem to disable the appropriate trapping regardless of any expressed compile-time or run-time settings.

Furthermore, some vendors may be not actually abide by the requirements of IEEE 754 compatibility[3], and in some cases the software systems all but eliminate the possibility of using the features.

The standard *requires* that any arithmetic involving a NaN produce a NaN, and that all comparisons of a NaN (even to see if a NanN is arithmetically equal to itself) must return false.

This allows a minimal kind of tracking: keeping only data within the fraction part of the NaN. This does not give us the facility to place entries into some auxiliary table at the time the NaN is created. The minimal requirement on the black-box program under test is solely that it take floating-point data as input, and produce floating-point data as output; the formatted output must provide enough data to understand the NaN. Seeing **NaN** will not do.

We hope that the program can return the NaN in a framework in which the meaning can be looked up in a table or interrogated in some way.

Thus our test frame looks like this:

We assume that we have a Fortran, C, (etc) module `F` that typically is handed some collection of (perhaps large) arrays of initialized input and perhaps (uninitialized) "output" arrays. The input is set up, and salted as appropriate for diagnostic purposes with NaNs.

The module `F` is invoked. Rather than merely printing the answer, the returned results are scanned for NaNs. In the presence of NaNs, a diagnostic routine is run that (in the case of a batch run) prints out as much information as can be derived from the data at hand. In an interactive environment, a kind of debugging frame can be set up.

Initially we assume that F is an entirely conventionally compiled Fortran (etc) program.

While there are conditions under which F may fail (for example, attempting some iterative convergence that is doomed because it now involves NaNs), we expect that for a large class of computations, it may exhibit a useful pattern of propagated specific NaNs in the output. These can be examined by a suitably equipped programming environment.

We are continuing to experiment with a test environment written in Lisp that appears adequate for this purpose (using Sun Sparc, and as an alternative, HP-PA-RISC architectures), for setting up calling and diagnosing results. The principal requirements of access to NaNs in an interactive framework could be supported in other language frameworks, so long as the can effectively communicate data across language barriers.

8 Implementation Issues

In this longish sections we become more specific, and show how to use just a bit more hardware. By contrast with our Mathematica programs above, these should run faster, but not nearly at full machine speed.

[2] brought to our attention by W. Kahan.

[3] if they claim "compatible format" that may mean "but not compatible operations"

If we seek to run at much closer to machine speed we can use trap-handling mechanisms. Unfortunately these are not standardized, and thus must be dealt with on a platform-specific basis. We treat this briefly in a subsequent subsection. Since not all productions of NaNs cause a trap, some operations will not result in a newly-indexed NaN. In particular, operations in which one computes with a NaN operand to get a NaN would not ordinarily trap. This may in fact be preferable to the more space- and time- consuming design immediately below.

Here is a design we implemented in Lisp with relative ease, and with a modicum of efficiency. The merit of Lisp is that it allows easy access to the overloading or modification even of built-in functions. Other languages with sufficiently capable macro or pre-pass compiler phases could probably be used.

First we summarize our raw material: the IEEE-754 standard makes available a number of special representations within the usual sign + fraction + exponent floating-point. Each format (single, double, extended) has signed infinities, signed zeros, and a collection of Not-a-Numbers (NaNs). The NaNs have a reserved exponent, but the fraction part is mostly free for encoding other information.

We propose to create, in the fraction part of a NaN an encoding via an index into a table, an indication of its origin. For example one might index into a table of strings where an entry might say "This NaN was formed by the division of 0.0 by 0.0 from within program F." or it might refer indirectly to previous NaN's by their index numbers: "This NaN was formed by the addition of NaN23 to 3.14 from within program G with arguments 0.0, 2.718, 3.14." One could also look up NaN23 for more information. (The production of NaNs by NaNs may not cause a trap, so if we wish to record information at this level we pay extra.)

Two components were needed for this to be implemented in our prototype Lisp system. First the IEEE float information and control must be made available (a package we called `:ieee-float`) and then a system for creating and printing NaNs `:nan-pack`.

Here we use a conceptually simple approach: we use "wrapped" or advised functions for the basic floating-point arithmetic operations (add, subtract, multiply, divide) as well as (real) square-root. Each of these wrappers checks its result for being a NaN, and only in that case calls a routine to record the information concerning the circumstances of arguments, location of the call, etc. in the fraction-part of the NaN.

8.1 Low-Level Design

We briefly review the design of the `:ieee-float` package in somewhat greater generality than strictly necessary for our purposes here, just to indicate its scope. We do not, for example make use of the control of rounding modes in the retrospective diagnosis arena.

This package (`:ieee`) makes the status and mode flags of the IEEE 754 standard for binary floating-point arithmetic available. This does not preclude a language from providing a higher-level access for specific architectures[4].

An integer-valued variable `exceptions` is maintained as a vector of bits to represent positions in the system `exceptions` word: For the Sun SPARC processor, these are numbered 0 through 4 for `inexact, division, underflow, overflow, invalid`. These *flag* bits are meant to be set as a consequence of hardware arithmetic operations, and read or cleared by the the users' programs. It is possible for the users' programs to set these flags as well, to simulate exceptions.

There are also *modes* that are meant to be set by the users' programs, to influence the arithmetic processing. These are not changed by hardware arithmetic.

The modes for IEEE-754 consist of `direction` and `precision` to influence the current rounding preferences. The direction of rounding may be chosen from `nearest, tozero, negative, positive` with mode values 0 through 3. The precision of rounding (for those machines with extended-precision registers performing all of the arithmetic regardless of intended target precision) may be chosen from `single extended, double` with mode values 0 through 3. Each of these names in the `:ieee` package is defined as a constant of the appropriate value. Not all machines have a model of operation where the rounding-precision mode matters, in which case these settings are ignored[5].

The function `(swapEX x)` sets the system and Lisp-accessible variable `exceptions` to the value of `x` and returns the previous value of that system register.

Two functions based on `swapEX` are also provided:

The function `(getEX)` stores into the Lisp-accessible variable `exceptions` the current exceptions, and also returns this integer as its value. As a side effect it clears the system exceptions register.

The function `(clearEX)` sets the system register and Lisp-accessible variable `exceptions` to 0 (i.e. no exceptions).

The function `(swapRD x)` sets the system and Lisp-accessible variable (rounding) `direction` to the value of `x` and returns the previous value of that system register.

The function `(swapRP x)` sets the system and Lisp-accessible variable (rounding) `precision` to the value of `x` and returns the previous value of that system register.

Other functions C library to enable the exceptions are also provided: `fpgetmask`, `fpsetmask`.

To demonstrate a use of these functions, we define `tryfp` to evaluate a single Lisp symbolic expression x and return two values: The result of evaluating x, and a list of the names of flags that were set.

An implementation note: we use `ignore-errors` below, but all we really want to do is mask off any software-created Lisp signals (i.e. "division-by-zero") that might abort or put us in a debug loop. These signals are NOT in general produced by the floating-point arithmetic, but by the cautious programs in the Lisp interpreter that check for divisors being zero. That's why we use the "raw" division instructions from speed-optimized compiling in the definitions of `ieee::single-/` and `ieee::double-/` below.

```
(defmacro tryfp (x)
  "return 2 values: x evaluated,
   and flags that were set"
  `(let((oldexceptions(swapEX 0))
        ; save old exceptions
     (values (ignore-errors ,x)
               ;evaluate x
               ; lisp should ignore excpts
             (analyzefp(swapEX oldexceptions))))))
```

[4] As is the case with CMU-CL on SPARC.

[5] Machine types which ignore rounding-precision include SPARC, MIPS, DEC Alpha. Machines that use this information include Intel 80X87, 486, Pentium (but not i860), Motorola 68881/2, 68040,68050 (but not 88100).

In this function we use `analyzefp` to convert the returned `exceptions` integer into a list of exception names. The mapping we use below is particular to the SPARC architecture. The 80x86 (for example) is different.

```
(defun analyzefp(n)
  "convert exception integer to a list of atoms"
  (declare (fixnum n)) ; n<=31 in usage
  (let (ilist
        (tlist '(inexact division underflow
                 overflow invalid domain)))
    ;; for SPARC these are the flag meanings are 0..5
    ;; as given in tlist.
    (mapc #'(lambda (h)
              (when (oddp n)
                (setf ilist (cons h ilist)))
              (setf n (ash n -1)))
             tlist)
        ilist))
```

The next two programs must be compiled with minimal checking to have the right side-effects on the exceptions register. In particular, the compiled form must invoke a single (or double) division without first checking for division by zero: the "normal" division in Allegro does not set the zero-divide flag because, in the hope of avoiding traps, it checks in the software before dividing!

```
(defun single-/ (x y)
  (declare (optimize speed (safety 0))
           (single-float x y))
  (/ x y))

(defun double-/ (x y)
  (declare (optimize speed (safety 0))
           (double-float x y))
  (/ x y))
```

8.2 A more Lisp-ish interface to the IEEE-float flags

Additional programs provide these facilities:

To inquire what the current rounding direction is: type `(ieee::direction)`. For SPARC, the result is 0=nearest, 1=tozero, 2=positive, 3=negative.

To change the rounding direction, you can use `setf`. For example, changing the rounding to positive can be done by typing `(setf (ieee::direction) 2)` or equivalently, `(setf (ieee::direction) ieee::positive)`

Similarly for `(ieee::precision)` the current rounding precision.

The exception flags can be examined or set in an analogous fashion: You can determine the condition of the underflow flag by `(ieee::underflow)` which returns `t` or `nil`. You can change the underflow bit by `(setf (ieee::underflow) nil)` to clears it and `(setf (ieee::underflow) t)` to simulate an underflow.

You can look at the vector of exception flags by `(ieee::getEX)` or use `(ieee::analyzfp (ieee::getEX))` which tells you which are set "in English."

8.3 Uninitialized Data

As mentioned earlier it is possible to "initialize" uninitialized arrays of floating-point numbers to NaNs. While this can be somewhat tedious if we store unique strings for each array location, its value as a debugging tool may be substantial. A simple function `(make-uninitialized-array n "Name")` creates a new array of length n, fills it with a collection of single-float NaNs, each of which "points to" the information that it is an uninitialized datum. In particular it provides the data as to which element it is by storing `uninitialized Name[i]`. A far less expensive ($O(1)$ rather than $O(n)$ space) trick is to initialize all entries to exactly the same datum: a NaN that indicates `uninitialized element of Name`. This is not as informative, of course.

8.4 Comments on Other Approaches: Trap Handling

A more efficient solution than wrapping is to use software trap-handlers in systems, at least in systems where trapping is supported. Trapping is not *required* for IEEE compliance. If trap-handlers are used, normal arithmetic would run at full speed.

The down-side to this is we have to write the trap-handlers, typically requiring special attention for each different platform. It may also not provide a uniform level of detailed attention.

It's worth noting that the SUN operating system (and other UNIX-like operating systems) may each have their own idea about handling errors. For example, the `libm` math library traditionally (and in SVID3) allows users to define their own return mechanism via a program (`matherr`). The documentation advises you to load a replacement for this program if you have special needs. `Matherr` is given the arguments, but insufficient information generally to reconstruct the program counter or the source-code location of the problem.

On a Sun SPARC, accessing the library `log` routine and invoking it on 0.0 with the default `matherr` results in several actions not all of which you may desire:

- a message: `log: SING error` is printed on `error-io` stream.
- The `DIVISION` flag is set.
- an answer is returned: the double-precision NaN for negative infinity.

In general, X/OPEN (XPG3) no longer sanctions the use of the `matherr` interface and specifies its own results for out-of-range and other problems with the math library; ANSI/ISO-C Standard is similar but covers only a subset of XPG3. If you are using a UNIX system, current information should be available in your system documentation.

Aspects of exceptions and interrupt handling are described in detail in the Sun Solaris UNIX manual in pages titled `siginfo(5)`, signal(5), `sigaction(2)` and `fpgetround(3c)`. At the level of detail available here, nearly anything can be programmed so long as a hardware interrupt is available (it is not always), and the software does not insist on disabling it. A system for treating NaNs resembling in outward appearance the previously described Lisp `:nanpack` can be created from these routines, although the small and apparently irrelevant details to be surmounted grow: Lisp already has an interactive framework, as well as storage allocation as needed.

If the C situation is somewhat dicey in terms of machine independent results, this is not unusual: of the currently widespread languages, most do not specify completely their arithmetic exception treatment. In this regard, Java stands out both in leaving nothing to the discretion of the implementation, and also in making any implementation of our idea here via traps impossible. Java removes access to the needed environment from any exception-handling routine,

so production and return of a NaN is rendered impossible. Furthermore there is but one NaN of each floating-point precision. An extensive proposal to alter Java behavior related to support of the IEEE floating-point standard is provided in the Borneo [3] design. Even so, in retaining compatibility with the Java exception handling mechanism, Borneo forbids the use of our technique. C++ similarly makes inadequate provision for exceptions, although by using the underlying C library `signal` routines one can usually access the needed features if they are provided in the operating environment.

8.5 Multiprocessing problems

In languages including Java and the version of Lisp [6] we have been using, it is possible to have multiple lightweight processes or threads running, in which case it is possible that there will be several processes computing floating point results. Under these circumstances it is generally necessary for each of the lightweight processes to maintain its own copy of the hardware registers that pertain to the floating-point status and signal vectors. Systems ordinarily keep track of the floating-point data registers, status and signal vectors on a full process switch, but might not keep track on a thread basis.

One possible way of dealing with this is to have, at least for each process using any floating-point, a simple procedure which is invoked on process switching. A more elaborate version of `tryfp` which deliberately makes a separate process, preserving and restoring the exception flags is included in our prototyping `:ieee` code. The definition of this function can be examined to see how to treat these apparently global registers as we require – namely as local variables.

8.6 Lisp, Traps, and Error Handling

The treatment of conditions in Common Lisp [8] is elaborate and in the case of arithmetic errors, somewhat vague. It specifies that `arithmetic-error` is a subtype of `error`, yet there are no requirements in the language that any arithmetic operation must signal an error. (For example consider division by zero. Steele [8] (p. 296) says "It is generally accepted that it is an error". *But the proposed CL standard does not require that an error be signalled.*)

In any case, the occurrence of (say) a divide-by-zero in Lisp is not necessarily related to floating-point arithmetic —for example, it could be the creation of a ratio data type from two integers, the second of which is zero. Such an operation is not in any apparent way connected to any "floating-point" operation codes, registers, or operating-system exception handling.

In various Lisp implementations we've tried, floating-point exceptions seem to be either ignored, or to cause computations to abort unless caught by `ignore-errors`. The default seems to be a choice of the implementors. For those hardware and software systems in which trapping can be enabled, the condition handling structures in Common Lisp provide a very nice model for control, diagnosis, and recovery. (IEEE-754 does not, in fact, specify this level of control.) The Carnegie-Mellon Common Lisp design provides this kind of interface.

8.7 NanPack Operations

Each of the operations looks something like the definition of `mult` below. Note that `mult` and its relatives can be expanded in-line to eliminate some function-call overhead, but there is no convenient way of eliminating the check to see if the answer is a NaN[6].

```
(defun mult (a b)
  "nanpack version of 2-argument
    single-float multiply"
  (declare  (inline *)  (single-float a b))
  (let ((value (* a b))
    (if (NaN-p value)
        (make-NaN (nanrept2 a b "Mult"))
      value))))
```

The function `nanrept2` "NaN Report of two floating-point arguments" used by `mult` and the other binary floating-point operators constructs a string from the arguments a and b given to `mult`, as well as some information on the call stack as to which (user-defined) program called `mult`. The function `make-NaN` places this string in an array and places an index to its location in the NaN returned.

A more elaborate function that a user might wish to construct could use some of our other programs to also report on exception flags:

```
(defun users-fun-wrapper (a b)
  "nanpack version of users-fun of 2 args"
  (let*
     ;;first save old EX flags set them to 0
    ((save_EX (ieee::swapEX 0))
     ;; next, evaluate the users-fun
     ;;of 2 arguments
     (value (the single-float  (users-fun a b))))
     ;; then analyze new settings of EX flags,
     ;; restore old ones
  (new_EX (ieee::analyzefp
               (ieee::swapEX save_EX)))

  (if (NaN-p value)
        ;; if a NaN was produced, make an
        ;; explanation string...
        (make-NaN
         (concatenate 'string
           ;; tell about any exception flags if any
           (if new_EX
              (format nil "Exception~p ~{~a~^ ~} in "
                      (length new_EX) new_EX)
              "")
           ;; tell the usual things about the call
           ;; as well
           (nanrept2 a b "user-fun")))

      ;; otherwise just return the normal value
      value)))
```

Note that in this function we concatenate (to create a longer string), the information on exceptions to the front of the string consisting of the normal information that is returned from `nanrept2`. This combined string is then passed into `make-NaN`.

If the diagnostic information recorded in the Nan reported that `user-fun-wrapper` was the caller on record for `user-fun` on the stack, we would be rather disappointed –

[6] We cannot *require* hardware trapping from the underlying system since this is not required by IEEE-754. Even though almost all systems can be set to trap on signalling NaNs, some systems extract a severe performance penalty, and operations on non-signalling NaNs will not ordinarily trap, so information will only be stored on the creation of a NaN.

we knew that. We want the information to include the user program one level higher. We "hide" `user-fun-wrapper` by invoking `(dont-report-on-stack 'user-fun-wrapper)`.

Then if `user-fun-wrapper` was itself invoked by `(setf r (foo 1 2 3))` where we have defined `foo` by

```
(defun foo (a b c d) (user-fun-wrapper 1.2 3.4))
```

then a sample output might be the result `#.EXCL::*NAN-SINGLE*`. Then a call to diagnose that result by `(get-nan-info r)` would reveal some information in a format illustrated here:

```
#0006: Exceptions OVERFLOW INVALID DIVISION in
user-fun called with  1.2 and 3.4 called
from (foo 1 2 3 4)
```

9 An example

Here is a deliberately overly-simplistic secant-method root-finder that demonstrates the usefulness of retrospective diagnosis of NaNs.

```
(defvar find-root-debug t) ;print out debug info

(defun find-root (func x0 x1 tolerance &optional max)
  "Starting from iterates x0 and x1
   find f(x[i])=0 by secant method.
   Return iterate x[n] such that
   |f(x[n])|< tolerance or n>max
   or f(x[n]) is a NaN."

  (do ((oldval (funcall func x0) val)
       (val (funcall func x1) (funcall func x1))
       (count 0 (1+ count)))
      ((or (and (numberp max) (>= count max))
           (nan-p val)
           (< (abs val) tolerance))
       ;;x1 is the final iterate, val is f(x1).
       (values x1 val))
    (let ((x2 (subtract x1
                (mult val
                    (divide (subtract x1 x0)
                            (subtract val oldval
                             ))))))
      ;; print info on the iteration
      (if find-root-debug
          (format t "X0: ~a, X1: ~a, f(X0): ~a,
                   f(X1): ~a  ->  X2: ~a~%"
                  x0 x1 oldval val x2))

      (setf x0 x1 x1 x2))))

(defun test2 (x)
  "sqrt((x-1)*(x-4)) + (x-7)*(x-7)
   is complex for 1<x<4"
  (add (real-sqrt (mult (subtract x 1.0)
                        (subtract x 4.0)))
       (mult (subtract x 7.0) (subtract x 7.0))))

;; Example

USER(38): (setf a (multiple-value-bind (trash answer)
              (find-root 'test2 6.0 6.5 0.0001)
              answer))

;; somewhat reformatted printout...

X0:  6.000, X1:  6.500, f(X0):  4.162,
    f(X1):  3.958  ->  X2: 16.193
X0:  6.500, X1: 16.193, f(X0):  3.958,
    f(X1): 98.117  ->  X2:  6.093
X0: 16.193, X1:  6.093, f(X0): 98.117,
    f(X1):  4.088  ->  X2:  5.653
X0:  6.093, X1:  5.653, f(X0):  4.088,
    f(X1):  4.587  ->  X2:  9.689
X0:  5.653, X1:  9.689, f(X0):  4.587,
    f(X1): 14.258  ->  X2:  3.740
#.EXCL::*NAN-SINGLE*

USER(39): (get-nan-info a)

"#0006: add called with NaN#0005 and 10.629862
      from    (TEST2 3.739653)"

USER(40): (get-nan-info 5)

"#0005: real-sqrt called with -0.7132602
      from    (TEST2 3.739653)"
```

What has happened here? The secant method wandered out of the domain of validity of the function. A negative argument to the `real-sqrt` function produced a NaN. The secant program returned as soon as it found a NaN as one of its iterates. We examined the result, the value of variable `a`, NaN #0006. From that we found it was produced by the program `add` operating on NaN #0005, which was produced by the square-root program.

We are not billing this as a robust design for a root-finder, since we could easily switch to another method such as bisection until a good point was found.) The point here is to aid the programmer to see the origin of the NaN in a useful and timely fashion.

10 Conclusions

These ideas can be refined further in computer algebra systems, directly in Common Lisp, or in other languages. Indeed, in most systems implementation in C may be most straightforward.

Our hope is that this short paper will prompt such refinement, allowing better access to the hardware capabilities from such systems. We hope, also, that we prompt designers or implementors of other languages to take advantage of these capabilities.

We intend to look for significant applications in which this kind of analysis can lead to a better debugging and code-explanation environment. The kinds we expect to have the simplest payoff are those with

1. large amounts of input encoding sensor data, some of which is of uncertain accuracy.
2. only modest levels of computation, and that of a primarily local nature.
3. typically graphical output in which the presence of NaNs can be easily detected.

The debugging or diagnosis of such codes would be achieved by re-running them with the input, seeded with inserted NaNs. Additional diagnostics might be produced by putting NaNs in previously uninitialized data arrays.

All the code for these experiments can be obtained on request from the author (contact `fateman@cs.berkeley.edu`).

11 Acknowledgments

We thank Steven Haflich of Franz Inc. for providing help with details of Allegro Common Lisp, and with Prof. W. Kahan on retrospective diagnostics and the IEEE-754 standard. Parts of `:nanpack` were initially written by Steven Lumetta as a project in CS283, Fall 1992, at the University of California at Berkeley.

References

[1] L. A. Clarke. "A system to generate test data and symbolically execute programs," *IEEE Trans. on Softw. Eng.* SE-2 no 2, September 1976, 215—222.

[2] Gosling, James, Bill Joy and Guy Steele. *The Java Language Specification* Addison-Wesley, 1996.

[3] Joseph D. Darcy, Cedric Krumbein, Howard Robinson, Robert Yung, "Borneo: Adding IEEE 754 floating-point support to Java", (unpublished, June, 1997).

[4] Dillon, Laura K. "Using symbolic execution for verification of Ada tasking programs." ACM Transactions on Programming Languages & Systems v12, n4 (Oct, 1990):643 (27 pages).

[5] Favaro, John M. "A FORTRAN symbolic executor based on MACSYMA," 1979 Macsyma Users Conference, MIT Lab. for Computer Science, 1979, 98–120.

[6] Franz Inc. "Allegro Common Lisp 4.3" Berkeley, CA.

[7] IEEE Computer Society Microprocessor Standards Committee Task P754, a standard for binary floating-point arithmetic, (see, for example, draft 8.0, *Computer 14*, 3 Mar. 1981, 52-63)

[8] Guy L. Steele, Jr. *Common Lisp the Language, 2nd ed.*, Digital Press, 1990.

[9] Wolfram, Stephen. *Mathematica*, 2nd ed. Addison Wesley, 1991.