JOURNAL OF OBJECT TECHNOLOGY

Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002

Vol. 5, No. 2, March–Aprile 2006

Not a Number of Floating Point Problems

Gary T. Leavens, Department of Computer Science, Iowa State University, USA

Floating-point numbers and floating-point arithmetic contain some surprising pitfalls. In particular, the widely-adopted IEEE 754 standard contains a number that is "not a number," and thus has some surprising properties. One has to be extremely careful in writing assertions about floating point numbers, to avoid these pitfalls. This column describes the problems and how a language might eliminate them.

That Indian bread is heavenly, isn't it?
Yes, it's nan of the above.

1 THE STANDARD BACKGROUND

Floating-point numbers in many modern programming languages follow the IEEE 754 standard [IEE85]. This standard has had many beneficial effects, for example on the accuracy and portability of floating-point software, none of which are the subject of this column.

Instead, I will discuss a set of problems that arise from the treatment of NaN, the "not a number" value[1], in floating-point operations.

The NaN value is intended to represent an "undefined" value. In particular, it is used to represent the value of expressions such as `0.0/0.0` and `log(-3.14)`. The designers of the standard could have chosen to cause exceptions or to trap such expressions in some way, and indeed it is an option in the IEEE 754 standard to have such expressions cause exceptions or traps. However, not all programming languages support exception handling mechanisms, and so the standard also provides for a special NaN value, which represents such errors.

2 YOUR INTRODUCTION TO NAN PROBLEMS

Once NaN is admitted as a floating-point value, one must define how all of the floating-point operations act on it. Operations that return floating-point numbers are usually defined to be *strict* with respect to NaN arguments in the sense that if any argument is NaN, then the result is NaN. For example, for all floating-point numbers x, $x + \mathrm{NaN} = \mathrm{NaN}$. Strictness does not cause problems.

[1] Technically there is one such NaN value for each floating-point type; for example Java has `Float.NaN` and `Double.NaN`.

Cite this column as follows: Gary T. Leavens: *Not a Number of Floating Point Problems*, in Journal of Object Technology, vol. 5, no. 2, March–Aprile 2006, pages 75–83, http://www.jot.fm/issues/issues_2006_3/column8

Problems arise when dealing with operators that do not return a floating-point number. In particular, comparisons return a Boolean value, and so they cannot be strict in the same sense as operators that return a floating-point value. So what should the value of NaN $<$ NaN be? There are really only three choices. Besides true and false, the other choice would be to cause an exception or a trap. But again, since not all programming languages support exceptions, the standard committee allowed for such operations to return a proper Boolean value. In the case of NaN $<$ NaN, it seems that the obvious answer is false, and this is the IEEE 754 standard's answer.

Indeed, the standard says that most Boolean-valued comparisons return false when either argument (or both) is NaN. The only exception is that for all x, NaN $\neq x$ is true. But following the general rule, for all x, NaN $= x$ is false, NaN $< x$ is false, and so are NaN $\leq x$, NaN $\geq x$, and NaN $> x$.

These decisions require specifiers to be doubly precise when dealing with floating-point numbers.

The difficulty is that formal methods relies on mathematics, especially as embodied in logic and theorem provers. Both mathematics and theorem provers give equality a special status. Equality's most basic attribute is that it is *reflexive*; that is, for all x, $x = x$. However, in the words of Mark Twain [Twa95] "this rule is flung down and danced upon" by the IEEE 754 standard, since NaN $=$ NaN is false.

Another problem for specifiers is the violation of tricotomy. The tricotomy law says that either $x < y$, $x = y$, or $x > y$; but if either x or y is NaN, then all three of these expressions may be false. Violation of tricotomy tells us that, with NaN, floating-point numbers are no longer totally ordered. Of course, this makes sense if you consider NaN to be "not a number" as only the numbers are totally ordered. The trouble is that type systems usually consider NaN to be a number.

3 YOUR AVERAGE EXAMPLE

Let's see some practical examples of the problems caused by these particular properties of NaN. In this column I will use Java [AGH00] and the Java Modeling Language (JML) [BCC$^+$05, LBR06] for such examples. Java floating point follows the IEEE 754 standard, in having a NaN value with the properties of NaN described above. Although Java has an exception handling mechanism, the built-in floating-point operations in Java never raise exceptions, but use NaN to represent errors.

Consider a Java method that takes an array of double precision floating point numbers and returns their average. Figure 1 gives a straightforward implementation in Java.

Specifying the code in Figure 1 is surprisingly difficult. Part of the difficulty comes from the well-known approximate behavior of floating point numbers [Gol91]. Since an implementation's algorithm may change the answer by a small amount, and since we want to permit different algorithms, the specification should not spec-

```
public double average(double [] items) {
    double tot = 0.0;
    for (int i = 0; i < items.length; i++) { tot += items[i]; }
    return tot / items.length;
}
```

Figure 1: The method `average`.

ify a single result, but a range of acceptable results. JML has some built-in convenience methods for specifying such a range of results. For doubles, one can use the class `org.jmlspecs.models.JMLDouble`, which has a static `approximatelyEqualTo` method that tests if the difference between its first two arguments within the tolerance given its third argument. Using this method, one could write the following postcondition in JML (assuming `epsilon` is a `public static final` field that is defined elsewhere).

```
/*@ ensures JMLDouble.approximatelyEqualTo(
  @          \result,
  @          (\sum int i; 0 <= i && i < items.length; items[i])
  @           / items.length,
  @          epsilon);
  @*/
```

However, given the theme of this column, we must immediately suspect whether or not this is a sensible specification.

First, we definitely do not want `epsilon` to be NaN. If `epsilon` were NaN, then it would make no sense to be approximately equal within `epsilon`, as all comparisons to NaN are false. But it will not do to write:

```
//@ public invariant epsilon != NaN;   // wrong!
```

which true for all values of `epsilon`, even when `epsilon` is NaN. This is because in Java `Double.NaN !=` x is true for all numbers, including `Double.NaN` itself. Fortunately, what we want can be specified by an invariant such as the following:

```
//@ public invariant 0.0 < epsilon && epsilon < 0.1;
```

which is false when `epsilon` is NaN. And alternative is to use Java's special test for NaN values and write:

```
//@ public invariant !Double.isNaN(epsilon);
```

Now we have to worry about what happens when one or more of the elements in the array `items` is NaN.[2] In such a case, the Java code returns NaN, which becomes the value of `\result` in the postcondition. But as `approximatelyEqualTo` is specified to return false when any of its arguments are NaN.[3] So the specification is violated whenever the code returns `NaN`, even when it "should" do so.

How can one fix this inconsistency between the code and the specification?

One technique is to specify that no element of the array `items` can be NaN. But again, one has to be doubly precise. A precondition such as the following won't do.

```
/*@ requires (\forall int i; 0 <= i && i < items.length;
  @                            items[i] != Double.NaN);
  @*/
```

The above precondition is equivalent to true, because `!=` returns true when either of its operands is NaN. So one has to write something like the following.

```
/*@ requires (\forall int i; 0 <= i && i < items.length;
  @                            !Double.isNaN(items[i]));
  @*/
```

However, even the precondition above is not enough to prevent problems. If `items` is a zero-length array, then the code for `average` will divide by zero, resulting in NaN. Of course, that will again lead to a violation of the postcondition. To prevent this one needs an additional precondition, like the following.

```
//@ requires items.length != 0;
```

(Notice that there are no worries about using `!=` for integers, since there are no "non-integer" values in Java's `int` type.)

We can also fix the inconsistency between the specification and the code in a different way by adding an additional "specification case." In JML method can have multiple specification cases, connected with the keyword "`also`." Each specification case is governed by a precondition. When the precondition in a specification case is

[2] I am using a version of JML that implicitly adds preconditions to assert that each argument of a reference type, such as `items` is not null [CR05].

[3] Making `approximatelyEqualTo` return false when one of its arguments is NaN is necessary to be consistent with the IEEE 754 standard and Java's arithmetic.

```
/*@   requires (\forall int i; 0 <= i && i < items.length;
  @                             !Double.isNaN(items[i]));
  @   requires items.length != 0;
  @   ensures JMLDouble.approximatelyEqualTo(
  @            \result,
  @            (\sum int i; 0 <= i && i < items.length; items[i])
  @             / items.length,
  @            epsilon);
  @ also
  @   requires (\exists int i; 0 <= i && i < items.length;
  @                             Double.isNaN(items[i]))
  @            || items.length == 0;
  @   ensures  Double.isNaN(\result);
  @*/
public double average(/*@ non_null @*/ double [] items) {
    double tot = 0.0;
    for (int i = 0; i < items.length; i++) { tot += items[i]; }
    return tot / items.length;
}
```

Figure 2: The method `average`, with two specification cases. In JML specifications are written above the method that they specify.

true, the method must satisfy that case's postcondition. (If there are several specification cases with true preconditions, then all of the corresponding postconditions must be satisfied. This idea of breaking up a specification into several cases is due to Jeannette Wing [Win83] and was independently reinvented by Alan Wills [Wil94].)

Thus, in JML, we could keep the specification we have developed so far, and add to it a specification case whose precondition allows `items` to contain NaN and also allows `items` to be empty. The postcondition should describe what the code does in this case, which is to return NaN. Of course we have to be doubly precise, and use `Double.isNaN` instead of using `==` to test for NaN, but if we do this, everything works. The result is the second specification case shown in Figure 2.

Looking at Figure 2, one immediately sees that the specification is more verbose than the code. Most of the extra specification, however, is there to handle NaN. We had to be quite careful about NaN in the specification, because we are necessarily writing Boolean assertions instead of code that returns floating point numbers. The part of the specification that is not concerned with NaN, namely the first specification case minus its first precondition (i.e., lines 3-8 of Figure 2), is quite similar in length to the method's code. While the second specification case is entirely optional, we cannot eliminate the first precondition in the first specification case. That precondition is necessary to prevent inconsistencies. Thus the presence of NaN adds an unavoidable cost to this specification.

It is worth pointing out that this cost is not a result of formalizing the specification. Even if we were to write an informal English description of the `average` method, the presence of NaN as a value makes it necessary to state what happens when one of the elements of `items` is NaN. To see this, just consider what a test case for the `average` method: what is the expected result when one of the elements is NaN?

This kind of example illustrates our experience with specifying many methods in the Java libraries that work with floating-point numbers. Unless one considers NaN as a special case, the specification is likely to be wrong, even if it looks right.

4 NAN OF THESE LANGUAGES

The specification problems with NaN in Java and JML result from an inconsistency between the value space of floating-point numbers with NaN and the Booleans. This inconsistency could be resolved in several ways.

The first is to permit a third Boolean value, NaB (which stands for "not a Boolean"). We would define all floating-point comparisons involving NaN to return NaB. However, this requires the definition of a three-valued logic for the Booleans. While various consistent three-valued logics have been defined [BCJ84], they are not popular in current programming languages or theorem provers. Furthermore, for consistency, one would probably need to have such "not a T" values for every type T. Consider, for example, what should be the result of the following if-expression?

```
(NaB ? 1 : 2)
```

In a language like Java, a reasonable answer might be "not-an-integer." While three-valued logics do not present insurmountable problems for specification languages, it is not clear how well they would fare with users.

The second way to resolve the inconsistency would be to ban NaN from the language. Built-in operations and user-defined methods would throw various exceptions whenever an expression would otherwise return NaN. Besides simplifying the language, this strategy makes the treatment of floating-point numbers consistent with integer arithmetic. It also avoids the mathematical problems with NaN and specification problems described above. Specifiers would not have to NaN as a special case, saving considerable overhead and eliminating a wide class of potential errors. Programmers already are familiar with exceptions and there are well established techniques for specifying them, including simply specifying the normal case using a precondition [Mey97]. The only disadvantage is the possible loss of efficiency in floating point computations. However, Hauser has argued that appropriately-designed exception handling mechanisms can both clarify floating point code and even improve its speed [Hau96].

Are their compromises between these approaches? One can imagine, as allowed by the IEEE 754 standard, a set of switches, global parameters, or options, that cause floating-point arithmetic to either cause exceptions or return NaN. However, once NaN is part of the language, specifications will have to consider it, except in programs where NaN values are prohibited. Libraries, however, would either have to consider NaN values in their specifications or require the switches to be set in such a way as to prohibit NaN values. This seems unsatisfactory from the point of view of specification and somewhat ad hoc.

A better approach might be to allow NaN as a value, but not in the standard floating-point types. Instead, there would be a separate type that included all of the normal floating-point values and NaN. (This is analogous to the treatment of null proposed by Fähndrich and Leino [FL03], and similar to ideas of Goguen and Meseguer [GM87].) Applying this idea to Java would mean that `double` and `float` would not include NaN as a value, and instead there would be two new types, `doubleWithNaN` and `floatWithNaN`. The type `double` would be a subtype of `doubleWithNaN`, and so assignment of `double` values to variables of the latter type would be permitted without casting. However, casting would be necessary to convert from `doubleWithNaN` to `double`, and the cast would throw an exception if the value was NaN.

To avoid the problems with the lack of reflexive equality, the `==` and `!=` operators would not be defined on the type `doubleWithNaN`. There would instead be a built-in primitive to test if a `doubleWithNaN` is NaN. Users and libraries could use this primitive together with casting to construct tests, but since they would not be using the `==` and `!=` syntax, they would not cause the same kind of understanding problems.

The same treatment could also be applied to other built-in comparison operators. That is, the operations `<`, `<=`, `>`, and `>=` would not be available for the type `doubleWithNaN`. Again, users could construct their own operations to do comparisons and such operations could be put in libraries, however the lack of a standard syntax would give users a clue that the tricotomy law might not hold.

It would be interesting to flesh out these approaches, implement them, and then compare them from many points of view, including efficiency, reliability of floating-point code, and ease and clarity of specification.

5 CONCLUSIONS

When writing specifications for methods involving floating-point numbers, specifiers have to be doubly careful. In particular, they must always consider the possibility of NaN as a value, and the peculiar ways in which comparisons work in the presence of NaN. While specification language features such as the ability to write multiple specification cases can help clarify the intended specification, language designers could completely eliminate these problems. A simple and consistent approach is to throw exceptions instead of using NaN.

If it is desired to keep NaN as a value, the type system could be adjusted to avoid some of its pitfalls. While the resulting type system is more complex, specifications would be considerably simplified for many common cases. This might ultimately help programmers and lead to more reliable languages.

ACKNOWLEDGMENTS

Thanks to the JML community for their work on JML and its tools. This work was supported in part by NSF grants CCF-0428078, and CCF-0429567.

REFERENCES

[AGH00] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language Third Edition*. Addison-Wesley, Reading, MA, 2000.

[BCC+05] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, June 2005.

[BCJ84] H. Barringer, J. H. Cheng, and C. B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21(3):251–269, October 1984.

[CR05] Patrice Chalin and Frèdèric Rioux. Non-null references by default in the java modeling language. In *Proceedings of the Workshop on the Specification and Verification of Component-Based Systems (SAVCBS'05)*, volume 31(2) of *ACM Software Engineering Notes*. ACM, 2005.

[FL03] Manuel Fähndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented langauge. In *OOPSLA '03: Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, volume 38(11) of *ACM SIGPLAN Notices*, pages 302–312, New York, NY, November 2003. ACM.

[GM87] Joseph A. Goguen and Jose Meseguer. Order-sorted algebra solves the constructor-selector, multiple representation and coercion problems. Technical Report CSLI-87-92, Center for the Study of Language and Information, March 1987. Appears in Second Annual Symposium on Logic in Computer Science, Ithaca, NY, June, 1987, pages 18-29.

[Gol91] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, March 1991.

[Hau96] John R. Hauser. Handling floating-point exceptions in numeric programs. *ACM Transactions on Programming Languages and Systems*, 18(2):139–174, 1996.

[IEE85] IEEE Standards Committee 754. *IEEE Standard for binary floating-point arithmetic, ANSI/IEEE Standard 754-1985.* Institute of Electrical and Electronics Engineers, New York, 1985. Reprinted in ACM SIGPLAN Notices, 22(2):9-25, 1987.

[LBR06] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06-rev29, Iowa State University, Department of Computer Science, January 2006. To appear in *ACM SIGSOFT Software Engineering Notes*.

[LH94] K. Lano and H. Haughton, editors. *Object-Oriented Specification Case Studies.* The Object-Oriented Series. Prentice Hall, New York, NY, 1994.

[Mey97] Bertrand Meyer. *Object-oriented Software Construction.* Prentice Hall, New York, NY, second edition, 1997.

[Twa95] Mark Twain. Fenimore Cooper's Literary Offenses, 1895. Reprinted in *Twain: Collected Tales, Sketches, Speeches, and Essays: Volume 2: 1891-1910*, pages 180-192 (Penguin Putman Inc., 1992).

[Wil94] Alan Wills. Refinement in Fresco. In Lano and Houghton [LH94], chapter 9, pages 184–201.

[Win83] Jeannette Marie Wing. A two-tiered approach to specifying programs. Technical Report TR-299, Massachusetts Institute of Technology, Laboratory for Computer Science, 1983.

ABOUT THE AUTHORS

Gary T. Leavens is a professor of computer science at Iowa State University in Ames, Iowa. He has taught there since receiving his Ph.D. from MIT in 1989. His research interests include programming and specification language design and semantics, program verification, and formal methods, with an emphasis on the object-oriented and aspect-oriented paradigms. His email is leavens@cs.iastate.edu. See also http://www.cs.iastate.edu/ leavens/.