Chapter 13

Formalisms for Certifying Floating-Point Algorithms

While the previous chapters have made clear that it is common practice to certify floating-point algorithms with pen-and-paper proofs, this practice can lead to subtle bugs. Indeed, floating-point arithmetic introduces numerous special cases, and examining all the details would be tedious. As a consequence, the certification process tends to focus on the main parts of the correctness proof, so that it does not grow out of reach.

For instance, the proof and even the algorithm may no longer be correct when some value is equal to or near a power of the radix, as being a discontinuity point of the ulp function. Moreover pen-and-paper proofs may be ambiguous, e.g., by being unclear on whether the exact value or its approximation is considered for the ulp.

Unfortunately, experience has shown that simulation and testing may not be able to catch the corner cases this process has ignored. By providing a stricter framework, formal methods provide a means for ensuring that algorithms always follow their specifications.

13.1 Formalizing Floating-Point Arithmetic

In order to perform an in-depth proof of the correctness of an algorithm, its specification must be precisely described and formalized. For floating-point algorithms, this formalization has to encompass the arithmetic: number formats, operators, exceptional behaviors, undefined behaviors, and so on. A new formalization may be needed for any variation in the floating-point environment.

Fortunately, the IEEE 754 standard precisely defines some formats and how the arithmetic functions behave on these formats: “Each operation shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then rounded that result […]”
This definition makes a formalization of floating-point arithmetic both feasible and practical, as long as the implementation (language, compiler, etc.) strictly follows the IEEE-754 requirements, which may not be the case in practice (see Chapter 7 and Section 3.4.6). However, a formalization can still take into account the specificity of the implementation; for the sake of simplicity and because it is not possible to be exhaustive, such a specificity will be ignored in the following.

Moreover, this single formalization can be used for describing the specification of any algorithm whose implementation relies on this standardized arithmetic.

13.1.1 Defining floating-point numbers

The first stage of a formalization lies in a proper definition of the set of floating-point numbers. This definition can be performed at several levels. First of all, one should define the set itself and the values that parameterize it, e.g., radix and precision. Then comes the actual representation of the numbers: how they translate from and to streams of bits. Finally, a semantics of the numbers is needed. It is generally provided by their interpretation as a subset of the real numbers.

Structural definition

The IEEE 754 standard describes five categories of floating-point data: signed zeros, subnormal numbers, normal numbers, signed infinities, and Not a Number (NaN) data. These categories can be used to define the set of data as a disjoint union of sets of data. Any given floating-point datum is described by one and only one of the following branches. For each category, some additional pieces of information represent the datum, e.g., its significand.

Floating-point data :=
| Zero:    | sign |
| Subnormal: | sign, significand |
| Normal:  | sign, significand, exponent |
| Infinity: | sign |
| NaN:     | payload |

While the same disjoint union could be used to define both binary and decimal floating-point numbers, the formalization may be simpler if the radix $\beta$ is encoded as a parameter of the whole type. The type of the “significand” fields has to be parameterized by the precision $p$, while the type of the “exponent” is parameterized by $e_{\min}$ and $e_{\max}$. The type of the “payload” could also be parameterized by $p$; but for clarity, we will assume it is not. The type of “sign” is simply the set $\{+,-\}$, possibly encoded by a Boolean. The fully featured disjoint union has now become: