WIKIPEDIA

NaN

In computing, **NaN**, standing for **not a number**, is a numeric data type value representing an undefined or unrepresentable value, especially in floating-point calculations. Systematic use of NaNs was introduced by the IEEE 754 floating-point standard in 1985, along with the representation of other non-finite quantities like infinities.

Two separate kinds of NaNs are provided, termed *quiet NaNs* and *signaling NaNs*. Quiet NaNs are used to propagate errors resulting from invalid operations or values, whereas signaling NaNs can support advanced features such as mixing numerical and symbolic computation or other extensions to basic floating-point arithmetic. For example, 0/0 is undefined as a real number, and so represented by NaN; the square root of a negative number is imaginary, and thus not representable as a real floating-point number, and so is represented by NaN; and NaNs may be used to represent missing values in computations.[1][2]

Contents

Floating point

In floating-point calculations, NaN is not the same as infinity, although both are typically handled as special cases in floating-point representations of real numbers as well as in floating-point operations. An invalid operation is also not the same as an arithmetic overflow (which might return an infinity) or an arithmetic underflow (which would return the smallest normal number, a denormal number, or zero).

IEEE 754 NaNs are represented with the exponent field filled with ones (like infinity values), and some non-zero

number in the significand (to make them distinct from infinity values); this representation allows the definition of multiple distinct NaN values, depending on which bits are set in the significand, but also on the value of the leading sign bit (but applications are not required to provide distinct semantics for those distinct NaN values).

For example, a bit-wise IEEE floating-point standard single precision (32-bit) NaN would be: s111 1111 1xxx xxxx xxxx xxxx xxxx xxxx where *s* is the sign (most often ignored in applications) and the *x* sequence represents a non-zero number (the value zero encodes infinities). The first bit from *x* is used to determine the type of NaN: "quiet NaN" or "signaling NaN". The remaining bits encode a *payload* (most often ignored in applications).

Floating-point operations other than ordered comparisons normally propagate a quiet NaN (*qNaN*). Most floating-point operations on a signaling NaN (*sNaN*) signal the invalid operation exception, the default exception action is then the same as for qNaN operands and they produce a qNaN if producing a floating-point result.

The propagation of quiet NaNs through arithmetic operations allows errors to be detected at the end of a sequence of operations without extensive testing during intermediate stages. However, note that depending on the language and the function, NaNs can silently be removed in expressions that would give a constant result for all other floating-point values e.g. NaN^0, which may be defined as 1, so in general a later test for a set *invalid* flag is needed to detect all cases where NaNs are introduced [3] (see Function definition below for further details).

In section 6.2 of the current IEEE 754-2008 standard there are two anomalous functions (the *maxnum* and *minnum* functions that return the maximum of two operands that are expected to be numbers) that favor numbers — if just one of the operands is a NaN then the value of the other operand is returned. For the next revision of the IEEE 754 standard, it is planned to replace these functions as they are not associative (when a signaling NaN appears in an operand).[4]

Comparison with NaN

A comparison with a NaN always returns an *unordered result* even when comparing with itself. The comparison predicates are either signaling or non-signaling on quiet NaN operands; the signaling versions signal the invalid operation exception for such comparisons. The equality and inequality predicates are non-signaling so $x = x$ returning false can be used to test if x is a quiet NaN. The other standard comparison predicates are all signaling if they receive a NaN operand, the standard also provides non-signaling versions of these other predicates. The predicate *isNaN(x)* determines if a value is a NaN and never signals an exception, even if x is a signaling NaN.

Comparison between NaN and any floating-point value x (including NaN and ±∞)

Comparison	**NaN ≥ *x***	**NaN ≤ *x***	**NaN > *x***	**NaN < *x***	**NaN = *x***	**NaN ≠ *x***
Result	Always **False**	Always **False**	Always **False**	Always **False**	Always **False**	Always **True**

Operations generating NaN

There are three kinds of operations that can return NaN:[5]

- Operations with a NaN as at least one operand.
- Indeterminate forms:

- The divisions $(\pm 0) / (\pm 0)$ and $(\pm \infty) / (\pm \infty)$.
- The multiplications $(\pm 0) \times (\pm \infty)$ and $(\pm \infty) \times (\pm 0)$.
- The additions $(+\infty) + (-\infty)$, $(-\infty) + (+\infty)$ and equivalent subtractions $(+\infty) - (+\infty)$ and $(-\infty) - (-\infty)$.
- The standard has alternative functions for powers:
  - The standard `pow` function and the integer exponent `pown` function define 0^0, 1^∞, and ∞^0 as 1.
  - The `powr` function defines all three indeterminate forms as invalid operations and so returns NaN.
- Real operations with complex results, for example:
  - The square root of a negative number.
  - The logarithm of a negative number.
  - The inverse sine or cosine of a number that is less than -1 or greater than 1.

NaNs may also be explicitly assigned to variables, typically as a representation for missing values. Prior to the IEEE standard, programmers often used a special value (such as −99999999) to represent undefined or missing values, but there was no guarantee that they would be handled consistently or correctly.[1]

NaNs are not necessarily generated in all the above cases. If an operation can produce an exception condition and traps are not masked then the operation will cause a trap instead.[6] If an operand is a quiet NaN, and there isn't also a signaling NaN operand, then there is no exception condition and the result is a quiet NaN. Explicit assignments will not cause an exception even for signaling NaNs.

Quiet NaN

Quiet NaNs, or qNaNs, do not raise any additional exceptions as they propagate through most operations. The exceptions are where the NaN cannot simply be passed through unchanged to the output, such as in format conversions or certain comparison operations (which do not "expect" a NaN input).

Signaling NaN

Signaling NaNs, or sNaNs, are special forms of a NaN that when consumed by most operations should raise the invalid operation exception and then, if appropriate, be "quieted" into a qNaN that may then propagate. They were introduced in IEEE 754. There have been several ideas for how these might be used:

- Filling uninitialized memory with signaling NaNs would produce the invalid operation exception if the data is used before it is initialized
- Using an sNaN as a placeholder for a more complicated object, such as:
  - A representation of a number that has underflowed
  - A representation of a number that has overflowed
  - Number in a higher precision format
  - A complex number

When encountered, a trap handler could decode the sNaN and return an index to the computed result. In practice, this approach is faced with many complications. The treatment of the sign bit of NaNs for some simple operations (such as absolute value) is different from that for arithmetic operations. Traps are not required by the standard. There are other approaches to this sort of problem that would be more portable.

Function definition

There are differences of opinion about the proper definition for the result of a numeric function that receives a quiet NaN as input. One view is that the NaN should propagate to the output of the function in all cases to propagate the indication of an error. Another view, and the one taken by the ISO C99 and IEEE 754-2008 standards in general, is that if the function has multiple arguments and the output is uniquely determined by all the non-NaN inputs (including infinity), then that value should be the result. Thus for example the value returned by hypot(±∞,qNaN) and hypot(qNaN,±∞) is +∞.

The problem is particularly acute for the exponentiation function pow(x,y) = x^y. The expressions 0^0, ∞^0 and 1^∞ are considered indeterminate forms when they occur as limits (just like ∞ × 0), and the question of whether zero to the zero power should be defined as 1 has divided opinion.

If the output is considered as undefined when a parameter is undefined, then pow(1,qNaN) should produce a qNaN. However, math libraries have typically returned 1 for pow(1,y) for any real number y, and even when y is an infinity. Similarly, they produce 1 for pow(x,0) even when x is 0 or an infinity. The rationale for returning the value 1 for the indeterminate forms was that the value of functions at singular points can be taken as a particular value if that value is in the limit the value for all but a vanishingly small part of a ball around the limit value of the parameters. The 2008 version of the IEEE 754 standard says that pow(1,qNaN) and pow(qNaN,0) should both return 1 since they return 1 whatever else is used instead of quiet NaN. Moreover, ISO C99, and later IEEE 754-2008, chose to specify pow(−1,±∞) = 1 instead of qNaN; the reason of this choice is given in the C rationale:[7] "Generally, C99 eschews a NaN result where a numerical value is useful. [...] The result of pow(−2,∞) is +∞, because all large positive floating-point values are even integers."

To satisfy those wishing a more strict interpretation of how the power function should act, the 2008 standard defines two additional power functions: pown(x,n), where the exponent must be an integer, and powr(x,y), which returns a NaN whenever a parameter is a NaN or the exponentiation would give an indeterminate form.

Integer NaN

Most fixed-size integer formats do not have any way of explicitly indicating invalid data. Converting NaN to an integer type, or performing an integer operation whose floating-point equivalent would produce NaN, usually throws an exception. In Java, such operations throw instances of `java.lang.ArithmeticException`.[8] In C, they lead to undefined behavior.

Perl's `Math::BigInt` package uses "NaN" for the result of strings that don't represent valid integers.[9]

```
> perl -mMath::BigInt -e "print Math::BigInt->new('foo')"
NaN
```

Display

Different operating systems and programming languages may have different string representations of NaN.

```
nan
NaN
NaN%
```

```
NAN
NaNQ
NaNS
qNaN
sNaN
1.#SNAN
1.#QNAN
-1.#IND
```

Since, in practice, encoded NaNs have a sign, a quiet/signaling bit and optional 'diagnostic information' (sometimes called a *payload*), these will often be found in string representations of NaNs, too, for example:

```
-NaN
 NaN12345
-sNaN12300
-NaN(s1234)
```

(other variants exist).

Encoding

In IEEE 754 standard-conforming floating-point storage formats, NaNs are identified by specific, pre-defined bit patterns unique to NaNs. The sign bit does not matter. Binary format NaNs are represented with the exponential field filled with ones (like infinity values), and some non-zero number in the significand field (to make them distinct from infinity values). The original IEEE 754 standard from 1985 (IEEE 754-1985) only described binary floating-point formats, and did not specify how the signaling/quiet state was to be tagged. In practice, the most significant bit of the significand field determined whether a NaN is signaling or quiet. Two different implementations, with reversed meanings, resulted:

- most processors (including those of the Intel and AMD's x86 family, the Motorola 68000 family, the AIM PowerPC family, the ARM family, the Sun SPARC family, and optionally new MIPS processors) set the signaling/quiet bit to non-zero if the NaN is quiet, and to zero if the NaN is signaling. Thus, on these processors, the bit represents an 'is_quiet' flag;
- in NaNs generated by the PA-RISC and old MIPS processors, the signaling/quiet bit is zero if the NaN is quiet, and non-zero if the NaN is signaling. Thus, on these processors, the bit represents an 'is_signaling' flag.

The former choice has been preferred as it allows the implementation to quiet a signaling NaN by just setting the signaling/quiet bit to 1. The reverse is not possible with the latter choice because setting the signaling/quiet bit to 0 could yield an infinity.

The 2008 revision of the IEEE 754 standard (IEEE 754-2008) makes formal recommendations for the encoding of the signaling/quiet state.

- For binary formats, the most significant bit of the significand field should be an 'is_quiet' flag. I.e. this bit is non-zero if the NaN is quiet, and zero if the NaN is signaling.
- For decimal formats, whether binary or decimal encoded, a NaN is identified by having the top five bits of the combination field after the sign bit set to ones. The sixth bit of the field is the 'is_quiet' flag. The standard follows the interpretation as an 'is_signaling' flag. I.e. the signaling/quiet bit is zero if the NaN is quiet, and non-zero if the NaN is signaling. A signaling NaN is quieted by clearing this sixth bit.

For IEEE 754-2008 conformance, the meaning of the signaling/quiet bit in recent MIPS processors is now configurable via the NAN2008 field of the FCSR register. This support is optional in MIPS Release 3 and required in

Release 5.[10]

The state/value of the remaining bits of the significand field are not defined by the standard. This value is called the 'payload' of the NaN. If an operation has a single NaN input and propagates it to the output, the result NaN's payload should be that of the input NaN (this is not always possible for binary formats when the signaling/quiet state is encoded by an 'is_signaling' flag, as explained above). If there are multiple NaN inputs, the result NaN's payload should be from one of the input NaNs; the standard does not specify which.

References

1. Bowman, Kenneth (2006). *An introduction to programming with IDL: Interactive Data Language*. Academic Press. p. 26. ISBN 978-0-12-088559-6.
2. Press, William H.; Teukolsky, Saul A.; Vetterling, William T.; Flannery, Brian P. (2007). *Numerical recipes: the art of scientific computing*. Cambridge University Press. p. 34. ISBN 978-0-521-88068-8.
3. William Kahan (1 October 1997). "Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic" (http://www.cs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF) (PDF).
4. "754R Minutes" (http://754r.ucbtest.org/minutes/2017-05-19-minutes.txt). 19 May 2017. Retrieved 25 June 2017.
5. David Goldberg. "What Every Computer Scientist Should Know About Floating-Point" (http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.39.2736).
6. "Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture" (http://www.intel.com/products/processor/manuals/index.htm). April 2008. pp. 118–125, 266–267, 334–335.
7. "Rationale for International Standard—Programming Languages—C, Revision 5.10" (http://www.open-std.org/jtc1/sc22/wg14/www/C99RationaleV5.10.pdf) (PDF). April 2003. p. 180.
8. http://docs.oracle.com/javase/8/docs/api/java/lang/ArithmeticException.html
9. "`Math::BigInt`" (http://perldoc.perl.org/Math/BigInt.html#Input). *perldoc.perl.org*. Retrieved 12 June 2015.
10. "MIPS® Architecture For Programmers – Volume I-A: Introduction to the MIPS64® Architecture" (http://cdn2.imgtec.com/documentation/MD00083-2B-MIPS64INT-AFP-05.04.pdf) (PDF). MIPS Technologies, Inc. 20 November 2013. p. 79. Retrieved 27 September 2017.

External links

- Not a Number (http://foldoc.org/Not-a-Number), foldoc.org
- IEEE 754-2008 Standard for Floating-Point Arithmetic (http://ieeexplore.ieee.org/servlet/opac?punumber=4610933) (subscription required)
