# Formal Verification of Programs Computing the Floating-Point Average

Sylvie Boldo

# Formal Verification of Programs Computing the Floating-Point Average*

Sylvie Boldo

Inria,
LRI, bâtiment 650,
Université Paris-Sud,
F-91405 Orsay Cedex, France
Sylvie.Boldo@inria.fr

**Abstract.** The most well-known feature of floating-point arithmetic is the limited precision, which creates round-off errors and inaccuracies. Another important issue is the limited range, which creates underflow and overflow, even if this topic is dismissed most of the time. This article shows a very simple example: the average of two floating-point numbers. As we want to take exceptional behaviors into account, we cannot use the naive formula `(x+y)/2`. Based on hints given by Sterbenz, we first write an accurate program and formally prove its properties. An interesting fact is that Sterbenz did not give this program, but only specified it. We prove this specification and include a new property: a precise certified error bound. We also present and formally prove a new algorithm that computes the correct rounding of the average of two floating-point numbers. It is more accurate than the previous one and is correct whatever the inputs.

## 1 Introduction

Floating-point 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. Which numbers and how operations behave on them is standardized in the IEEE-754 standard [13] of 1985, which was revised in 2008 [14].

Computer arithmetic [11], is mostly known (if known at all) to be inaccurate, as only a finite number of digits is kept for the mantissa. A more ignored fact is that only a finite number of digits is kept for the exponent. This creates the underflow and overflow exceptions, that are often dismissed, even by floating-point experts. We are here mostly interested in handling overflow, even if underflow will also play its part.

* This work was supported by both the VERASCO (ANR-11-INSE-003) and the FastRelax (ANR-14-CE25-0018-01) projects of the French National Agency for Research (ANR).

The chosen example is very simple: how to compute the average of two floating-point numbers:

$$\frac{x+y}{2}.$$

The naive formula `(x+y)/2` is quite accurate, but may fail due to overflow, even if the correct result is in the range. For example, consider the maximum floating-point number $M$, then `(M+M)/2` overflows while the correct result is $M$. This problem has been known for decades and has been thoroughly studied by Sterbenz [17], among some examples called "carefully written programs".

This study is especially interesting as Sterbenz does not fully give a correct program: he specified what it is required to do, such as symmetry and gives hints about how to circumvent overflow. We are interested in writing and proving the behavior of such a program, that produces an accurate result without overflowing. And of course, we look for an improved algorithm which would give a correct result, also without overflowing.

All the theorems stated in this article correspond to Coq theorems. This development, meaning the C codes and full proofs are available from the following web page

`https://www.lri.fr/~sboldo/research/`

The outline of this article is as follows. Basics about floating-point arithmetic are given in Section 2. The methodology of the verification, and what is supposed to be verified are in Section 3. The formal proofs about the algorithms are described in Section 4. The annotations of the C programs and the corresponding proofs, including overflow, are in Section 5. Section 6 concludes and gives a few perspectives.

## 2 Basics about Floating-Point Arithmetic

The IEEE-754 standard [13] of 1985, which was revised in 2008 [14] describes the floating-point formats, numbers and roundings and all modern processors comply with it. We adopt here the level 3 vision of the standard: we do not consider bit strings, but the representation of floating-point data. The format will then be ($\beta$, $p$, $e_{min}$, $e_{max}$), where $e_{min}$ and $e_{max}$ are the minimal and maximal unbiased exponents, $\beta$ is the radix (2 or 10), and $p$ is the precision (the number of digits in the significand).

In that format, a floating-point number is then either a triple $(s, e, m)$, or an exceptional value: $\pm\infty$ or a $NaN$ (Not-a-Number). For non-exceptional values, meaning the triples, we have additional conditions: $e_{min} \leq e \leq e_{max}$ and the significand $m$ has less than $p$ digits. The triple can be seen as the real number with value

$$(-1)^s \times m \times \beta^e.$$

We will consider $m$ as an integer and we therefore require that $m < \beta^p$. The other possibility is that $m$ is a fixed-point number smaller than $\beta$. In this

setting, the common IEEE-754 formats are `binary64`, which corresponds to (2, 53, −1074, 971) and `binary32`, which corresponds to (2, 24, −149, 104).

Non-exceptional values give a discrete finite set of values, which can be represented on the real axis as in Figure 1. Floating-point numbers having the same exponent are in a binade and are at equal distance from one to another. This distance is called the unit in the last place (ulp) as it is the intrinsic value of the last bit/digit of the significand of the floating-point number [15]. When going from one binade to the next, the distance is multiplied by the radix, which gives this strange distribution. Around zero, we have the numbers having the smallest exponent and small mantissas, they are called subnormals and their ulp is that of the smallest normal number.

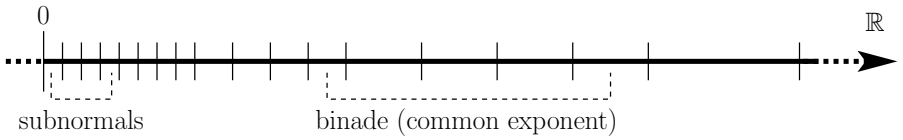


**Fig. 1.** Distribution of the floating-point numbers over the real axis.

Floating-point arithmetic tries to mimic real arithmetic but, in many cases, the exact result of an operation on two floating-point numbers is not a floating-point number. For example, in `binary64`, 1 and $2^{-53}$ are floating-point numbers, but $1 + 2^{-53}$ is not, as it would require 54 bits for the significand. The value therefore needs to be rounded. The IEEE-754 standard defines 5 rounding modes. We will here only use the default rounding mode: rounding to nearest ties to even, denoted by $\circ$. Rounded addition will be denoted by $\oplus$, rounded subtraction by $\ominus$ and rounded division by $\oslash$.

The main rule of the IEEE standard of floating-point computation for basic operations is the following one, called *correct* rounding: each operation gives the same result as if it was first performed with infinite precision, and then rounded to the desired format. This is a very strong mathematical property that has two essential consequences: portability and accuracy. It also implies that rounding is non-decreasing. A last property is the fact that division by the radix is an exact operation, provided the input is not subnormal.

For some ugly details, as for the difference between signaling and quiet NaNs, the sign of $0 - 0$ or the value of $\left(\sqrt{-0}\right)$, we refer the reader directly to the standard [14]. Other major references are an article by Goldberg [11] and the Handbook of Floating-Point Arithmetic [15].

## 3 Methodology and Desired Specification

### 3.1 Methodology

To give a high guarantee on our mathematical results and programs, we rely on formal methods. Floating-point arithmetic that has been formalized since 1989 in order to formally prove hardware components or algorithms [8, 16, 12]. We use the Flocq library [7], a formalization in Coq which offers a multi-radix and multi-precision formalization for various floating- and fixed-point formats (including floating-point with or without gradual underflow, meaning subnormals) with a comprehensive library of theorems.

Following the methodology described in [4, 5, 3], we use the Frama-C / Jessie / Why3 chain and the ACSL language to perform formal verification of C programs at the source-code level. Frama-C is an extensible framework which combines static analyzers for C programs, written as plug-ins, within a single tool. In this work, we use the Jessie plug-in for deductive verification. C programs are annotated with behavioral contracts written using the ANSI C Specification Language [1] which tries to be as near C statements as possible. The Jessie plug-in translates them to the Why3 verification platform [2]. Finally, the Why3 platform computes verification conditions from these programs, using traditional techniques of weakest preconditions, and emits them to a wide set of existing theorem provers, ranging from interactive proof assistants to automated theorem provers. In this work, we use the Coq proof assistant, the automated theorem prover Gappa [10] which uses interval arithmetic to prove properties that occur when verifying numerical applications, and the SMT prover Alt-Ergo [9].

### 3.2 Desired Specification

The first point we want to specify is the accuracy of the ideal average function. In principle, we would like an error less than half a unit in the last place, which corresponds to correct rounding. But this is very difficult to achieve while preventing overflow as noted by Sterbenz [17]. This requirement will be weakened to a few ulps for the first program, as long as several other properties are kept. More precisely, we require:

- the program never overflows,
- $average(x, y)$ is within a few ulps of $\frac{x+y}{2}$,
- $\min(x, y) \leq average(x, y) \leq \max(x, y)$,
- $average(x, y) = average(y, x)$,
- $average(-x, -y) = -average(x, y)$,
- $average(x, y)$ has the same sign as $\frac{x+y}{2}$,

Sterbenz specified two facts related to underflow. First, $average(x, y) = 0$ if and only if $y = -x$, except in case of underflow. Second, the program should not underflow unless $0 < \left|\frac{x+y}{2}\right| < \eta$, where $\eta = 2^{p-1+E_i}$ is the smallest normalized positive number. Our specifications are stronger than Sterbenz's and will be detailed in Section 4

This paper will formally prove the previous assumptions, and will determine and prove the accuracy of two programs: an accurate one based on Sterbenz's hints and a correct one. We will also weaken the underflow assumptions. For that, we will first need to write a correct program. Sterbenz suggested several ways to compute the average:

- $(x \oplus y) \oslash 2$, which is very accurate (see below for the error bound), 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 opposite signs.
- As for underflow, Sterbenz suggests a scaling. We will prove that it is useless.

On the internet, we found a reference to Sterbenz's book and a corresponding program in the user notes on Fortran programming[1]. An excerpt of this program is given:

```
real function average (x, y)
      real      x, y, zero, two, av1, av2, av3, av4
      logical   samesign
      parameter (zero = 0.0e+00, two = 2.0e+00)

      av1(x,y) = (x + y) / two
      av2(x,y) = (x / two) + (y / two)
      av3(x,y) = x + ((y - x) / two)
      av4(x,y) = y + ((x - y) / two)

[...definition of samesign...]

      if (samesign) then
        if (y .ge. x) then
          average = av3(x,y)
        else
          average = av4(x,y)
        endif
      else
        average = av1(x,y)
      endif
      return
      end
```

The problem is that this program is incorrect: it does not fulfill one of Sterbenz's requirement: $average(-x,-y) = -average(x,y)$. For example, consider the IEEE binary64 format and the values $x = -2^{53}$ and $y = -1.25$, then $average(-x,-y) = average(2^{53}, 1.25) = average4(2^{53}, 1.25) = 2^{52} + 1$, but $-average(x,y) = -average(-2^{53}, -1.25)$
$= -average3(-2^{53}, -1.25) = 2^{52}$. The reason is the test $y \geq x$ that should be $|y| \geq |x|$ to preserve the symmetry.

[1] http://www.ibiblio.org/pub/languages/fortran/ch4-9.html

## 4 Formal Proof of the Algorithms

This formal proof was done in the FLT format of the Flocq library [7]. This corresponds to a generic floating-point format with gradual underflow and no overflow. This may seem strange as we are mostly interested in overflow here, but overflow will be taken into account at the program level in the next Section. The reason is that underflow happens and can be handled, while overflow must be prevented. We will use radix 2 and rounding to nearest, ties to even $\circ$. We will denote by $p$ the precision and $E_i$ the minimal exponent, so that $2^{E_i}$ is the smallest positive floating-point number and $2^{p-1+E_i}$ is the smallest normal positive floating-point number.

We will here define the algorithms at the Coq level, and prove that they fulfill all the stated properties. For that, we will study all the algorithms in all the different cases. But we will be smarter as far as formal proofs are concerned: as $average4(x, y)$ is exactly $average3(y, x)$, we will only have to study the $average1$, $average2$ and $average3$ functions.

The interesting points here will be first the rounding error of the functions, and then the handling of underflow. In fact, we will prove that scaling is useless and that gradual underflow behaves perfectly. For sign correctness, the most problematic case is computing the average of 0 and $2^{E_i}$ which gives 0, even if computed correctly, as rounding is to nearest, ties to even. In the other cases, the sign is correct. We will therefore prove the following properties concerning underflow: if the average is exactly 0, then the algorithm returns 0. If the absolute value of the average is greater or equal to $2^{E_i}$, then the returned value is non-zero.

### 4.1 The average1 function

The $average1$ function is the simplest one, the naive one to compute the average.

```
Definition average1 (x y : R) :=round_flt(round_flt(x+y)/2).
```

That is to say $average1(x, y) = (x \oplus y) \oslash 2$.
In fact, this function is correct: it computes the correctly-rounded exact average.

**Theorem 1.** *For all floating-point numbers $x$ and $y$,*

$$(x \oplus y) \oslash 2 = \circ\left(\frac{x+y}{2}\right).$$

This holds in our algorithmic model without overflow.

*Proof.* Let us denote by $r$ the floating-point number $r = (x \oplus y) \oslash 2$. We have two sub-cases. When $|x + y| \leq 2^{p+E_i}$, then $x \oplus y$ has the minimal exponent, meaning a subnormal number or just above. It is therefore computed without error [11, 7]. Then, $r = (x \oplus y) \oslash 2 = (x + y) \oslash 2 = \circ\left(\frac{x+y}{2}\right)$.

When $|x+y| > 2^{p+E_i}$, then $|x \oplus y| \geq 2^{p+E_i}$. In this case, the division by 2 is exact as $x \oplus y$ is a normal number. Then $r = (x \oplus y) \oslash 2 = \frac{x \oplus y}{2} = \circ\left(\frac{x+y}{2}\right)$. $\square$

This correct rounding easily implies all the basic requirements on this function: $average1(x,y) = average1(y,x)$, $average1(-x,-y) = -average1(x,y)$, $average1(x,y)$ has the same sign as $\frac{x+y}{2}$. The fact that $average1(x,y)$ is between $\min(x,y)$ and $\max(x,y)$ is slightly more difficult as rounding is involved. The facts that $\frac{x+y}{2} = 0$ implies $average1(x,y) = 0$ and that $2^{E_i} \leq \left|\frac{x+y}{2}\right|$ implies $average1(x,y) \neq 0$ are also quite simple from basic floating-point properties of the rounding.

The rounding error here is very small as it is equivalent to only one rounding:

$$\left| average1(x,y) - \frac{x+y}{2} \right| \leq \frac{1}{2}\mathrm{ulp}\left(\frac{x+y}{2}\right).$$

An interesting point is the fact that this algorithm requires $x$ and $y$ to be of different signs in order to not overflow. But the preceding proofs do not require it and are valid (in our model without overflow) whatever the values of $x$ and $y$.

### 4.2 The average3 function

The $average3$ function is the more complex one, designed to prevent overflow when $x$ and $y$ share the same sign.

```
Definition average3 (x y : R) :=
      round_flt(x+round_flt(round_flt(y-x)/2)).
```

That is to say $average3(x,y) = x \oplus ((y \ominus x) \oslash 2)$.

Some of the basic requirements on this function are not difficult to prove: $average3(-x,-y) = -average3(x,y)$ is easy, so is the fact that $\frac{x+y}{2} = 0$ implies $average3(x,y) = 0$ and *vice versa*. Proving that $average3(x,y)$ has the same sign as $\frac{x+y}{2}$ is slightly more difficult.

The fact that $\min(x,y) \leq average3(x,y) \leq \max(x,y)$ is more difficult as many roundings are involved, including possible underflows. Without loss of generality, we assume that $x \leq y$. We have left to prove that $x \leq x \oplus ((y \ominus x) \oslash 2) \leq y$. The first inequality is simple: as $y \geq x$, then $y \ominus x \geq 0$, then $(y \ominus x) \oslash 2 \geq 0$. Then $x \leq x + ((y \ominus x) \oslash 2)$, and then $x \leq x \oplus ((y \ominus x) \oslash 2)$ as $x$ is in the floating-point format.

The difficult part is $x \oplus ((y \ominus x) \oslash 2) \leq y$. We split into two different subcases: either the rounding down of $y - x$, that is denoted by $\bigtriangledown(y-x)$ equals 0, or is positive. It is non-negative as $y \geq x$. When $\bigtriangledown(y-x) > 0$, this amounts to prove that $\circ\left(\frac{\circ(y-x)}{2}\right) \leq y - x$. When $y - x$ is in the format, this is trivial. When not, then we prove that $\circ\left(\frac{\circ(y-x)}{2}\right) \leq \bigtriangledown(y-x)$ by real number inequality manipulations, and the study of whether $\circ(y-x)$ is the rounding up or down. Then we have left to prove that $\bigtriangledown(y-x) \leq y - x$, which holds by definition. When $\bigtriangledown(y-x) = 0$, we have two cases: if $x = y$, the result holds. The only remaining case corresponds to $x < y$ and $\bigtriangledown(y-x) = 0$. As $x$ and $y$ are in the floating-point format, this is impossible as $y - x \geq 2^{E_i}$.

Another difficult point is that $2^{E_i} \leq \left|\frac{x+y}{2}\right|$ implies $average3(x,y) \neq 0$. This relies on the intermediate fact that, for all positive floating-point number $f$, then $\circ(f/2) < f$, even when underflow occur.

The proof of $\circ(f/2) < f$ for a positive floating-point number $f$ is a case split: if the exponent of $f$ is greater than $E_i$, then $\circ(f/2) = f/2 < f$. When $f = n2^{E_i}$ with $|n| < 2^p$, then we have left to prove that the integer rounding to nearest even of $n/2$ is strictly smaller than $n$. This is done by studying $n$: as $f > 0$, then $n \geq 1$. When $n = 1$, the result holds as $0 < 1$. For bigger $n$, we prove that this integer rounding is smaller than $n/2 + 1/2$ which is smaller than $n$.

Given this lemma, we assume that $x$ and $y$ share the same sign and that $2^{E_i} \leq \left|\frac{x+y}{2}\right|$. Without loss of generality, we assume $x \leq y$. We prove that $x \oplus ((y \ominus x) \oslash 2) \neq 0$ by contradiction. If a floating-point addition is zero, it is exact, therefore we know that $x + ((y \ominus x) \oslash 2) = 0$. Therefore $x = -((y \ominus x) \oslash 2) \leq 0$ as $y - x \geq 0$. We split into two subcases: if $x < 0$, we will prove the contrary of the previous lemma applied to $-x$. We have left to prove that $-x \leq -x \oslash 2$. But $-x = ((y \ominus x) \oslash 2)$. As $y \leq 0$, we have $y - x \leq -x$, then $y \ominus x \leq -x$, hence the result. Now, we assume that $x = 0$. Then the hypotheses are rewritten into $2^{E_i} \leq \left|\frac{y}{2}\right|$ and $y \oslash 2 = 0$, which is impossible as $2^{E_i}$ cannot round to 0. This property is the first to rely on the fact that $x$ and $y$ share the same sign.

The last property is the bound on the rounding error. The first subcase is when $\left|\frac{x+y}{2}\right|$ is exactly $\frac{2^{E_i}}{2}$. It corresponds to $x = 0$ and $y = \pm 2^{E_i}$ or *vice versa*. This very special case is not difficult, but must be studied differently from the general case. The general case corresponds to $average3(x,y)$ being non-zero. Then, following the idea of the previous subsection, we have either $\circ(y - x)$ or $\circ\left(\frac{\circ(y-x)}{2}\right)$ that is computed exactly. The final rounding error is therefore small and bounded as follows:

$$\left| average3(x,y) - \frac{x+y}{2} \right| \leq \frac{3}{2}\mathrm{ulp}\left(\frac{x+y}{2}\right)$$

provided $x$ and $y$ share the same sign.

The last missing property is the link between the values of $average3(x,y)$ and $average3(y,x)$. But they may not be equal, contrary to what happens with $average1$. Symmetry is achieved otherwise, by the sign study.

### 4.3 The average2 function

The $average2$ function is rather simple, even if it contains 2 multiplications. This is not a problem on recent architectures as the cost of addition and multiplication is nearly the same.

```
Definition average2 (x y : R) :=
    round_flt(round_flt(x/2) + round_flt(y/2)).
```

That is to say $average2(x,y) = (x \oslash 2) \oplus (y \oslash 2)$.

In fact, this function is correct provided $x$ is not too small: it computes the correctly-rounded exact average.

**Theorem 2.** *For all floating-point numbers $x$ and $y$ such that $2^{E_i+2p+1} \leq |x|$,*

$$(x \oslash 2) \oplus (y \oslash 2) = \circ\left(\frac{x+y}{2}\right).$$

This holds when $x$ is not too small. Consider for example $x = y = 2^{E_i}$. Then, the average is also $2^{E_i}$ while the algorithm returns 0.
Note also that the assumption $2^{E_i+2p+1} \leq |x|$ can be replaced by $2^{E_i+2p+1} \leq |y|$ by symmetry.

*Proof.* Let us denote by $r$ the floating-point number $r = (x \oslash 2) \oplus (y \oslash 2)$. As $x$ is big enough, we have $x \oslash 2 = \frac{x}{2}$. Then we have two subcases, depending on the magnitude of $y$.

If $|y| \geq 2^{p+E_i}$, then we have the same property: $y \oslash 2 = \frac{y}{2}$. Then $r = (x \oslash 2) \oplus (y \oslash 2) = \frac{x}{2} \oplus \frac{y}{2} = \circ\left(\frac{x+y}{2}\right)$.

If $|y| < 2^{p+E_i}$, then it is subnormal and the division may be inexact. But $x$ is big enough so that this error is too small to impact the result. More precisely, we prove that $r = (x \oslash 2) \oplus (y \oslash 2) = \frac{x}{2} \oplus (y \oslash 2) = \frac{x}{2} = \circ\left(\frac{x+y}{2}\right)$.

This is proved by using twice the following result: given a floating-point number $f$ and a real $h$ such that $2^{p+e_i} \leq |f|$ and $|h| \leq \frac{\mathrm{ulp}(f)}{4}$, then $\circ(f+h) = f$. □

As for the *average*1 function, the correct rounding implies all the previous requirements and gives a half ulp error bound. This hold provided either $x$ or $y$ is big enough.

### 4.4 Putting all parts together: the average functions

**Accurate Sterbenz algorithm** Following Sterbenz's ideas and the previous definitions, here is the definition of an accurate average function:

```
if x and y do not have the same sign
      return (x ⊕ y) ⊘ 2
else
      if |x| ≤ |y|
            return x ⊕ ((y ⊖ x) ⊘ 2)
      else
            return y ⊕ ((x ⊖ y) ⊘ 2)
```

Then the properties are easily derived from the properties of *average*1 and *average*3. They may be sometimes long as many subcases have to be studied, but the proofs are straightforward. The worst case is the proof that $average(-x, -y) = -average(x, y)$, as all sign possibilities (positive, negative and zero) have to be considered. The formal proof of the whole algorithm is a Coq file about 1,400 lines long.

What is left to prove is that no overflow occurs. Another difficulty is the specification of this program that will be described in Section 5.

**Correct algorithm** From the previous properties of *average*1 and *average*2, another algorithm can be defined, that will return the correctly-rounded average:
`let` $C := 2^{E_i+2p+1}$
`if` $C \leq |x|$
    `return` $(x \oslash 2) \oplus (y \oslash 2)$
`else`
    `return` $(x \oplus y) \oslash 2$

This program returns $\circ\left(\frac{x+y}{2}\right)$. This means that the specification reduces to this property, as it easily implies everything Sterbenz could wish for a correctly written average function. What is left to prove is that no overflow occurs. Another point is the value of $C$. The correct rounding will hold whatever $C$ greater or equal to $2^{E_i+2p+1}$ in our model without overflow. We may therefore increase this value as long as overflows are prevented. The advantage is efficiency: it would more often use 3 operations instead of 4.

## 5 Specifications and Formal Verification of the Programs

### 5.1 Absolute value

Both programs require an absolute value for tests. This may come from a standard library or playing with the first bit. As long as the specification is the same, any function will make the following programs work. We choose to define and prove it using a condition.

```
/*@ ensures \result == \abs(x); */
double abs(double x) {
  if (x >= 0) return x;
  else return (-x);
}
```

The corresponding proof is automatically done by Alt-Ergo.

### 5.2 Accurate average

The accurate program to be proved is the following one, written in C. It corresponds to Sterbenz's hints.

```
1  /*@ axiomatic Floor {
2    @  logic integer floor (real x);
3    @  axiom floor_prop: \forall real x; floor(x) <= x < floor(x)+1;
4    @ } */
5
6  /*@ logic real ulp_d (real x) =
7    @ \let e = 1+ floor (\log(\abs(x)) / \log(2));
8    @     \pow(2,\max(e-53,-1074)); */
9
10 /*@ logic real l_average (real x, real y) =
11   @    \let same_sign =
12   @      (x >= 0) ? ((y >=0) ? \true : \false) : ((y >=0) ? \false : \true);
13   @    (same_sign) ? ((\abs(x) <= \abs(y)) ?
14   @      \round_double(\NearestEven, x+\round_double(\NearestEven,
```

```
15    @            \round_double(\NearestEven, y-x)/2))  :
16    @      \round_double(\NearestEven, y+\round_double(\NearestEven,
17    @            \round_double(\NearestEven, x-y)/2))) :
18    @      \round_double(\NearestEven,\round_double(\NearestEven, x+y)/2);
19    @ */
20
21  /*@  lemma average_sym: \forall double x; \forall double y;
22    @         l_average(x,y) == l_average (y,x);
23    @  lemma average_sym_opp: \forall double x; \forall double y;
24    @          l_average(-x,-y) == - l_average (x,y);
25    @
26    @ lemma average_props: \forall double x; \forall double y;
27    @         \abs(l_average(x,y) - (x+y)/2) <= 3./2*ulp_d((x+y)/2)
28    @   &&  (\min(x,y) <= l_average(x,y) <= \max(x,y))
29    @   && (0 <= (x+y)/2 ==> 0 <= l_average(x,y))
30    @   && ((x+y)/2 <= 0 ==> l_average(x,y) <= 0)
31    @   && ((x+y)/2==0 ==> l_average(x,y)==0)
32    @   && (0x1p-1074 <= \abs((x+y)/2) ==> l_average(x,y) != 0);
33    @ */
34
35
36  /*@   ensures \result == l_average(x,y);
37    @   ensures \abs((\result - (x+y)/2)) <= 3./2*ulp_d((x+y)/2);
38    @   ensures \min(x,y) <= \result <= \max(x,y);
39    @   ensures 0 <= (x+y)/2 ==> 0 <= \result;
40    @   ensures (x+y)/2 <= 0 ==> \result <= 0;
41    @   ensures (x+y)/2 == 0 ==> \result == 0;
42    @   ensures 0x1p-1074 <= \abs((x+y)/2) ==> \result != 0;
43    @ */
44
45  double average(double x, double y) {
46    int same_sign;
47    double r;
48    if (x >= 0) {
49      if (y >=0) same_sign=1;
50      else same_sign=0; }
51    else {
52      if (y >=0) same_sign=0;
53      else same_sign=1; }
54    if (same_sign ==1) {
55      if (abs(x) <= abs(y)) r=x+(y-x)/2;
56      else r=y+(x-y)/2; }
57    else r=(x+y)/2;
58    //@ assert r==l_average(x,y);
59    return r;
60  }
```

The full annotated program is given above. Here are some details about the annotations. We only consider the **double** type meaning the binary64 type of the IEEE-754. First, the floor function, which rounds down a real number to an integer, is specified at lines (1–4). The ulp function, which gives the unit in the last place in double precision ulp_d, is then defined at lines (6–8). An interesting point is that it takes a real number as input, and not only a floating-point number. We want to compare the result to the ulp of the exact result.

The next big block at lines (10–19) defines a logic function that computes the average following the algorithm described in Section 4.4. In the ACSL syntax, it exactly describes what the program does. Why is it needed? The reason is that we want to prove that $average(x, y) = average(y, x)$ and this means two calls of the function. As a generic C function may have side effects, this cannot be stated as is. Therefore, we had to define a logic function, that has forcefully no side effects and prove properties on this logic function called l_average. We

will also of course prove it is equivalent to the real C program. Then comes the various properties of the l_average function: symmetry, sign, rounding error, and so on at lines (21–33).

Next comes the specification of the average C function: its equivalence with the logical function, the rounding error, the fact that the result is between the minimum and the maximum of $x$ and $y$, the fact that the sign is correct and that the result is non-zero when the exact average is big enough. Last is the C program with an assertion at line (58), that serves as logical cut to ensure the program is equivalent to its logical counterpart.

Now that the program is fully written, specified and annotated, we have to prove it. The toolchain generates a bunch of theorems, we have to prove all of them in order to verify that the program will not fail, and that it will respect its specification. The "not fail" point is crucial here as it will require to prove there is no overflow, without assuming range values for the inputs.

| Proof obligations | | **Alt-Ergo** | **Coq** | | **Gappa** |
|---|---|---|---|---|---|
| | | | | Nb lines | |
| Previous Coq proof (spec + proof) | | | 7.83 | 1,432 | |
| VC for model lemmas | Lemma average_sym | | 5.39 | 3 | |
| | Lemma average_sym_opp | | 5.45 | 6 | |
| | Lemma average_props | | 7.60 | 125 | |
| VC for average_ensures_default | 1. assertion | 0.17 | | | |
| | 2. postcondition | 0.61 | | | |
| | 3. assertion | 0.23 | | | |
| | 4. postcondition | 0.42 | | | |
| | 5. assertion | 0.06 | | | |
| | 6. postcondition | 0.04 | | | |
| | 7. assertion | 0.06 | | | |
| | 8. postcondition | 0.06 | | | |
| | 9. assertion | 0.06 | | | |
| | 10. postcondition | 0.06 | | | |
| | 11. assertion | 0.08 | | | |
| | 12. postcondition | 0.99 | | | |
| | 13. assertion | 0.05 | | | |
| | 14. postcondition | 0.06 | | | |
| | 15. assertion | 0.06 | | | |
| | 16. postcondition | 0.06 | | | |
| | 17. assertion | 0.09 | | | |
| | 18. postcondition | 1.21 | | | |
| | 19. assertion | 0.06 | | | |
| | 20. postcondition | 0.51 | | | |
| | 21. assertion | 0.07 | | | |
| | 22. postcondition | 0.54 | | | |
| | 23. assertion | 0.06 | | | |
| | 24. postcondition | 0.04 | | | |
| VC for average_safety | 1. floating-point overflow | | | | 0.00 |
| | 2. floating-point overflow | | | | 0.00 |
| | 3. floating-point overflow | | 10.43 | 8 | |
| | 4. floating-point overflow | | | | 0.00 |
| | 5. floating-point overflow | | | | 0.00 |
| | 6. floating-point overflow | | 9.41 | 8 | |
| | 7. floating-point overflow | | | | 0.00 |
| | 8. floating-point overflow | | | | 0.00 |
| | 9. floating-point overflow | | | | 0.00 |
| | 10. floating-point overflow | | | | 0.00 |
| | 11. floating-point overflow | | | | 0.00 |
| | 12. floating-point overflow | | | | 0.00 |
| | 13. floating-point overflow | | | | 0.00 |
| | 14. floating-point overflow | | | | 0.00 |
| | 15. floating-point overflow | | | | 0.00 |

| | | | | |
|---|---|---|---|---|
| 16. floating-point overflow | | | | 0.01 |
| 17. floating-point overflow | | | | 0.00 |
| 18. floating-point overflow | | | | 0.00 |
| 19. floating-point overflow | | | | 0.00 |
| 20. floating-point overflow | | | | 0.00 |
| 21. floating-point overflow | | | | 0.00 |
| 22. floating-point overflow | | | | 0.01 |
| 23. floating-point overflow | | | | 0.00 |
| 24. floating-point overflow | | | | 0.00 |
| 25. floating-point overflow | | | | 0.00 |
| 26. floating-point overflow | | | | 0.00 |
| 27. floating-point overflow | | 9.72 | 8 | |
| 28. floating-point overflow | | | | 0.00 |
| 29. floating-point overflow | | | | 0.00 |
| 30. floating-point overflow | | 9.75 | 8 | |
| 31. floating-point overflow | | | | 0.00 |
| 32. floating-point overflow | | | | 0.00 |

Let us now detail the VC (verification conditions) we have to prove. The list of theorems is given in the table above. Timings are in seconds, and the number of lines of Coq proofs is also given. The previous proofs described in Section 4 are given, just to give an order of magnitude of the respective proofs. Then comes the proofs of what is in the logic annotations: the lemmas. There are three of them and all are easily proved using the previous algorithm proofs. Two difficulties arose: the first one is to prove that the ulp defined in the C annotations is the same as in the Coq formalization. The second difficulty is to prove that the Coq definition is the same as the logical definition in the annotations. Then comes the postconditions of the average C function. Given the previous lemmas, they are straightforward and proved automatically.

Last but not least, are the proofs related to overflow, as this is the only possible way for this program to fail (for example, there is no pointer access or division by zero). Near all of them are proved automatically. Indeed, most operations do not overflow due to the case study of the signs of $x$ and $y$ and this is handled automatically using Gappa. For a few operations, it is not sufficient and we need to rely on the fact that $\min(x, y) \leq average(x, y) \leq \max(x, y)$, and a small Coq proof is then necessary.

### 5.3 Correct average

The correct program for computing the average is the following one, with hypotheses on the value of $C$.

```
1  /*@    requires 0x1p-967 <= C <= 0x1p970;
2    @    ensures \result == \round_double(\NearestEven, (x+y)/2) ;
3    @ */
4
5  double average(double C, double x, double y) {
6    if (C <= abs(x))
7      return x/2+y/2;
8    else
9      return (x+y)/2;
10 }
```

This specification is quite simpler. The result is the correct rounding of the average. Note that the value of $C$ must be between $2^{-967}$ and $2^{970}$. The $2^{-967}$ exactly corresponds to $2^{E_i+2p+1}$ in the binary64 format as $E_i = -1074$ and $p = 53$. As for $2^{970}$, the reason is overflow (see below).

| Proof obligations | | **Alt-Ergo** | **Coq** | | **Gappa** |
|---|---|---|---|---|---|
| | | | | Nb lines | |
| Previous Coq proof (spec + proof) | | | 3.82 | 536 | |
| VC for average_ensures_default | 1. postcondition | | 5.96 | 20 | |
| | 2. postcondition | | 2.71 | 9 | |
| VC for average_safety | 1. floating-point overflow | | | | 0.00 |
| | 2. floating-point overflow | | | | 0.00 |
| | 3. floating-point overflow | | | | 0.00 |
| | 4. floating-point overflow | | 14.32 | 64 | |
| | 5. floating-point overflow | | | | 0.00 |

Proofs of behavior are quite simple as they are calls to the previously studied *average*1 and *average*2 functions. The difficult part, as expected, is overflow. It is handled automatically by Gappa, except the proof that $x + y$ does not overflow, provided that $|x| < C \leq 2^{970}$. More precisely, even if $y$ is the biggest floating-point number, if $|x| < 2^{970}$, then $x \oplus y$ will not overflow as it will round to $y$.

## 6 Conclusion and Perspectives

The initial goal was to prove a program computing the average without overflow. This has first been achieved using Sterbenz's hints. This program has been successfully written, specified and proved. All the wanted properties have been proved, and a very good error bound on the rounding error is given. Even if the program is tricky, the proofs are not that long, even if some are tricky. Then another program is presented, which is both new, simpler to write and to prove. It is more efficient and more accurate. It handles all overflow and underflow cases, and gives the most accurate possible result: a correct rounding of the exact average.

The usual method to handle exceptional cases is scaling. This means computing the order of magnitude of the inputs (for example their exponent), and multiplying by a chosen power of the radix before and after the computation, in order to prevent any underflow or overflow during the computation. In particular, Sterbenz recommends scaling on this example to prevent underflow. We have proved this scaling to be useless, which causes a much more efficient program as scaling is costly.

An interesting point is that the overhead to prove the program, compared to the algorithm proofs, is rather low. When the program is specified (which was a difficult task), the proof is either automatic, or simple calls to the previous proofs. Surprisingly, the overflow proofs were not difficult: they were either automatic using Gappa or easily deduced from previous properties. The only difficult one was explained in the previous Section. We did not expect the other 36 theorems to be so easily handled. This case study shows that the difficult point about overflow is not proving it does not happen, but finding the algorithms such

that it does not happen. This example is among Sterbenz's "carefully written programs", and this is the reason why it behaves as expected. We did not expect this well-behavior to extend to the overflow proofs.

An unexpected difficulty was in the formalizations that describe the average computation of the accurate program. There are three of them:

- the Coq formalization, written directly in Coq and given in Section 4. It was written to be short and easily used in the formal proof assistant.
- the l_average logic function, written in the ACSL syntax in the annotations of the C program. It was written to ensure that this function is free from side effects, so that we can state that l_average(x,y) == l_average (y,x). Its translation in Coq is much longer and much more tedious to use.
- the average C function, written in the C program. Its translation in Coq depends upon the path taken in the program and its definition is based on floating-point operation postconditions. On a typical goal, the definition of the result of the C function average relies on about 20 hypotheses and 40 lines, which makes it difficult to read.

In the proofs, we handle these three different formalizations. They are very near, so the equivalence proofs are straightforward, but rather long and cumbersome.

As for the perspectives, the first one is to consider radix 10. Unfortunately, the same properties do not hold with the same algorithm. More precisely, the accurate program can produce a result smaller than the minimum of the values when using radix 10 [17] and the correct program is not correct anymore. Therefore, other algorithms have to be created, so that they could fulfill all the requirements, without overflowing. Correct rounding may probably be achieved using odd rounding [6], but it will probably be much more costly than in radix 2.

Another perspective is how to handle overflow in everyday programs. A method for the formal verification is to put preconditions that give ranges on the inputs and let Gappa prove the overflow requirements. This method is sometimes not optimal, but it works very well with satisfactory results. But on basic blocks from libraries, such as Two-Sum or Fast-Two-Sum, we want the best possible results. It means we want to have the tightest precondition, in order to cover all cases that do not fail. And this requires additional work.

Unfortunately, programs are often not carefully written with overflow in mind. There are overflowing examples in an overwhelming proportion of them. Our work is therefore either to give precise conditions for them to work correctly, or to rewrite them.

## Acknowledgments

The author is indebted to a referee of a previous version of this work, who rightfully pitied the fact that no program existed for a correctly-rounded average and pointed out the previously dismissed average2 function. The author is also thankful to P. Zimmermann and V.Lefèvre for constructive discussions that turned $E_i + 2p + 2$ into $E_i + 2p + 1$ in Theorem 2.

## References


1. Baudin, P., Cuoq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, version 1.9 (2013), `http://frama-c.cea.fr/acsl.html`
2. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages. Wrocław, Poland (August 2011)
3. Boldo, S.: Deductive Formal Verification: How To Make Your Floating-Point Programs Behave. Thèse d'habilitation, Université Paris-Sud (Oct 2014)
4. Boldo, S., Filliâtre, J.C.: Formal verification of floating-point programs. In: Kornerup, P., Muller, J.M. (eds.) Proceedings of the 18th IEEE Symposium on Computer Arithmetic. pp. 187–194. Montpellier, France (Jun 2007)
5. Boldo, S., Marché, C.: Formal Verification of Numerical Programs: from C Annotated Programs to Mechanical Proofs. Mathematics in Computer Science 5, 377–393 (2011)
6. Boldo, S., Melquiond, G.: Emulation of FMA and Correctly-Rounded Sums: Proved Algorithms Using Rounding to Odd. IEEE Transactions on Computers 57(4), 462–471 (2008)
7. Boldo, S., Melquiond, G.: Flocq: A unified library for proving floating-point algorithms in Coq. In: Antelo, E., Hough, D., Ienne, P. (eds.) 20th IEEE Symposium on Computer Arithmetic. pp. 243–252. Tübingen, Germany (2011)
8. Carreño, V.A., Miner, P.S.: 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)
9. Conchon, S., Contejean, E., Iguernelala, M.: Canonized rewriting and ground AC completion modulo Shostak theories : Design and implementation. Logical Methods in Computer Science 8(3), 1–29 (Sep 2012)
10. Daumas, M., Melquiond, G.: Certification of bounds on expressions involving rounded operators. Transactions on Mathematical Software 37(1), 1–20 (2010)
11. Goldberg, D.: What every computer scientist should know about floating-point arithmetic. ACM Comput. Surv. 23(1), 5–48 (Mar 1991)
12. Harrison, J.: Formal verification of floating point trigonometric functions. In: Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design. pp. 217–233. Austin, Texas (2000)
13. IEEE standard for binary floating-point arithmetic. ANSI/IEEE Std 754-1985 (1985)
14. IEEE standard for floating-point arithmetic. IEEE Std 754-2008 (Aug 2008)
15. Muller, J.M., Brisebarre, N., de Dinechin, F., Jeannerod, C.P., Lefèvre, V., Melquiond, G., Revol, N., Stehlé, D., Torres, S.: Handbook of Floating-Point Arithmetic. Birkhäuser (2010)
16. Russinoff, D.M.: 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 1, 148–200 (1998)
17. Sterbenz, P.H.: Floating point computation. Prentice Hall (1974)