Chapter 4

Basic Properties and Algorithms

In this chapter, we present some short yet useful algorithms and some basic properties that can be derived from specifications of floating-point arithmetic systems, such as the ones given in the various successive IEEE standards. Thanks to these standards, we now have an accurate definition of floating-point formats and operations. The behavior of a sequence of operations becomes at least partially\(^1\) predictable (see Chapter 7 for more details on this). We therefore can build algorithms and proofs that use these specifications.

This also allows the use of formal proofs to verify pieces of mathematical software. For instance, Harrison uses HOL Light to formalize floating-point arithmetic [168, 171] and check floating-point trigonometric functions [169] for the Intel-HP IA-64 architecture. Russinoff [355] used the ACL2 prover to check the AMD-K7 floating-point multiplication, division, and square root instructions. Boldo, Daumas, and Théry use the Coq proof assistant to formalize floating-point arithmetic and prove properties of some arithmetic algorithms [33, 256].

4.1 Testing the Computational Environment

4.1.1 Computing the radix

The various parameters (radix, significand and exponent widths, rounding modes, etc.) of the floating-point arithmetic used in a computing system may strongly influence the result of a numerical program. Indeed, very simple and short programs that only use floating-point operations can find these parameters. An amusing example of this is the C program (Listing 4.1), given by Malcolm [146, 269], that returns the radix \(\beta\) of the floating-point system. It works if the active rounding mode is one of the four rounding modes of IEEE

\(^1\)In some cases, for instance, intermediate calculations may be performed in a wider internal format. Some examples are given in Section 3.3.
Chapter 4. Basic Properties and Algorithms

754-1985 (or one of the rounding direction attributes of IEEE 754-2008 [187]). It is important to make sure that a zealous compiler does not try to "simplify" expressions such as \((A + 1.0) - A\). See Chapter 7 for more information on how languages and compilers handle floating-point arithmetic.

C listing 4.1 Malcolm's algorithm (Algorithm 4.1, see below), written in C.

```c
#include <stdio.h>
#include <math.h>

#pragma STDC FP_CONTRACT OFF

int main (void)
{
  double A, B;
  A = 1.0;
  while ((A + 1.0) - A == 1.0)
    A *= 2.0;
  B = 1.0;
  while ((A + B) - A != B)
    B += 1.0;
  printf ("Radix B = %g\n", B);
  return 0;
}
```

Let us describe the corresponding algorithm more precisely. Let \(\circ\) be the active rounding mode. The algorithm is

Algorithm 4.1 Computing the radix of a floating-point system.

\[
\begin{align*}
A & \leftarrow 1.0 \\
B & \leftarrow 1.0 \\
\textbf{while } \circ (\circ (A + 1.0) - A) = 1.0 & \textbf{ do } \\
& \quad A \leftarrow \circ (2 \times A) \\
\textbf{end while} \\
\textbf{while } \circ (\circ (A + B) - A) \neq B & \textbf{ do } \\
& \quad B \leftarrow \circ (B + 1.0) \\
\textbf{end while} \\
\textbf{return } B
\end{align*}
\]

Incidentally, this example shows that analyzing algorithms sometimes depends on the whole specification of the arithmetic operations, and especially the fact that they are correctly rounded:

- If one assumes that the operations are exact, then one erroneously concludes that the first loop never ends (or ends with an error due to an overflow on variable \(A\)).