A Formally-Proved Algorithm to Compute the Correct Average of Decimal Floating-Point Numbers

Sylvie Boldo, Florian Faissole, Vincent Tourneur

A Formally-Proved Algorithm to Compute the Correct Average of Decimal Floating-Point Numbers

Sylvie Boldo*†, Florian Faissole*†, and Vincent Tourneur‡*†
*Inria
†LRI, CNRS & Univ. Paris-Sud, Université Paris-Saclay,
bâtiment 650, Université Paris-Sud, F-91405 Orsay Cedex, France
‡LRDE, EPITA, 14-16, rue Voltaire, F-94276 Le Kremlin-Bicêtre Cedex, France

Abstract

Some modern processors include decimal floating-point units, with a conforming implementation of the IEEE-754 2008 standard. Unfortunately, many algorithms from the computer arithmetic literature are not correct anymore when computations are done in radix 10. This is in particular the case for the computation of the average of two floating-point numbers. Several radix-2 algorithms are available, including one that provides the correct rounding, but none hold in radix 10. This paper presents a new radix-10 algorithm that computes the correctly-rounded average. To guarantee a higher level of confidence, we also provide a Coq formal proof of our theorems, that takes gradual underflow into account. Note that our formal proof was generalized to ensure this algorithm is correct when computations are done with any even radix.

1. Introduction

Floating-point (FP) computations are everywhere in our lives. They are used in control software, used to compute weather forecasts, and are a basic block of many hybrid systems: embedded systems mixing continuous, such as sensors results, and discrete, such as clock-constrained computations. Computer arithmetic [1], [2], is mostly known (if known at all) to be inaccurate, as only a finite number of digits is kept for the mantissa. On top of that, only a finite number of digits is kept for the exponent. This creates the underflow and overflow exceptions, that are often dismissed. We are here mostly interested in handling underflow.

Which numbers are available and how operations behave on them was standardized in the IEEE-754 standard [3] of 1985, which was revised in 2008 [4]. This revision in particular includes radix-10 FP numbers and computations. Mainframes with decimal FP units are now available. This leads to a new branch of computer arithmetic dedicated to decimal arithmetic, for developing both hardware and software.

The chosen example is very simple: how to compute the average of two decimal FP numbers:

$$\circ\left(\frac{x+y}{2}\right),$$

with $\circ$ being a rounding to nearest, for instance with tie-breaking to even. This computation seems easy in radix 2, but it is not that easy due to spurious overflow. See Section 2 for references to radix-2 average algorithms. This question of computing the average was hardly

studied in radix 10 before. The naive formula `(x+y)/2` is rather accurate, but does not always give the correct result in radix 10, meaning the rounding to nearest of the mathematical average (see also Section 3 for additional incorrect algorithms).

In order to have a high guarantee on this mathematical result, we will rely on formal methods. Floating-point arithmetic has been formalized since 1989 in order to formally prove hardware components or algorithms [5], [6], [7]. We will use the Coq proof assistant [8] and the Flocq library [9]. Flocq offers a multi-radix and multi-precision formalization for various floating- and fixed-point formats (including FP with or without gradual underflow) with a comprehensive library of theorems. Its usability and practicality have been established against test-cases [10].

All the theorems stated in this article correspond to Coq theorems available at:
Notations are as follows: $\oplus$ denotes the rounded addition, $\ominus$ the rounded subtraction, and $\oslash$ the rounded division. The significand of an FP number x is denoted by M_x.

The outline of this article is as follows. Section 2 gives some references on previous works about computing the average of binary FP numbers. Section 3 provides counter-examples to many simple algorithms, demonstrating the need of a rather complex algorithm described and proved in Section 4. The formal proof and its iterations are described in Section 5. Section 6 concludes and gives a few perspectives.

2. Radix-2 Average Algorithms

How to compute the average of two FP numbers is a problem known for decades. It has been thoroughly studied by Sterbenz [11], among some examples called "carefully written programs".

This study is especially interesting as Sterbenz did not fully give a correct program: he specified what it is required to do, such as symmetry, gave hints about how to circumvent overflow and advised scaling to prevent underflow. The proposed algorithms are then

- $(x \oplus y) \oslash 2$, which is very accurate, but may overflow when x and y share the same sign.
- $(x \oslash 2) \oplus (y \oslash 2)$ is also accurate, and may underflow. Moreover, it requires an additional operation.
- $x \oplus ((y \ominus x) \oslash 2)$ is less accurate than the first one, but it does not overflow if x and y have the same sign.

A corresponding algorithm has been proved by Boldo [12] to guarantee both the accuracy and Sterbenz's requirements. This gives a long program as a full sign study is needed. Moreover, Boldo has proposed another algorithm that computes the correctly-rounded average of two *binary64* FP numbers:

```
double average(double C, double x, double y) {
  if (C <= abs(x))
    return x/2+y/2;
  else
    return (x+y)/2;
}
```

with a constant C that can be chosen between 2^{-967} and 2^{970}. This program was formally proved to provide the correct result even in case of underflow and to never create spurious overflow [12].

A comparable algorithm is given by Goualard [13]. The problem is slightly different as he wants to compute the midpoint of an interval, which does not have the same meaning when exceptional values or exceptional intervals are given as input. He also lists several algorithms used in practice and their defects. In addition to the previous ones, he cites:

- $(x \ominus (x \oslash 2)) \oplus (y \oslash 2)$, that is less accurate when underflow happens, but behaves well with overflow.
- many other formulas using directed rounding modes such as $\triangle(\triangle(x/2) + \triangle(y/2))$, where $\triangle$ denotes the rounding towards $+\infty$. This formula does not have the containment property: the midpoint may be outside the input interval.

A last algorithm, namely $x \ominus (x \ominus y) \oslash 2$, is given by Kornerup *et al.* [14]. It is proved to provide the correct rounding, but under strong hypotheses: either $0 \leq y \leq x \leq 2y$ or $2y \leq x \leq y \leq 0$.

The collection of possible algorithms is therefore large. Nevertheless, these simple algorithms without any test do not compute the correct rounding when FP computations are done in radix 10, as explained in the next section.

3. Unsuccessful Radix-10 Average Algorithms

As explained above, there are many basic formulas which mathematically compute the average of two FP numbers. We studied these straight-line formulas, but we found counter-examples for each one. Therefore, we had to create a more complex algorithm described in Section 4. All counter-examples given in this part use a radix-10 FP format with four significant digits. They can easily be generalized to any higher precision.

3.1. Formulas based on $(x \oplus y) \oslash 2$

The most naive formula is $(x \oplus y) \oslash 2$, which provides a correct rounding in radix 2, provided that no overflow occurs.

In radix 10, the result of the division of an FP number by 2 is either correct (when the mantissa is even) or a midpoint (when the mantissa is odd). The tie-breaking rule of the rounding mode is therefore used to choose the direction of the rounding in this latest case. Using this fact, it is easy to build counter-examples: if an FP number is negligible compared to the other one in the addition and the division by 2 needs to be rounded, the result will be incorrect. For instance, let $x = 3001 \times 10^{10}$ and $y = 1000 \times 10^{-10}$. The rounded sum of x and y is equal to x because y is too small, and the division of x by 2 needs the tie-breaking rule to be rounded ($x/2 = 1500.5 \times 10^{10}$). With a tie broken to even (here towards $-\infty$), we get $(x \oplus y) \oslash 2 = 1500 \times 10^{10}$. But since y is positive, the exact value $(x + y)/2$ is slightly greater than the midpoint 1500.5×10^{10}, so the rounding should have been towards $+\infty$ and should have produced the result 1501×10^{10}.

Let us try to improve this formula by replacing the division by 2 by a multiplication by 5 followed by a division by 10. Indeed, in radix 10 the division by 10 is always exact (except in case of underflow), since we just have to reduce the exponent by 1. A formula using this method may be $(x \oplus y) \otimes 5 \oslash 10$. But the previous counter-example still is problematic with this formula, since the error comes from the addition (y is absorbed in x). It seems that all straight-line algorithms involving only $x \oplus y$ will fail the same way.

3.2. Formulas based on $(x \oslash 2) \oplus (y \oslash 2)$

In order to circumvent the $x \oplus y$, we now try another formula: $(x \oslash 2) \oplus (y \oslash 2)$. This algorithm also works well in radix 2, except when underflow occurs. Unfortunately, this algorithm does not avoid the previous issue. If we consider the previous counter-example $x = 3001 \times 10^{10}$ and $y = 1000 \times 10^{-10}$ of Section 3.1, then $x/2$ is a midpoint and $(x \oslash 2) \oplus (y \oslash 2)$ does not produce the correct rounding.

Let us assume we have an FMA operator available, it is now possible to get rid of one division to have only two roundings: $\circ(x \times 0.5 + (y \oslash 2))$. The division of y by 2 may be inexact, but the other operations (division of x and addition) are done without intermediate rounding, thanks to the FMA operator. This means that the sign of y will impact the rounding of $x/2$ in the previous counter-example. But we can still find cases where the result is incorrect: let $x = 2001 \times 10^{10}$ and $y = 2001 \times 10^{8}$. We get $y \oslash 2 = 1000 \times 10^{8}$, the exact result of this division is a midpoint which has been rounded. This value is then added to $x/2 = 1000.5 \times 10^{10}$, and this gives the value 1010.5×10^{10}, which is rounded to 1010×10^{10}. The expected result is the rounding of 1010.505×10^{10}, which is 1011×10^{10}.

Despite these cases where this formula gives a wrong result, it is a basic block of our algorithm (see Section 4). In addition to this block, we need a test in order to apply this formula only when correct.

4. Algorithm for the Average of Decimal FP Numbers

Now let us focus on our solution, presented in Algorithm 1 and let us prove its correctness. We consider for now radix-10 FP numbers with unbounded exponent range, and a precision greater than 1. This generic FP format is denoted by $\mathbb{F}$. The proof with gradual underflow is described later on in Section 5. We also assume that an FMA is available, which is common on processors with a decimal FP unit.

```
1 Function Average10(x, y)
2     (a, b) = TwoSum(x, y)
3     if ∘(a × 0.5 − (a ⊘ 2)) = 0 then
4         return ∘(b × 0.5 + (a ⊘ 2))
5     else
6         return ∘(a × 0.5 + b)
```

Algorithm 1: Decimal Average Algorithm

The FMA operator is indeed used at Lines 3, 4, and 6. This algorithm also relies on the `TwoSum` operator [15], [16], which computes the sum (a in the following formulas) of two FP numbers and the rounding error (b). It is known that $(a, b) = \mathrm{TwoSum}\,(x, y) \implies x+y = a+b$, as the error of an FP addition can always be exactly represented with an FP number (and `TwoSum` exactly computes it). The average of the input of `TwoSum` is then equal to the average of its output. Since b is the rounding error of a, we also have $|b| \leq \frac{\mathrm{ulp}\,(a)}{2}$.

The average computed by the algorithm is then the average between a and b (the output of `TwoSum`, Line 2). Using the hypothesis given by the `TwoSum` function, we prove that in both cases of the test at Line 3, this algorithm computes the correctly rounded average. This test checks whether $a/2$ is exactly representable in the FP format $\mathbb{F}$, by computing the subtraction between the exact value $a/2$ and the rounded value $a \oslash 2$.

Note that we suppose in this section that a is positive, the generalization to negative values is done by symmetry (see Section 5.3.2).

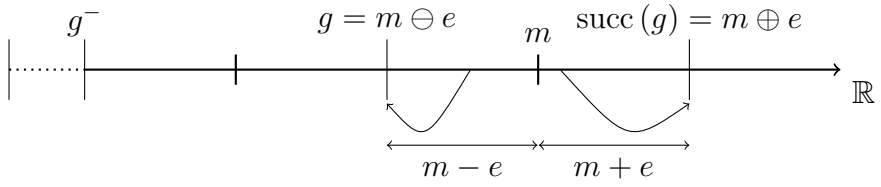
Let us detail the proof with first a few intermediate lemmas.

Lemma 1. *Let* $m = g + \frac{1}{2} \times \mathrm{ulp}\,(g)$ *with* $g \in \mathbb{F}$, $m > 0$ *and* $0 < e \leq \frac{\mathrm{ulp}\,(g)}{2}$.

- $m \ominus e = g$
- $m \oplus e = \mathrm{succ}\,(g)$

Proof. The proof relies on the fact that m is the midpoint between g and $\mathrm{succ}\,(g)$ and that e is positive and small enough. This can be explained more easily by looking at the respective values of the various variables as shown in Figure 1. □

Figure 1. Layout of the values of Lemma 1.



This lemma does not hold when m is a negative power of the radix. A similar lemma could be stated without assuming that $m > 0$ but this special case has to be removed or handled. For the sake of simplicity, we preferred this limited version, and in the following proof, we used this lemma only with $m = a > 0$.

Lemma 2 states the properties of an FP number that cannot be exactly divided by 2:

Lemma 2. *Let* $x \in \mathbb{F}$.

$$x/2 \notin \mathbb{F} \implies |M_x| \geq 2 \times 10^{p-1} + 1 \wedge \mathrm{odd}\,(M_x)$$

Proof. Let $x \in \mathbb{F}$ such as $x/2 \notin \mathbb{F}$.

Assume by contraposition that $\mathrm{even}\,(M_x)$, so $\frac{M_x}{2} \in \mathbb{Z}$.
Let $n = \frac{M_x}{2}$, $\frac{x}{2} = \frac{M_x}{2} \times 10^{E_x} = n \times 10^{E_x}$. We have $x \in \mathbb{F}$, so $|M_x| < 10^p$, which implies $|n| < 10^p$. We can deduce $\frac{x}{2} \in \mathbb{F}$, because $\frac{x}{2} = n \times 10^{E_x}$ with:

$$n \in \mathbb{Z} \wedge E_x \in \mathbb{Z} \wedge |n| < 10^p$$

Therefore, the first assumption $\mathrm{even}\,(M_x)$ was false, because it contradicts $\frac{x}{2} \notin \mathbb{F}$, so we have $\mathrm{odd}\,(M_x)$.

Now, assume that $|M_x| < 2 \times 10^{p-1}$.
We have $\mathrm{odd}\,(M_x)$, so $\mathrm{even}\,(M_x - 1)$ and $\frac{M_x-1}{2} \in \mathbb{Z}$. Let $n = 10 \times \frac{M_x}{2}$, so $\frac{x}{2} = n \times 10^{E_x-1}$. With the equality $\frac{M_x}{2} = \frac{M_x-1}{2} + \frac{1}{2}$, we prove that $n \in \mathbb{Z}$:

$$n = 10 \times \left(\frac{M_x - 1}{2} + \frac{1}{2}\right) = 10 \times \frac{M_x - 1}{2} + 5$$

Moreover, with the assumption $|M_x| < 2 \times 10^{p-1}$, we bound the value of n: $\left|\frac{M_x}{2}\right| < 10^{p-1}$, so $|n| < 10^p$. So $\frac{x}{2} \in \mathbb{F}$, because $\frac{x}{2} = n \times 10^{E_x-1}$ with:

$$n \in \mathbb{Z} \wedge E_x - 1 \in \mathbb{Z} \wedge |n| < 10^p$$

Again, by contradiction, the assumption $|M_x| < 2 \times 10^{p-1}$ is false, so $|M_x| \geq 2 \times 10^{p-1}$.

We have $\mathrm{odd}\,(M_x) \wedge |M_x| \geq 2 \times 10^{p-1}$, but $\mathrm{even}\,(2 \times 10^{p-1})$, so $|M_x| > 2 \times 10^{p-1}$, and finally, $\mathrm{odd}\,(M_x) \wedge |M_x| \geq 2 \times 10^{p-1} + 1$ □

Let us now state the main theorem (nearly) stating the correctness of Algorithm 1.

Theorem 1. *Let $a \in \mathbb{F}$ and $b \in \mathbb{F}$ such that $|b| \leq \frac{1}{2} \times \mathrm{ulp}\,(a)$ and $a > 0$:*

- *If $\frac{a}{2} \in \mathbb{F}$:*

$$\circ\left(\frac{a+b}{2}\right) = \circ(b \times 0.5 + a \oslash 2)$$

- *If $\frac{a}{2} \notin \mathbb{F}$:*

$$\circ\left(\frac{a+b}{2}\right) = \circ(a \times 0.5 + b)$$

Proof. Let $a \in \mathbb{F}$ and $b \in \mathbb{F}$ such that $|b| \leq \frac{1}{2} \times \mathrm{ulp}\,(a)$ and $a > 0$.

- When $b = 0$:

$$\circ\left(\frac{a}{2} + b\right) = \circ\left(\frac{a}{2} + \frac{b}{2}\right) = \circ\left(\frac{a}{2}\right)$$

  So we can assume that $b \neq 0$ in the following of the proof.

- First case: $\frac{a}{2} \in \mathbb{F}$

$$\circ\,(b \times 0,5 + a \oslash 2) = \circ\left(\frac{b}{2} + \frac{a}{2}\right) = \circ\left(\frac{a+b}{2}\right)$$

- Second case: $\frac{a}{2} \notin \mathbb{F}$
  With the equality $\frac{M_a}{2} = \frac{M_a-1}{2} + \frac{1}{2}$, we have $\frac{a}{2} = \left(\frac{M_a-1}{2} + \frac{1}{2}\right) \times 10^{E_a}$.

According to Lemma 2, we have $\mathrm{odd}\,(M_a)$, so $\mathrm{even}\,(M_a - 1)$, and $\frac{M_a-1}{2} \in \mathbb{Z}$. This lemma also tells $|M_a| \geq 2 \times 10^{p-1} + 1$, so:

$$\begin{aligned} & 2 \times 10^{p-1} + 1 \leq |M_a| < 10^p \\ \Longrightarrow\ & 2 \times 10^{p-1} \leq |M_a - 1| < 10^p + 1 \\ \Longrightarrow\ & 10^{p-1} \leq \left|\frac{M_a - 1}{2}\right| < \frac{10^p + 1}{2} \\ \Longrightarrow\ & 10^{p-1} \leq \left|\frac{M_a - 1}{2}\right| < 10^p \end{aligned}$$

Let $c = \frac{M_a-1}{2} \times 10^{E_a}$, $c \in \mathbb{F}$ because:

$$\frac{M_a - 1}{2} \in \mathbb{Z} \wedge E_a \in \mathbb{Z} \wedge \left|\frac{M_a - 1}{2}\right| < 10^p$$

We also have $\mathrm{ulp}\,(a) = \mathrm{ulp}\,(c)$ because $E_a = E_c$, $10^{p-1} \leq M_a < 10^p$, and $10^{p-1} \leq M_c < 10^p$.
We can rewrite $\frac{a}{2}$ as:

$$\frac{a}{2} = c + \frac{1}{2} \times 10^{E_a} = c + \frac{1}{2} \times \mathrm{ulp}\,(c)$$

- If $b > 0$, we have: $0 < b \leq \frac{1}{2} \times \mathrm{ulp}\,(a)$. So $0 < \frac{b}{2} \leq \frac{1}{2} \times \mathrm{ulp}\,(a)$. Therefore, according to Lemma 1:

$$\circ\left(\frac{a}{2} + b\right) = \circ\left(\frac{a}{2} + \frac{b}{2}\right) = \mathrm{succ}\,(c)$$

- If $b < 0$, we have: $0 < -b \leq \frac{1}{2} \times \mathrm{ulp}\,(a)$. So $0 < -\frac{b}{2} \leq \frac{1}{2} \times \mathrm{ulp}\,(a)$. Therefore, with Lemma 1 again:

$$\circ\left(\frac{a}{2} - (-b)\right) = \circ\left(\frac{a}{2} - \left(-\frac{b}{2}\right)\right) = c$$

So, in every case, we have:

$$\circ\left(\frac{a}{2} + b\right) = \circ\left(\frac{a}{2} + \frac{b}{2}\right)$$

□

There is left to prove that the test $\circ(a \times 0.5 - (a \oslash 2)) = 0$ corresponds to $a/2 \in \mathbb{F}$. This is indeed the case with an unbounded exponent range as both are equivalent to $a/2 = a \oslash 2$.

Note that, at Line 6 of the algorithm, b is not divided by 2. In practice, we are only interested in the sign of b. Therefore, we may use either b or $b \oslash 2$. We choose to use b for two reasons. First, it saves an FP operation. Second, when considering gradual underflow, it becomes crucial as a small b may lead to have $b \oslash 2 = 0$ and to lose the sign information.

5. Formal Proof

All lemmas and theorem of Section 4 have been formally proved using the Coq proof assistant [8]. Coq comes with both a specification language and a tactic language to perform proofs in an interactive way.

This section is organized as follows. Section 5.1 presents the Flocq library and its formalization of FP numbers. To fully explain our main theorem, Section 5.2 presents the formalized algorithm: this is straightforward as it only requires knowledge of Flocq's notations to define FP algorithms in a given rounding mode. Section 5.3 emphasizes on the several generalizations of our algorithm proof and how the proof assistant has helped us to generalize the correctness proof of the algorithm in several ways.

5.1. Overview of the Flocq library

We use Flocq, a library of FP arithmetic written in Coq, that contains most basic technical results we needed to achieve our formal proofs [9], [10].

Flocq provides an abstract representation of FP numbers. FP numbers in a given format (such as *binary32* or *decimal64*) are just a subset of real numbers `R`. FP formats are intended to characterize numbers of the form $m \cdot \beta^e$ (with m and e integers). Several formats are available, such as fixed-point and floating-point formats. The radix will here always be 10.

The two formats we use are `FLX` and `FLT`. The `FLX` format corresponds to FP numbers with unbounded exponents. It depends on the precision p and requires $|m| < \beta^p$. The `FLT` format takes gradual underflow into account. It therefore depends on both the precision p and the minimal exponent $e_{\min}$. It requires both $|m| < \beta^p$ and $e \geq e_{\min}$.

Given a format, the common rounding modes are formally defined (such as towards $+\infty$ or towards zero). Rounding to nearest is more interesting: the IEEE-754 roundings to nearest, with tie-breaking to even or away from zero are available, but a generic rounding to nearest is also available. More precisely, we have a rounding to nearest, which rounds to the nearest FP number when unique, and chooses depending on a tie-breaking rule denote by τ when on a midpoint. This allows us to do a single proof for both IEEE-754 roundings to nearest, but also for all rounding to nearest with any tie-breaking-rule (tie-breaking to odd, tie-breaking towards zero, or any combination). See also Section 5.3.2.

5.2. Formalization of the algorithm

In the second line of the average Algorithm 1, a `TwoSum` is called to transform the inputs. Although there is an existing formalization of the `TwoSum` algorithm in Coq, we are not interested here in relating it to our formalization. Indeed, the `TwoSum` algorithm has been formalized in radix 2 in *PFF*, another Coq library of FP numbers [17]. Hence, integrate `TwoSum` in our development would require to write a radix-10 Flocq version of the algorithm. It raises the general question of interoperability between Coq libraries, known to be difficult to handle.

Actually, we know that the inputs of our algorithm are outputs of a `TwoSum`. The only thing we need to assume is that the absolute value of the second input is less than or equal to

half of the ulp of the first one (*i.e.* `Rabs y <= ulp x / 2`). As explained in Section 4, this hypothesis is crucial to prove that our algorithm provides a correctly-rounded result.

In Coq, we define the Algorithm 1 as:

```
Definition average10 (x y : R) :=
  if (Req_bool (round (x/2 - round (x/2))) 0)
    then round (y/2 + round (x/2))
    else round (x/2 + y).
```

Note that `Req_bool u v` is a predicate which returns true if u is equal to v and false otherwise. Our aim is to prove that this algorithm provides a correctly-rounded result, which corresponds to the following theorem (`format x` means that $x \in \mathbb{F}$ in the given format):

```
Theorem average10_correct :
  forall x y, format x → format y → Rabs y <= ulp x / 2 →
   average10 x y = round ((x+y)/2).
```

We define `format` and `round` depending on the chosen format. We have first proved this theorem in the `FLX` format (meaning with an infinite exponent range) assuming the precision is greater than 1. We have also proved that it holds with gradual underflow (the `FLT` format), see Section 5.3.1.

5.3. Generalization of the results

On the one hand, proof assistants could be difficult to use as every detail must be justified. On the other hand, they may assist the user with the generalization of results: removing hypotheses for instance. We provide in this section three different generalizations of the correctness proof of our average algorithm.

5.3.1. Gradual underflow. The first generalization is underflow's handling. The formal proof has first been done in the Flocq's `FLX` format (FP numbers with unbounded exponents). While trying to perform the reasoning with the `FLT` format (taking gradual underflow into account), we had to patch the Coq proofs to show how the algorithm remains correct with gradual underflow. We distinguished two cases:

- if $b = 0$, then a might be a subnormal number. In the `FLX` format, $a \neq 0 \Rightarrow o(a) \neq 0$, which is not necessarily true in the `FLT` format when a is subnormal. Hence, the test we make in Line 3 of Algorithm 1 does not check whether $a/2$ is exactly representable in the `FLT` format.
  Nevertheless, as $b = 0$, it does not matter which executed branch is taken. The returned value is indeed correctly-rounded in both cases.
- if $b \neq 0$, as $|b| \leq \frac{\mathrm{ulp}\,(a)}{2}$, a is necessarily greater than or equal to $10^{p+e_{min}}$ and hence both a, $a/2$, and $a \oslash 2$ are normal FP numbers. As a consequence, the test we make at Line 3 of Algorithm 1 really checks whether $a/2$ is exactly representable and the previous proof holds.
  Moreover, when b is subnormal, its sign may be lost when computing $b/2$ (when $b = \mathrm{succ}\,(0)$). However, as explained in Section 4, as b is not divided by 2 at Line 6 of Algorithm 1, we avoid this problem.

5.3.2. Symmetry. We consider a generic tie-breaking rule τ. We proved that, for all τ, if we compute with this tie-breaking rule, then the result is the correctly-rounded average with the same tie-breaking rule. We assumed until then that $a > 0$. We of course want to generalize this to any a.

Given a tie-breaking rule τ, if we consider the function that associates to x the value $-\circ^{\tau}(-x)$, it is a rounding to nearest with another tie-breaking rule, that we denote $\tilde{\tau}$. For the IEEE-754 tie-breaking rules, this is useless as both are symmetric.

With a negative a, the application of the theorem to $-a$, $-b$ and $\tilde{\tau}$ proves that $\mathrm{average10}^{\tilde{\tau}}(-a,-b) = \circ^{\tilde{\tau}}((a+b)/2)$. By easy manipulations and the definition of $\tilde{\tau}$, we get $\mathrm{average10}^{\tau}(a,b) = \circ^{\tau}((x+y)/2)$ for a negative a. When $a = 0$, we easily prove that $b = 0$ and we compute the correct 0.

5.3.3. Even radix. At last, we formally proved that Algorithm 1 remains correct for any even radix β. It is possible to put the radix as a parameter of the proof environment and then to replay the formal proofs to see where it breaks. As we never use specific properties of 10 apart from its parity, most pieces of the proofs remain the same. Nonetheless, we had to handle rather simple proof goals about integer arithmetic that were automatically done by Coq for $\beta = 10$.

6. Conclusion and Perspectives

Correctly rounded algorithms to compute the average of two FP numbers exist in radix-2 arithmetic. Unfortunately, they are not correct in radix-10 arithmetic. In this paper, we have shown that various naive possibilities are unsuccessful and do not return a correctly-rounded average. Then, we have provided and proved an algorithm averaging two decimal FP numbers with correct rounding. Our algorithm may seem costly compared to straight-line formulas, due to the 6 flops of the *TwoSum* algorithm. However, it is quite short, easy to understand and correct for any precision $p \geq 2$. Even if its correctness proof is rather technical, it is quite readable. Furthermore, the algorithm remains correctly-rounded if we take gradual underflow into account. A surprising point is that the structure of the proof stays the same and that the modifications were minor.

Moreover, our algorithm still works for any even radix. The algorithm provided by Boldo [12] for the radix-2 case exploits the fact that there is a division by 2 in the average computation. We think it is possible to provide a similar correct algorithm to compute $\frac{x+y}{10}$ in radix-10 arithmetic, or more generally $\frac{x+y}{\beta}$ with an even radix β. In contrast, we compute the average of two FP numbers without exploiting specific properties of 10 (apart from its parity). That is why our algorithm can be generalized to any even radix β.

In addition, we have formalized the algorithm in the Coq proof assistant. A comprehensive proof of its correctness has been provided in order to increase the confidence in our results. We first proved its correctness using the `FLX` format of Flocq, which assumes the exponents are unbounded. Then, it has been easy to adapt the proofs using the `FLT` format, in which gradual underflow is taken into account. Furthermore, the correctness of the algorithm has been formally proved for any even radix. The formalization and the proof consist in about

700 lines of Coq code, which is rather concise and comparable in magnitude with the formal proof provided by Boldo for binary FP arithmetic.

A first perspective is overflow's handling. Provided that the *TwoSum* algorithm returns finite FP numbers, our algorithm does not overflow. However, a difficulty arises from the possible occurrence of spurious overflow in the *TwoSum* algorithm [18], and even more spurious in our case as $(a+b)/2$ may be finite when $a \oplus b$ is not. Nevertheless, overflow is easy to check *a posteriori*. Indeed, if *TwoSum* returns at least one special FP number (NaN, $+\infty$ or $-\infty$), our average algorithm returns either NaN or an infinity.

Another possible perspective is the generalization of the algorithm to compute the average $\frac{X_1+X_2+\cdots+X_n}{n}$ of n decimal FP numbers X_1, X_2, ..., X_n. It is a more difficult problem and it is even not always possible to compute a correctly rounded summation of n FP numbers for $n > 3$ [14], except for the use of a long accumulator [19].

The raise of decimal FP arithmetic creates the need for certifying decimal FP programs. In the case of our decimal average, we could have written a C program, but we could not have formally certified it. There are available tools to formally verify C programs, such as Frama-C and Why3 [20], [21] that were used for the radix-2 average algorithm [12]. However, they do not support decimal arithmetic. There is support neither in the annotation language, nor in the various translations and computations of the tools to handle decimal FP numbers and operations. If we choose to stay within Coq, the Flocq library offers support for vectors of bits for binary FP numbers (so that overflow and exceptional values may be taken into account), but not for decimal numbers. This can be developed, but it is a cumbersome unrewarding task, that probably requires critical decimal applications to be completed.

Acknowledgment

This research was partially supported by Labex DigiCosme (project ANR-11-LABEX-0045-DIGICOSME) operated by ANR as part of the program "Investissement d'Avenir" Idex Paris-Saclay and by FastRelax ANR-14-CE25-0018-01.

References

[1] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Comput. Surv.*, vol. 23, no. 1, pp. 5–48, Mar. 1991. [Online]. Available: http://doi.acm.org/10.1145/103162.103163

[2] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres, *Handbook of Floating-Point Arithmetic*. Birkhäuser, 2010.

[3] "IEEE standard for binary floating-point arithmetic," *ANSI/IEEE Std 754-1985*, 1985.

[4] "IEEE standard for floating-point arithmetic," *IEEE Std 754-2008*, Aug 2008.

[5] V. A. Carreño and P. S. Miner, "Specification of the IEEE-854 floating-point standard in HOL and PVS," in *HOL95: 8th International Workshop on Higher-Order Logic Theorem Proving and Its Applications*, Aspen Grove, UT, Sep. 1995.

[6] D. M. Russinoff, "A mechanically checked proof of IEEE compliance of the floating point multiplication, division and square root algorithms of the AMD-K7 processor," *LMS Journal of Computation and Mathematics*, vol. 1, pp. 148–200, 1998.

[7] J. Harrison, "Formal verification of floating point trigonometric functions," in *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, Austin, Texas, 2000, pp. 217–233.

[8] The Coq Development Team, *The Coq Proof Assistant Reference Manual v8.6*, 2016.

[9] S. Boldo and G. Melquiond, "Flocq: A unified library for proving floating-point algorithms in Coq," in *20th IEEE Symposium on Computer Arithmetic*, E. Antelo, D. Hough, and P. Ienne, Eds., Tübingen, Germany, 2011, pp. 243–252.

[10] ——, *Computer Arithmetic and Formal Proofs*. ISTE Press - Elsevier, Dec. 2017.

[11] P. H. Sterbenz, *Floating point computation*. Prentice Hall, 1974.

[12] S. Boldo, "Formal Verification of Programs Computing the Floating-Point Average," in *17th International Conference on Formal Engineering Methods*, ser. Lecture Notes in Computer Science, M. Butler, S. Conchon, and F. Zaïdi, Eds., vol. 9407. Paris, France: Springer International Publishing, Nov. 2015, pp. 17–32. [Online]. Available: https://hal.inria.fr/hal-01174892

[13] F. Goualard, "How do you compute the midpoint of an interval?" *ACM Trans. Math. Softw.*, vol. 40, no. 2, pp. 11:1–11:25, Mar. 2014. [Online]. Available: http://doi.acm.org/10.1145/2493882

[14] P. Kornerup, V. Lefevre, N. Louvet, and J.-M. Muller, "On the computation of correctly rounded sums," *IEEE Transactions on Computers*, vol. 61, no. 3, pp. 289–298, March 2012.

[15] T. J. Dekker, "A floating-point technique for extending the available precision," *Numerische Mathematik*, vol. 18, no. 3, pp. 224–242, 1971.

[16] D. E. Knuth, *The Art of Computer Programming*, 3rd ed. Reading, MA, USA: Addison-Wesley, 1998, vol. 2.

[17] M. Daumas, L. Rideau, and L. Thery, "A Generic Library for Floating-Point Numbers and Its Application to Exact Computing," in *Theorem Proving in Higher Order Logics*, ser. Lecture Notes in Computer Science. Edinburgh, United Kingdom: Springer Berlin / Heidelberg, 2001, pp. 169–184. [Online]. Available: https://hal.archives-ouvertes.fr/hal-00157285

[18] S. Boldo, S. Graillat, and J.-M. Muller, "On the robustness of the 2Sum and Fast2Sum algorithms," *ACM Transactions on Mathematical Software*, vol. 44, no. 1, Jul. 2017. [Online]. Available: https://hal-ens-lyon.archives-ouvertes.fr/ensl-01310023

[19] U. W. Kulisch, *Advanced arithmetic for the digital computer - design of arithmetic units*. Springer, 2002.

[20] S. Boldo and J.-C. Filliâtre, "Formal verification of floating-point programs," in *Proceedings of the 18th IEEE Symposium on Computer Arithmetic*, P. Kornerup and J.-M. Muller, Eds., Montpellier, France, Jun. 2007, pp. 187–194. [Online]. Available: http://www.lri.fr/~filliatr/ftp/publis/caduceus-floats.pdf

[21] S. Boldo, "Deductive formal verification: How to make your floating-point programs behave," Thèse d'habilitation, Université Paris-Sud, Oct. 2014. [Online]. Available: http://www.lri.fr/~sboldo/files/hdr.pdf