The Ada Numerics Model

Jean-Pierre Rosen

Adalog, 2 rue du Docteur Lombard, 92130 Issy-Les-Moulineaux, France.; email: rosen@adalog.fr

Abstract

This paper describes the challenges of making portable calculations across different architectures, and how the Ada model addresses the issues.

Keywords: Ada, numerics, floating points, fixed points.

1 What is numerical analysis?

All programming languages feature so-called “real” types. However, these types are very different from mathematical reals. The mathematical set $\Re$ cannot be represented on a computer: it has an infinite number of values, even for a bounded segment. A computer can represent only types with a finite number of values, and these values can be (at most) rational numbers, since there is no finite representation of irrational numbers.

However, computers are intended to perform computations for the real world, and if you want to compute the circumference of a circle given its radius, you will need Π, which *is* irrational!

Therefore, we can define numerical analysis as the art of making *not too wrong* computations in the real world, using only the finite subset of rational numbers that a computer can handle.

Moreover, since the result is not exact, it is important to be able to compute how wrong (or more precisely uncertain) the result is.

2 Hardware formats and languages

There are many ways of representing real numbers on a computer For example, [1] describes 76 different floating point formats! Moreover, there are often several available formats on a given computer, allowing various trade-offs between range, accuracy, and memory space. For example, the popular IEEE-754 standard [2,3] features 5 standard formats, (3 binary, 2 decimal), + extensions. The old DEC/Vax architecture is another interesting case, as pictured in figure 1:

Size	Exponent	Mantissa
32 bits	8	23
64 bits	8	55
	11	52
128 bits	15	112

Figure 1 VAX floating point formats

Note that there are two different 64 bits format, one with more accuracy (longer mantissa) and the other one with more range (longer exponent). Talking about “short” or “long” floats cannot describe this situation.

Actually, the notion of short or long floats dates back to the early times of Fortran, when most computers had only two floating point formats. Most today’s languages still define various floating point types just by the size of the type, without any idea of the actual accuracy (or range) implied by the type, and no definition of the accuracy of computations.

The IEEE-754 standard tried to address these issues by defining a number of standard formats and the associated accuracy, including a precise definition of arithmetic: two computers implementing the standard will give exactly the same results. However, the relation to programming languages is delegated to the programming language standard, including the means to adjust a certain number of features (like exceptions). And from a programmer’s point of view, the issue of portability for computers that do not implement the standard remains.

3 Ada model and real types

3.2 Model of arithmetic

The Steelman requirements [4] called for both “approximate” and “exact” computations whose accuracy could be chosen by the user, independently of the underlying architecture.

The solution offered by Ada is inspired by the notion of approximate values in physics: a value does not stand only for itself, but represents a small range of values corresponding to an uncertainty around the value. Based on this, a range arithmetic can be defined, the so-called Brown’s model [5].

Physicists use two kind of approximations: absolute approximations, where the uncertainty is the same for the whole range of values (i.e. value is 5V±0.1V), and relative approximations where the uncertainty is proportional to the value (i.e. value is 5V±1%). Similarly, Ada offers two kind of real types: fixed point types corresponding to absolute approximation, and floating point types corresponding to relative approximation.

The syntax of the definition of a fixed point type is as follows:

```
-- Binary fixed
type name is delta step range min .. max;
type Volts is delta 0.01 range 0.0 .. 100.0;
```

```
-- Decimal fixed
type name is delta step digits number_of_digits
  [ range  min .. max ];
type Euros is delta 0.01 digits 11;
```

The syntax of the definition of a floating point type is as follows:

```
type name is digits number_of_digits [range min..max];
type Length is digits 5 range 0.0..40.0E6;
```

Unlike most other languages, the programmer does not choose one of the types provided by the computer, but expresses the requirements on the mathematical properties of the type. It is up to the compiler to choose an appropriate machine type that fulfils the requirements.

The definition of an Ada real[2] type defines a set of values, called *model numbers* that must be represented exactly on every implementation. If no machine type is available that satisfies the declaration (i.e. that can represent exactly all model numbers), the declaration is rejected by the compiler. The machine type chosen by the compiler may include more values than the model numbers: these extra values are called *machine numbers* and provide more accuracy than the minimum guaranteed by the declaration.

Since the programmer specifies the requirements for accuracy and range, the compiler can choose the most appropriate among all available machine types.

This defines the accuracy of the definition of data. In addition, if the compiler implements the numerics annex, there are additional requirements on the accuracy of operations, including for the functions provided by the numerical libraries: elementary functions, linear algebra, and random number generators.

The principle of these requirements, following Brown's model, is as follows:

- If both operands are model numbers and the mathematical result of the operation is a model number, then the computed result must be that model number, exactly.
- Otherwise, if the mathematical result lies between two model numbers, the computed result can be any value belonging to the *model interval* bounded by the nearest model numbers that surround the mathematical result. This means that the compiler can keep more accuracy if hardware permits, but that the inaccuracy is bounded independently of the hardware.
- The above principle is extended when both operands are only known to belong to some intervals: the mathematical operation is (formally) performed between all values of operands in their respective model intervals, thus defining a *result interval* that can extend over several model intervals, and is not necessarily bound by model numbers; the computed result must belong to this result interval, extended to the nearest model numbers.

This is basically like computing approximations in physics, except for the extra "digitalization noise" due to extending the intervals to the nearest model numbers.

Note that with this model, there is no notion of *underflow*. Some arithmetic models make a special case when a mathematical result is not strictly zero, but the computed result is an exact zero. In Ada, this situation means that zero belongs to the result interval and is an acceptable result; it is not a special condition.

As far as portability of computations is concerned, a program running on two different computers will *not* give the exact same result; however, both results will belong to an interval whose range can be computed independently of the implementation. What Ada guarantees is not an absolute result (which is meaningless anyway, since there is always some uncertainty), but portable bounds on the maximum error of computations.

In addition, the standard requires that all static expressions be evaluated *exactly*; no error can be introduced at compile time by differences of evaluations within the compiler.

3.3 Fixed point vs. floating point types

All languages provide floating point types, and programmers are used to them. But few languages provide fixed point types, and people who are not familiar with them often do not consider their use. They constitute however an additional possibility of Ada that can often better match the problem domain than floating point types. Figure 2 shows a comparison of the respective model numbers of a fixed point type and of a floating point type:

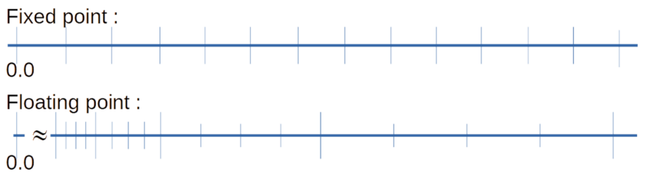

Figure 2 Floating points vs. fixed points model numbers

Fixed point model numbers are evenly spaced, while floating point model numbers are very tight when close to zero, but lose (absolute) accuracy when away from zero. Fixed point types are often more appropriate to represent money or physical values, like readings from measurements devices. This is especially the case for time, where the zero value is arbitrary[3], and there is no reason to provide more accuracy when getting closer to zero[4].

3.2 Numerical portabilities

Portability of numerical computations is not a single goal. There are actually several kinds of numerical portabilities, as allowed by the Ada model.

[2] Remember that the term "real" in Ada encompasses both floating point and fixed point types.

[3] In Ada, the type duration is a fixed point type.

[4] Except for astronomers who model the big bang, of course…

For example, you may want to benefit from the “natural” types of your computer, and be able to determine the confidence range of your result. You will use predefined types, like Float or Long_Float. Knowing your algorithm and given the various attributes provided by Ada, a numerical analyst is able to determine the accuracy of the result. This can be called *a posteriori* portability: the same program run on machine A will print “PI=3.14+-0.05”, while on machine B it will print “PI=3.1415+-0.0005”. Both results are correct, although not identical.

In other cases, you may have requirements on the accuracy of the result; for example, the maximum error on the computed speed of a vehicle must not exceed 5 km/h. Like any requirements, it should be provably met! Given the Ada model, your numerical analyst is able to determine the required accuracy (and range) of the data that are part of the computation. These can be expressed in Ada terms, and guaranteed by the implementation, independently of any architecture. This can be called *a priori* portability: different machines may give different results, but they will all be within the stated requirements.

Conclusion

For many programming languages, as soon as a numeric value includes a decimal point, the only choice is between short and long floats, which usually boils down to short floats if memory space is an issue, or long floats “just to be on the safe side” otherwise. Not exactly an engineered approach…

Ada offers a rich range of real types that can provably match stated requirements and guarantee the maximum uncertainty of the computed results, independently of the underlying architecture. When it comes to choosing a programming language for a development, this aspect of Ada should really be more known to all those with requirements on the accuracy of computations.

References

[1] http://www.quadibloc.com/comp/cp0201.htm

[2] Wikipedia, IEEE 754, https://en.wikipedia.org/wiki/IEEE_754

[3] "IEEE Standard for Binary Floating-Point Arithmetic". ANSI/IEEE Std 754-1985.

[4] Department of Defense, *Requirements for High Order Computer Programming Languages "STEELMAN"*, June 1978. https://en.wikisource.org/wiki/Steelman_language_requirements

[5] W. S. Brown, *A Simple but Realistic Model of Floating-Point Computation*, ACM Transactions on Mathematical Software, Volume 7 Issue 4, Dec. 1981, pp. 445-480.