Symbolic Execution and NaNs: Diagnostic Tools for Tracking Scientific Computation

Richard Fateman

(Based in part on *ISSAC-99 Poster Session)*

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.

Symbolic Execution/Proofs

The idea is simple: run your favorite program **f** with a symbolic input. If **f(x)** computes $\mathbf{x^2}$ then there is a good reason to believe that we have in effect also proved that **f(3)=9** etc.

Symbolic execution runs into trouble with more complicated calculations. That is, the size of expressions involving symbolic parameters can grow quite rapidly. Here is an example: Consider Newton's method to find the square root of *v*. Each iteration is

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

Three iterations starting with the symbolic root *r* gives us this rather complicated result.

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

In fact, this result is actually a pretty good estimate! Not easy to prove though. If *r* is between 0.5 and 4, and *v* is 2, then x_3 gives at least 3 correct decimals of ⍰2.

We can't economically keep "all information symbolically" so we propose an alternative to help debug partially symbolically

Partial evaluation using floats + NaNs (Not a Numbers in IEEE 754 fp standard). *There are 2^{51} different NaNs available in double precision. Call them k_1, k_2 ….*

1. Use NaNs for "symbols"
2. Use NaNs for "uninitialized"
3. Use NaNs for "uncertain" or intervals
4. Use NaNs for exceptional values, bigfloats ...

Initially Creating NaNs

“Uninitialized” arrays: We could use a single “uninitialized” NaN value or a different one for each array, or even each element. Each such NaN can be implemented as a pointer into a hash-table with a tag declaring “uninitialized Array A[5,6].”

Other initial NaNs might exist be created by calling routines to create tagged uncertain values, stand-ins for symbolic values, or other concepts with associated arithmetic (red?).

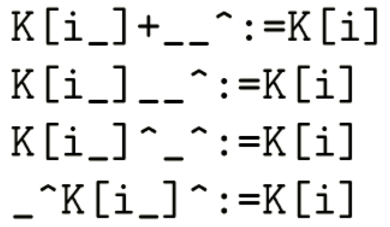
We can create NaNs by division of ordinary values by zero. We might tag such a result by “Division by zero in subroutine F”

NaNs are then propagated by operations on them...

Operations on NaNs

Every operation *n*□*m* on one or more NaNs results in a NaN, either one of the NaN inputs, or a new NaN encoding a result of *n*□*m*. Programmable or default operations possible.

Here is a simpleminded Mathematica version of NaN contagion: any operation (+, *,^) on K[i] returns K[i]:



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

Example of Computing with NaNs

Square this matrix

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

Numerically(!) we get this result

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

Fitting into Systems

It is generally unacceptable to replace all arithmetic operations with calls to subroutines as in this demo with Mathematica. Running at full speed on ordinary values + additional operations on NaNs should be done by exception-handling (trapping) software.

If we wish to have a retrospective diagnostic tracking of all NaNs then we can store (until we run out of patience or memory) all results in a hash-table, such as “This NaN_{534} was formed by the division of NaN_{230} by 0.0 at source line 1031 in subroutine F”. Of course some of the information, must be extracted from symbolic debugger information outside our program.

Fitting into Systems (continued, 1)

Some programming languages/compilers do not support NaNs properly. E.g.

```
If (a>=b) then f else g
```

may be erroneously transformed into

```
If (b>a) then g else f.
```

Comparisons of NaNs always result in **`False`**. IEEE conforming hardware is commonly disabled by today's languages (e.g. initial Java approach to FP).

Appropriate access to exceptions, trapping routines, NaNs etc is required at runtime. (cf. Borneo variant of Java)

Fitting into Systems (continued, 2)

Retrospective Diagnosis:

At the end of a computer run it must be possible to examine the results in the context of the hash-table, and so a formatted output that merely shows "**NAN**" is unacceptable.

This requirement is easily met in a "residential" computational system like the typical interactive Lisp implementation, and this is one good reason for our choice of prototype program framework.

A Lisp Version

A detailed design and description of an implementation in Allegro Common Lisp is given in the full paper. This system is based in part on access to trapping routines, and so a portion of the implementation is machine and OS specific.

Thanks to graduate student Steven Lummetta, Steven Haflich of Franz Inc, and Prof. W. Kahan. (This implementation was done in 1992 on Sun Microsystems Sparc OS 4.3, ported in part to HP/Precision Architecture.)

Details at

`http://www.cs.berkeley.edu/~fateman/papers/retrodiag.pdf`