Comments on Java Numerics

David Hough

ABSTRACT

PRELIMINARY - SUBJECT TO REVISION - NOT FOR PUBLICATION

Java is not a machine-oriented language; it is intended to be as machine-independent as possible. Identical numerical results on every machine is an important part of that goal.

Java can be improved in its suitability for numerical computation, by allowing operator notation to be extended to user-defined types, and by recognizing IEEE 754 exceptions.

troff, postscript, and ascii versions of this draft will be available at http://www.validgh.com/.java.

Jerome Coonen awoke the numeric-interest@validgh.com mailing list from a long dormancy by arguing that the Java language definition should not mandate a uniform style of expression evaluation that is suboptimal on certain architectures, in the sense of being sometimes slower and sometimes less accurate, than methods native to that particular architecture. The following are my personal views on these and some other aspects of Java for numerical computing. While I expect to be involved in testing Java implementations for correctness and performance, I was not involved in the definition or implementation of Java, and so my descriptions of the current state of Java may be incomplete or out of date, but I hope they're not grossly wrong.

***** What Java is**

The next paragraphs discuss some features about the current definition of Java and attempt to explain their rationale.

Java is NOT a machine-oriented language

Among its other goals, C aims to promote machine-independent programming AND machine-dependent programming. Different language features cater to the different needs of these goals. Few rules on the size on integral and floating-point types, no rules on expression evaluation, conditional compilation, and ready access to internal representations of types, all act together to make it possible to write programs that are very specific to particular computers. Yet by avoiding these features and a few others it is possible to write relatively machine-independent programs, as long as every machine on which those programs are run conforms to the programmer's mental model of what's important. If the programmer thought to allow for 16, 32, and 64-bit ints, then the program may only be machine independent until compiled and run on a system with 36 or 48-bit ints. If the programmer thought to allow any of IEEE 754's varieties of floating-point arithmetic, then the program may only be machine independent until compiled and run on a DEC VAX, IBM 370, or Cray supercomputer. Creating and testing programs that really are independent enough to be run on all these machines is an arduous task.

In contrast, promoting machine-dependent programming would have to be considered an anti-goal of Java; that's what the "100% Pure Java" campaign is all about: assisting software producers in avoiding dependencies on particular machines. Instead Java programs are intended to produce identical results on all supported machines. One important reason for that goal is to facilitate validation of claimed Java implementations: all floating-point results should agree to the last bit. The alternative is most conventional languages like Fortran and C, in which hardware faults and compiler bugs sometimes masquerade as "roundoff differences" and conversely, roundoff differences are sometimes misclassified as hardware faults and compiler bugs, particularly with aggressively optimizing compilers. Persons involved in intensive numerical work put up with these differences, and the costs associated with analyzing and correcting them, when their programs and problems demonstrate a high enough performance benefit. In contrast, most Java programming is likely to be somewhat less numerically intensive, and the tradeoff is the other way: the cost of dealing with ultimately insignificant numerical differences doesn't justify the performance improvement available by tolerating them.

If there is to be but one Java expression evaluation model, how should it be chosen? The Java expression evaluation model is a simple one supported by most old and new computers used for floating-point computation: only single and double precision variables and registers exist, and floating-point instructions apply to only one or two input operands at a time. No extra-precise accumulators or fused three-operand instructions may be employed. That model is supported in hardware or may be efficiently emulated on all current popular RISC workstations and CISC PC's. It is true that x86-architecture PC's could sometimes provide faster and more accurate results by exploiting extended-precision accumulators, and PowerPC's could sometimes provide faster and more accurate results by exploiting fused multiply-add instructions, but the performance penalties for adhering to Java's model are not overwhelming and could be reduced further with a few modifications in the next revisions of those architectures. In contrast, based on past experiences of compilers' routine misapplication of extended-precision registers and fused operations, there is plenty of reason to doubt that the benefits of those architectures would be readily achieved if they were mandated by Java, even if there were no performance degradation on machines which had not provided hardware support for those features.

Java doesn't allow VAX, IBM 370, or Cray floating-point formats; Java implementations on those systems must emulate IEEE 754 single and double precision. Java's insistence on a single expression evaluation model means that systems that are intended only to execute Java will not support other kinds of expression evaluation, but there are numerous other languages - namely almost every other language - available for that purpose.

Functions are fully defined in Java

Defining a function by writing a reference implementation in Java defines its floating-point behavior completely; implementations of that function that produce different results are defective to a greater or lesser extent. Java numerical functions intended to be part of a portable Java implementation are best defined by such reference implementations. That includes the class math functions, although they may not yet be so defined, and so are not a good example; consider instead a BLAS dot product. If the definition of Java ddot were written in Java then all implementations of Java ddot would produce identical results. The implementation may differ, but the results must be identical. In contrast, another dot product function nddot might be defined to produce a native dot product "sort of like ddot" but perhaps faster or more accurate or both. Because it's not defined as a Java program, nor defined as a mathematical abstraction like a correctly-rounded dot product, its exact definition is left to the good taste of the implementer - who may follow standard industry practice by denying the suitability of his implementation for any particular purpose while advertising its speed or accuracy derived by exploiting characteristics of particular hardware through a native implementation. Your nddot and mine may have the same name, and work sort of alike on bland data, but may exhibit strikingly different characteristics when challenged. This should not happen with programs defined by a Java implementation - that with a looser language definition, they COULD run faster or more accurately on SOME machines, is of no help on ALL machines.

Similarly there will be implementations of languages that can't be called Java because they are tuned to run faster on a particular machine - and maybe more or less accurately, though history is not encouraging for the former. So call them Fava, Kava, or Lava as languages "sort of like Java" with the exact definition left to the good taste of the implementer.

Probably there will be Java implementations that can be converted to Fava, Kava, or Lava by a compile-time option. For most programs these options will produce minor performance enhancements at best.

***** What Java needs**

The next few paragraphs describe some extensions to Java that would make it a better language for scientific computing. For all I know, some of these may be planned for future releases.

Operator extension to user-defined types

Java's definers wisely defined only a few numerical types as primitive in the language, and excluded operator overloading, a technique available in some languages that can be used to redefine the meaning of integer addition, for instance. I sympathize with that exclusion. Not so wisely, though, there is no operator notation extension to cover necessary user-defined types, such as complex, interval, or extended precision, not to mention complex extended intervals. Thus adding numerical types to Java by class definitions is either unduly burdensome to users, because every operator becomes a function invocation, or programs require a source code preprocessor of the style popular in Fortran circles twenty years ago - with the result that the code executed by the interpreter or examined by a debugger does not resemble the source code used by the programmer.

IEEE 754 Exceptions

Java recognizes certain exceptions such as integer overflow, but doesn't recognize the five IEEE 754 exceptions. These are essential to building robust fast programs. The minimum useful support is to detect specified exceptions when they arise in basic blocks of code and take alternate action if they do. By not specifying how much of the protected block is executed when an exception is detected, the definition would permit implementations to use flags or traps and to optimize within blocks fairly freely. There would be discernible differences in implementations that examined side effects produced by basic blocks whose partial execution was interrupted by an exception, but such programs could be deemed illegal Java.

Unmentioned exceptions would produce IEEE 754 nonstop defaults as at present. Handling exceptions arising in external functions would require more infrastructure since the external functions might be separately compiled and thus would have to be careful about preserving flags or traps.

Exception handling that protects only basic blocks is quite a bit less than IEEE 754 requires and recommends, but it should suffice for many applications.

Undefined areas

If the elementary transcendental functions in class math do not have a published reference implementation yet, then their implementation currently depends on good taste. It would be better to have a reference implementation such as a translation of fdlibm. In some ways a correctly-rounded reference implementation would be even better, but such an implementation based only on high-precision software arithmetic would be very slow, and one based on large tables would be unsuitable for the embedded applications for which Java was first designed; these points might be overlooked if good public implementations of correctly-rounded transcendental functions were available, but that has not happened yet.

Java conversions from binary floating-point to decimal string representations are intended to be correctly rounded but are somewhat limited in releases so far. Correctly-rounded base conversion is an important aspect of portable identical results.

***** Interval arithmetic**

Interval arithmetic is a method to replace uncertain floating-point results with intervals guaranteed to contain the correct results. When it can be successfully applied, it thus produces better results than most floating-point computations usually have, because they are guaranteed. Although there have been many successes, interval arithmetic is still difficult to apply successfully, for a variety of reasons. So interval arithmetic should be available as a java class (with operator extensions as mentioned above) but it is far from being a substitute for "point methods," as interval enthusiasts refer to ordinary floating-point arithmetic. Most Java applications at present are supposed, rightly or wrongly, to be either obviously right enough or obviously wrong. As they get more complicated numerically, people will realize that while getting identical results on all machines is an important benefit of Java programming, if it can't be determined whether they are identical CORRECT results, the benefit is of little value.