Trichotomy vs. NaN

Abstract:

Any two integers, or rational numbers, or real numbers x and y must satisfy just one of the three order predicates

$$x < y \text{ , or } x = y \text{ , or } x > y \text{ .}$$

This is *Trichotomy*. It characterizes the totally ordered nature of those number systems. And it is one of the properties of numbers that cannot be preserved in its entirety on real computers. What supplants Trichotomy, and why, is explained in these notes, which also suggest two notations by which programming languages may accommodate the loss of Trichotomy gracefully. Which notation do you prefer?

What goes wrong with Real Numbers?

Obviously real computers are restricted to a finite subset of real numbers. This restriction gives rise to *Overflow*, *Underflow* and *Roundoff* on computers. Computers are restricted in another less obvious way; they cannot stop to think and ask questions that their programmers had not anticipated. Otherwise computers might be stopping thousands of times per second. Instead computers must continue computing under programs' control. Every operation must either produce a result and continue or else, in exceptional circumstances, transfer control. But what if the exception was unanticipated and no suitable site had been prepared to accept the transfer? In 1996 this brought down the *Ariane V*, a half-billion dollar rocket and its satellite payload.

Loss of control is a punishment too risky to impose indiscriminately upon all programs that encounter unanticipated exceptions. Programmers should be allowed the option of deferring judgment; programs should be able to detect the occurrence of exceptional events subsequently at times and places of the programmer's choosing. Otherwise programmers would be obliged to program "defensively" by inserting tests and branches to preclude every untoward eventuality regardless of how very rarely they can happen, if they can happen at all. Besides slowing the program down, such tests and branches slow down the programmer by obscuring a program's intent and exacerbating its vulnerability to error.

Thus we infer that a computer's number system has to be a *Completed* system, closed under all algebraic operations. Exceptional operations like 3/0 or 0/0 have to be allowed to produce predefined results, including a *Signal* as a side-effect, and to continue along the expected path of control, unless the programmer asks for a *Trap*. The signal can be a *Flag*, something like a global variable that is changed from its null value only if an exception of appropriate type occurs, and can subsequently be sensed, saved, restored or reset by the program.

Infinity

Two kinds of exceptions will be discussed in these notes. One is the *Divide-by-Zero* exception, though it would be better called

" Exact Creation of Infinity from Finite Operands."

Examples of this exception are 3/0 , log(0) and arctanh(1) . Its predefined result is always a signed infinity, either $+\infty$ or $-\infty$, and its signal is the raising of the Divide-by-Zero Flag unless the Divide-by-Zero Trap is enabled.

(*Overflow* is another way to generate $\pm\infty$ but it is an approximation, not exact, and raises the Overflow Flag instead. The distinction is important for an expression like $x/(y\cdot z)$ when all of x, y and z are so huge that $y\cdot z$ overflows and forces $x/(y\cdot z)$ to 0 although $(x/y)/z = 0.125$. Programs must be permitted to detect this later.)

We have inferred that programmers' needs are served best if the computer's real number system is Completed, augmented by $\pm\infty$, even though this augmentation is a mixed blessing.

To an augmented real number system, exceptional values like ∞ look like potato chips: one or two are never enough. To keep $1/(1/x) = x$ at least approximately for every real number x including $\pm\infty$, the augmented system has to distinguish $+0 := 1/(+\infty)$ from $-0 := 1/(-\infty)$. These two zeros are equal numerically ($+0 == -0$) but distinguishable in just two ways:

CopySign(1, ± 0) = ± 1 respectively, and
$1/(\pm 0)$ produces $\pm\infty$ respectively, just as $1/(\pm\infty) = \pm 0$.

In other words, the sign of zero is detectable only by performing one of the two discontinuous operations CopySign(y, x) or 1/x when x is zero. This allows applications programmers (but not compiler writers) to ignore the sign of zero unless they care about it. For all arithmetic operations the sign of a zero result is predictable from the following rules:

SignBit of a Product = Exclusive OR of operands' SignBits ;
SignBit of a Quotient = Exclusive OR of operands' SignBits ;
SignBit of $x - y$ is the same as that of $x + (-y)$ and of $-y + x$;
SignBit of $x + x$ is the same as that of x for all x ;
$x - x$ yields +0 for all finite x .

Only the last is arbitrary, and it changes if rounding is directed towards $\pm\infty$ instead of towards nearest. The other rules are just what might be expected. A consequence of these rules is that compilers must not "Optimize" expression $x \pm 0$ to x , nor $x-y$ to $-(y-x)$, nor $x\cdot 0$ to 0 , lest the uses of a signed zero be ruined. Those uses are valuable enough to deserve an extensive exposition of their own; this is not it.

NaN

Any augmented real number system must violate certain cancellation laws and identities. The first of them is familiar:

x/x is not 1 if x is 0 or ∞ ;
$x-x$ is not 0 if x is ∞ .
$x\cdot 0$ is not 0 if x is ∞ .

What predefined values can these exceptional expressions produce? *NaN* ("Not a Number") has been introduced to serve that purpose, and is the last augmentation necessary to complete the real number system for all algebraic operations. Besides 0/0 , ∞/∞ , $\infty-\infty$ and $\infty\cdot 0$, other real-valued operations, for instance $\sqrt{(-3)}$, log(–5) and arcsin(7) , produce NaNs too, but sparingly:

A NaN is worth creating only if any other finite or
infinite real value would cause worse confusion.

Unfortunately, the *Field* of real numbers cannot determine its completions uniquely. Smaller completions are augmented by NaN but no ∞, or by NaN plus just one unsigned ∞ and one unsigned 0 instead of two of each. Bigger completions can include the *Algebraically Closed* Field of *Complex* numbers, and the non-Field of intervals used in *Interval Arithmetic*. Early in the 1980s IEEE Standard 754 for Binary Floating-Point chose augmentation by ± 0, $\pm\infty$ and NaNs as a compromise between extreme parsimony and burdensome prodigality.

Note that algebraic operations with NaNs are NOT undefined. On the contrary, they follow a well-defined rule:

> If $f(x)$ is an algebraic expression that can take different values as x runs through all finite and infinite real values, then $f(\text{NaN})$ is NaN.
> (But $f(\text{NaN})$ must match $f(x)$ if this value is independent of x .)

And when an operation creates a NaN from finite or infinite operands, it must signal *Invalid Operation* and raise the Invalid Operation Flag unless the Invalid Operation Trap is enabled.

(The same signal or trap occurs when a *Signaling NaN* is acted upon arithmetically and turns into an ordinary silent NaN for lack of a trap handler. IEEE 754 requires conforming systems to support at least one NaN of each kind, and recommends that silent NaNs preserve at least partial information about their ancestry when they descend from NaNs. Thus, a silent NaN can reveal its point of creation; a signaling NaN can reveal which datum is not yet initialized. These debugging aids remain mostly unexploited for lack of compiler support.)

When an operation upon a NaN produces a NaN, no signal issues. Consequently NaNs propagate themselves silently through almost all arithmetic operations. If that is all they did, we could never get rid of them and they would be useless; we would be better off to quit computation at the first sign of a NaN.

How do NaNs go away?

Extended Definitions

Many a function f of real arguments can be extended in a natural way to accommodate $\pm\infty$ and NaN. Extensions that accommodate ∞ usually are accomplished by a limiting process; examples are $\exp(-\infty) = 0$ and $\ln(+\infty) = +\infty$, neither of which need signal. Extension by a limiting process can fail utterly, as in the example $\sin(\infty)$ which generates a NaN and an Invalid Operation signal. Mere discontinuity is not a good reason for extension to fail but calls instead for consensus upon a convention or at least tolerance of diverse extensions; for instance

$$\text{signum}(0) := 0, \quad \text{SIGN}(0) := 1 \quad \text{and} \quad \text{CopySign}(1, \pm 0) := \pm 1$$

exemplify tolerance. The power function's $0^0 := 1$ and consequent $\infty^0 := 1$ and $\text{NaN}^0 := 1$ exemplify conventions now followed widely if not universally. Still, troublesome cases remain.

Take $\max\{x, y, z, ...\}$ for example. Surely its value should not change when its arguments are permuted, but many an implementation violates this expectation when an argument is NaN. One way to avoid confusion is to define $\max\{\text{NaN}, y, z, ...\}$ to be NaN and issue an Invalid Operation signal. However, another way more useful for "windowing" is to ignore any NaN among the arguments of $\max\{...\}$ unless all of them are NaN, and return NaN with no signal in only this case. Ideally both ways of implementing $\max\{...\}$ should be available. Alas, no way preserves all relationships valid for finite real arguments; $\max\{x, y\} == (x+y + |x-y|)/2$ must be violated when both arguments are infinite or when either is $-\infty$ or NaN, as we'll see.

In general, for real valued functions f of real arguments the rule

" Create a NaN (and signal Invalid Operation) or quietly propagate
a NaN just when any other result would worsen confusion "

is a clear statement of intent but not always dispositive. And it gives no guidance at all for non-numeric or integer functions of real variables.

Order Predicates

A programmer who has anticipated the possible creation of NaNs can test for their existence in three ways. One way tests the Invalid Operation Flag; this is quicker than scanning arrays of intermediate results for NaNs. Another way invokes a predicate like isNaN(x) designed to return *True* if x is NaN and *False* otherwise. A third way, and the only way available to programs programmed in a language oblivious to NaNs , is to detect the way a NaN affects the order predicates. The predicate x == x , *True* for every finite and infinite x , must be *False* by definition when x is NaN . (Rashly optimizing compilers may spoil this.) And x != y matches !(x == y) for every x and y and is therefore *True* only if x or y is NaN . The four order predicates

$$x < y, \quad x \le y, \quad x \ge y \quad \text{and} \quad x > y$$

are all *False* if x or y is NaN . Thus, Trichotomy is violated by NaNs , but their behavior in order predicates is well-defined. Consequently a programmer can predict what NaNs will do in and to his program, and can then code steps to replace or ignore NaNs if they arise.

It was not always so. Things somewhat like NaNs have been built into the hardware of earlier computers. CDC 6000 and *Cyber* computers had "Infinities" and "Indefinites" in the 1960s. In the 1970s CRAYs still had "Indefinites"; and DEC PDP-11 and VAX computers had very similar "Reserved Operands." They were all printed or displayed as "????." But their roles in order predicates were undefined and therefore accidental; they were worse than useless to programmers. Real NaNs have many uses, but this is no place to enumerate them all.

Because programmers who had not anticipated that NaNs might arise could be ambushed by a violation of Trichotomy, the four order predicates

$$x < y, \quad x \le y, \quad x \ge y \quad \text{and} \quad x > y$$

must also signal Invalid Operation whenever x or y is NaN . A programmer who does take NaNs into account will find these signals to be nuisances. This is why non-signaling order predicates have to be provided too, all four of them silently *True* if x or y is NaN . Two notations have been proposed for them:

$x \not\ge y$	and $x <? y$	silently take the same value as	$!(x \ge y)$,
$x \not> y$	and $x \le? y$	" " " " " " " " " "	$!(x > y)$,
$x \not< y$	and $x \ge? y$	" " " " " " " " " "	$!(x < y)$, and
$x \not\le y$	and $x >? y$	" " " " " " " " " "	$!(x \le y)$.

The first column's notation is analogous to " != ". The second column uses " ? " to stand for an "Unordered" relation between NaN and everything (including itself) in the augmented real number system. But the symbol " ? " is used elsewhere in *Java*'s conditional expressions of the form " ?(L, t, f) " to abbreviate *Algol*''s expression " (if L then t else f)". The symbol " ! " is used elsewhere in programming languages only to negate predicates.

Which of the notations in the first two columns do you prefer? Why?