Formal correctness of comparison algorithms between binary64 and decimal64 floating-point numbers

Arthur Blot, Jean-Michel Muller, Laurent Théry

Formal correctness of comparison algorithms between binary64 and decimal64 floating-point numbers

Arthur Blot[1], Jean-Michel Muller[1], and Laurent Théry[2]

[1] École Normale Supérieure de Lyon, Lyon, France,
arthur.blot@ens-lyon.org jean-michel.muller@ens-lyon.fr
[2] INRIA Sophia Antipolis - Méditerranée, France,
Laurent.Thery@inria.fr

Abstract. We present a full Coq formalisation of the correctness of some comparison algorithms between binary64 and decimal64 floating-point numbers.

1 Introduction

Both binary and decimal formats are defined in the IEEE 754-2008 standard. However, no operation is defined to compare them, and the "naïve" approach of converting a decimal floating point number to the closest binary one before comparing it to another binary FP number can lead to inconsistencies. Let us take for example these two FP numbers:

$$\begin{aligned} x &:= 7205759403792794/2^{56} && \text{(in binary64 format)} > 1/10 \\ y &:= 1/10 && \text{(in decimal64 format)} \end{aligned}$$

If we convert y to the binary64 format, we get x. This means the native approach would give that $x \leq y$, but actually $x > y$.

The paper [4] by Brisebarre et al. details and proves the correctness of several algorithms which can compare these numbers in a both efficient and entirely reliable way. The initial problem being simple, it would be nice if the algorithms and their correctness proofs were elementary too. This is unfortunately not the case. In particular, the correctness of some of the algorithms requires some non-trivial computation that cannot be performed by hand. The contribution of this paper is to propose a formalisation that carefully checks all the aspects of the correctness proofs.

We use the Coq proof assistant [9] for our formalisation. The paper is organised as follows. In a first section we detail the formal setting in which we have made our formalisation. Then, we present the algorithms and their proof of correctness.

2 Formal Setting

Coq is a generic system that lets us define new objects and build formal proofs of their properties. There are two main ways of adding new knowledge into the system. We can work conservatively defining a new notion in term of already defined ones or we can work axiomatically by providing the characteristics properties of the new notion. In our formalisation, the two approaches have been used. We now illustrate this with some elementary examples.

Positive numbers are represented in Coq in a conservative way by a recursive datastructure `positive` that mimics binary numbers. It is composed of 3 constructors: `xI` (a number that starts with a 1), `xO` (a number that starts with a 0), `xH` (the number 1). If we take the little endian convention, the binary number 1101 is represented as `(xI (xO (xI xH)))`. Thanks to this recursive datastructure, it is then possible to equip positive numbers with usual functions. For example, we give below the definition of the function that computes the successor function. It is defined recursively by a discussion on the structure of its argument `a`.

```
Function Psucc a :=
  match a with
    xI a1 => xO (Psucc a1) | xO a1 => xI a1 | xH => xO xH
  end.
```

Relative numbers are then defined on top of positive numbers using these constructor `Zero`, `Zpos` for positive numbers and `Zneg` for negative numbers. Usual notations can be associated with these numbers, so that `(Zneg (xI (xO (xI xH))))` is printed and parsed back as -13.

Similarly to the definition of the `Psucc` function, addition (`+`) is defined for relative numbers. Having this function as an algorithm inside Coq makes it possible to compute with relative numbers like in any programming language. Thus, `-13 + 4` evaluates to `9`. Furthermore, it is possible to reason about this function and prove that the distributive property holds.

```
Lemma add_assoc m n p, m + (n + p) = (m + n) + p.
```

Having a system where one can mix computation and proofs was mandatory for the formalisation of the properties we were aiming at. Coming back to this formalisation, we are using the Flocq library [2] to reason about floating point numbers. These numbers are parametrised by a radix β. They are defined as a record that contains two fields.

```
Record float (beta : radix) := Float { Fnum : Z; Fexp : Z }.
```

For example, the object `{|Fnum := 7205759403792794; Fexp := -56|}` is a floating-point number whose mantissa is 7205759403792794 and exponent is -56.

In our formalisation, we prove the correctness of the comparison algorithms with respect to the comparison over real numbers. The real numbers are axiomatised in Coq. An abstract type `R` is defined. A set of properties is assumed that makes it a complete, totally ordered, Archimedean field. On top of this

basic field, the standard Coq real library and the Coquelicot[1] library build the usual real analysis. This contains standard notions like series and limits from which usual functions such as exponential, sine and cosine are defined. Thanks to these libraries, it is possible to give an interpretation of a floating point number as a real number.

```
Definition F2R (f : float beta) := (Fnum f) * beta ^ (Fexp f).
```

The floating point number f represents the real numbers $m * \beta^e$ where m is the mantissa, e the exponent and β the base. Now the problem is simple, we want to build an algorithm `comp` that takes two floating point numbers `f1` and `f2` in base 2 and in base 10 respectively and returns their comparison. We use the standard Coq type `comparison` for the returned type of the comparison function `comp`. This type contains three elements `Lt`, `Eq`, `Gt` that represent the three possible values of a comparison. Our goal is then to prove that, under some conditions `Cond` that we will explain later, the following theorem holds

```
Lemma comp_correct (f1 : float 2) (f2 : float 10) :
  Cond →    comp f1 f2 = Lt ↔ F2R f1 < F2R f2
         ∧  comp f1 f2 = Eq ↔ F2R f1 = F2R f2
         ∧  comp f1 f2 = Gt ↔ F2R f1 > F2R f2.
```

This lemma relates the result of an evalutaion of the algorithm `comp` with a relation between two real numbers. When proving an algorithm `eq` that performs an equality test, the returned type is simply a boolean so the correctness theorem looks like

```
Lemma eq_correct (f1 : float 2) (f2 : float 10) :
  Cond → eq f1 f2 = true ↔ F2R f1 = F2R f2.
```

Note that as real numbers are axiomatised in Coq, we can not directly compute with them. Fortunately, there exist some tools that address this issue. One tool that we are using in this formalisation is the `interval` tactic [8]. It tries to automatically solve goals that contain real expressions by computing an approximation using standard techniques of interval arithmetic. For example, the following error bound of an approximation of the exponential function between 0 and 1 can be proved automatically.

```
Lemma bound_exp_taylor2 x :
  0 ≤ x ≤ 1 → Rabs ((1 + x + x ^ 2 / 2) - exp x) ≤ 22 / 100.
```

Another technique to approximate reals numbers are continued fractions. We are using them in our formalisation. Following closely the material that is presented by Khinchin [7], we have developed a library to compute and reason about continued fractions inside Coq. The main basic result that is needed here is that continued fractions represent the best rational approximations one can get. The formal statement is the following.

```
Lemma halton_min (n : nat) (p q : Z) (r : R) :
   0 < q < q[r]_n.+1 → t[r]_n ≤ Rabs (q * r - p).
```

Approximating a real number r by continued fractions can be seen as an iterative process. At rank n, we get an approximation, called convergent, $s_n = p_n/q_n$. One way of quantifying the quality of an approximation p/q is by the value, $|qr - p|$. Applied to s_n, this gives $\theta_n = |q_n r - p_n|$, written `t[r]_n` in our formalisation. The previous lemma `halton_min` simply states that the convergent at rank n is the best approximation for all the rational numbers p/q with q less than q_{n+1}.

3 Formalisation

We want to compare a binary floating-point number x_2 and a decimal floating-point number x_{10}. This is reflected in our formalisation by introducing two variables of type `float`.

```
Variable x2 : float 2.
Variable x10 : float 10.
```

Furthermore, x_2 belongs to the binary64 format and x_{10} to the decimal64 format. This means that we have:

$$x_2 = M_2 \cdot 2^{e_2 - 52}$$
$$x_{10} = M_{10} \cdot 10^{e_{10} - 15}$$

where:

$$-1022 \leq e_2 \leq +1023 \quad |M_2| \leq 2^{53} - 1$$
$$-383 \leq e_{10} \leq +384 \quad |M_{10}| \leq 10^{16} - 1$$

Without loss of generality, we can also suppose that $x_2 > 0$ and $x_{10} > 0$. If they are of opposite sign, the comparison is trivial and if they are both negative, taking the opposite of both numbers brings us back to the case we consider.

In our formalisation, as $x_2 > 0$, we shift the exponent is such a way that we have $2^{52} \leq M_2$. This corresponds to our `M2_bound` assumption.

```
Definition M2 := Fnum x2.
Hypothesis M2_bound : 2 ^ 52 ≤ M2 ≤ 2 ^ 53 - 1.
Definition e2 := Fexp x2 + 52.
Hypothesis e2_bound : -1074 ≤ e2 ≤ 1023.
```

Similarly for x_{10} but without any shifting, we get

```
Definition M10 := Fnum x10.
Hypothesis M10_bound : 1 ≤ M10 ≤ 10 ^ 16 - 1.
Definition e10 := Fexp x10 + 15.
Hypothesis e10_bound : -383 ≤ e10 ≤ 384.
```

The four hypothesis `M2_bound`, `e2_bound`,`M10_bound` and `e10_bound` are implicitely conditions of all the theorems we present in the following.

Let us start deriving new facts from these definitions. First, M_{10} can also be "normalised". If we consider $\nu = 53 - \lfloor \log_2(M_{10}) \rfloor$, we have that:

$$0 \leq \nu \leq 53$$

$$2^{53} \leq 2^{\nu} M_{10} \leq 2^{54} - 1$$

These correspond to the theorems `v_bound` and `norm2_bound` of our formalisation.

Now, we express x_2 and x_{10} with the largest common power of 2 possible. For this, we define $m = M_2$, $n = M_{10} 2^v$, $h = \nu + e_2 - e_{10} - 37$ and $g = e_{10} - 15$. This gives us the following equalities (theorems `x2_eq` and `x10_eq` respectively).

$$x_2 = m \cdot 2^h \cdot 2^{g-\nu} \qquad\qquad x_{10} = n \cdot 5^g \cdot 2^{g-\nu}$$

The initial problem of comparing x_2 and x_{10} reduces to comparing $m \cdot 2^h$ with $n \cdot 5^g$. We also get bounds over m, h, n and g.

$$\begin{aligned} 2^{52} &\leq m \leq 2^{53} - 1 \\ 2^{53} &\leq n \leq 2^{54} - 1 \\ -398 &\leq g \leq 369 \\ -787 &\leq h \leq 716 \end{aligned}$$

These are easily proven in Coq and correspond to the theorems `m_bound`, `n_bound`, `g_bound` and `h_bound`.

4 Handling "simple" cases

The first step of the algorithm consists in checking if the result can be determined by looking at the exponents g and h only. In order to compare 5^g and 2^h, a function φ is introduced. Its definition is the following.

$$\varphi(h) = \lfloor h \cdot \log_5 2 \rfloor$$

It is relatively easy to check that one can determine the result of the comparison between x_2 and x_{10} if g and $\varphi(h)$ differ. Formally, this means that the following two properties hold:

$$\begin{aligned} &\text{If} \quad g < \varphi(h) \quad \text{then} \quad x_2 > x_{10} \\ &\text{If} \quad g > \varphi(h) \quad \text{then} \quad x_2 < x_{10} \end{aligned}$$

They correspond to the theorems `easycomp_lt` and `easycomp_gt` in our formalisation.

In order to get an algorithm, one still needs to provide a way to compute $\varphi(h)$. One way to do this is to use the integer value $\lceil 2^k \cdot \log_5 2 \rfloor$ for a sufficient large value of k. In our case, $k = 19$ is enough, the following property holds.

$$\varphi(h) = \left\lfloor h \cdot \lceil 2^{19} \cdot \log_5 2 \rfloor \cdot 2^{-19} \right\rfloor \qquad \text{for } |h| \leq 1831$$

This corresponds to the theorem `prop2_1`. Unfortunately, such a theorem is outside the reach of the `interval` tactic which does not handle the floor and to the nearest functions automatically.

In order to overcome this problem, we use the characteristic properties of the rounding functions (x is a real number, y an integer)

$$\begin{array}{lll} \lfloor x \rfloor = y & \text{iff} & y \leq x < y + 1 \\ \lceil x \rfloor = y & \text{if} & |x - y| < 1/2 \end{array}$$

For example, in order to prove that $\lceil 2^{19} \cdot \log_5 2 \rfloor = 225799$, we reduce the problem to showing that $|2^{19} \cdot \log_5 2 - 225799| < 1/2$ which is solved automatically by the `interval` tactic. The proof of the `prop2_1` theorem is done in a similar way. We first generate the 3663 subgoals that represents all the possible values for h. For each of the subgoal, we guess what the value z for $\lfloor h \cdot \log_5 2 \rfloor$ is. We then need to prove that $\varphi(h) = z$ and $\left\lfloor h \cdot 225799 \cdot 2^{-19} \right\rfloor = z$. For both of these goals, we use the characteristic property of the floor function. Showing that $z \leq h \cdot \log_5 2 < z + 1$ and $z \leq h \cdot 225799 \cdot 2^{-19} < z + 1$ is done automatically by the `interval` tactic. This brute-force method is not the most elegant approach but it works. It takes Coq 40 minutes to check the theorem `prop2_1`.

Once all the properties of the function φ are proved, it is easy to write down an actual algorithm for the simple cases

```
Definition easycomp : comparison :=
  let h := v + e2 - e10 - 37 in
  let g := e10 - 15 in
  let phih := (225799 * h) / (2 ^ 19) in
  if g < phih then Gt
  else if g > phih then Lt
  else Eq.
```

and derive its associated theorem of correctness

```
Theorem easycomp_correct:
  (easycomp = Lt → F2R x2 < F2R x10) ∧
  (easycomp = Gt → F2R x2 > F2R x10) ∧
  (easycomp = Eq ↔ g = phi h).
```

Note that we are using the ideal integer arithmetic of Coq in a more realistic programming language we would have to deal with bounded arithmetic. Nevertheless, we can run our algorithm on the example given in the introduction and get the expected result.

```
Compute easycomp {| Fnum := 7205759403792794; Fexp := -56 |}
                 {| Fnum :=                1; Fexp :=  -1 |}.
  = Eq
  : comparison
```

Our correctness theorem tells us that all we can conclude is that $g = \varphi(h)$.

5 Exact testing

The algorithm in the previous section covers the cases where g differs from $\varphi(h)$. In this section, we assume that $g = \varphi(h)$.

5.1 Finding the needed precision

We define:

$$f(h) = 5^{\varphi(h)} \cdot 2^{-h} = 2^{\lfloor h \log_5 2 \rfloor \cdot \log_2 5 - h}$$

which verifies:

$$\begin{aligned} f(h) \cdot n > m &\Rightarrow x_{10} > x_2 \\ f(h) \cdot n < m &\Rightarrow x_{10} < x_2 \\ f(h) \cdot n = m &\Rightarrow x_{10} = x_2 \end{aligned}$$

Now, the idea is to get a precise enough approximation of $f(h) \cdot n$ in order to retrieve the exact comparison. Our goal is therefore to find a lower bound η over the smallest non-zero value of the error $|m/n - f(h)|$ in the range where

$$\begin{aligned} 2^{52} \leq m &\leq 2^{53} - 1 \\ 2^{53} \leq n &\leq 2^{54} - 1 \\ -787 \leq h &\leq 716 \end{aligned}$$

We can also add an extra constraint that directly comes from the definition of n.

$$\text{If} \quad 10^{16} \leq n \quad \text{then} \quad \text{even}(n)$$

A good candidate for this lower-bound, as proposed in [4], is $\eta = 2^{-113.7}$. In order to formally check that this lower-bound holds, we split in four the interval of h. For $0 \leq h \leq 53$, it is easily to prove that the lower bound is greater than 2^{-107}. This corresponds to the theorem `d_h_easy1`. Similarly, the lower bound is greater than 2^{-108} for $-53 \leq h \leq 0$ (`d_h_easy2`). For the two remaining intervals $54 \leq h \leq 716$ and $-787 \leq h \leq -54$, we make use of our library for continued fractions. The key idea is that in order to find the lower bound for a given h it is not necessary to enumerate all the rationals m/n but only the convergents $s_n = p_n/q_n$ that approximate $f(h)$ and whose denominator are bounded by $2^{53}-1$. The property that justifies this computation is the following:

$$\text{If} \quad q_n \leq n < q_{n+1} \quad \text{and} \quad |f(h) - m/n| < 1/2n^2 \quad \text{Then} \quad m/n = p_n/q_n$$

This is the `conv_2q2` theorem of our library that directly follows the proof given in [7]. In our case, as $\eta = 2^{-113.7}$ and $2^{53} \leq n \leq 2^{54} - 1$, we are sure that $\eta < 1/2n^2$, so our theorem applies.

For the first interval $54 \leq h \leq 716$, the lower bound ϵ_p/ϵ_q is reached for $h = 612$, $m = 3521275406171921$ and $n = 8870461176410409$ with

$\epsilon_p = 95837236471514977157488559853369253873503974784033193441759367158$
$879170025294997936467046601695051723045003185390596549988774013061$
$693908401178199189358693060060129 1$

and

$$\epsilon_q = 150766046227125863784130855355431583102341132913540640713808524072186206546962278594854444134050092053000527172816154049523115277896304282171615809979653733004672632613248399492700530151704033193099264$$

This corresponds to the theorem `pos_correct` of our formalisation. It is proved by a double enumeration. First, we enumerate all the h from 54 to 716, and for each of them, we enumerate all the convergents of $f(h)$ whose denominator is smaller than $2^{53} - 1$ and check that the error is bigger that ϵ_p/ϵ_q. The entire proof takes 80 seconds to be checked by Coq.

For the second interval $-787 \leq h \leq -54$, a similar technique is used. This time the lower bound ϵ'_p/ϵ'_q is reached for $h = 275$, $m = 4988915232824583$ and $n = 12364820988483254$ with

$$\epsilon'_p = 1125294231712324001348355690549630718090619037618685690393602513 97$$

and

$$\epsilon'_q = 1860451484274037501915433537364990769170345094056549724607571572043696050968719646334648132324218750$$

This corresponds to theorem `neg_correct` and takes 100 seconds to be validated by Coq.

The theorem `d_h_min` collects all the previous results and proves the smallest non-zero error is bigger than $\eta = 2^{-113.7}$. We thus know that if the error is less than η, the distance is 0 and we must be in the equality case.

5.2 Direct method

The first way to use the previous result is to precompute f in a table indexed by h. Then, assuming μ approximates $f(h) \cdot n$ with accuracy $\epsilon < \eta/4$, we have the following implications:

$$\begin{array}{llll} \text{If} & \mu > m + \epsilon \cdot 2^{54} & \text{Then} & x_{10} > x_2 \\ \text{If} & \mu < m - \epsilon \cdot 2^{54} & \text{Then} & x_{10} < x_2 \\ \text{If} & |m - \mu| \leq \epsilon \cdot 2^{54} & \text{Then} & x_{10} = x_2 \end{array}$$

This corresponds to the theorem `direct_method_correct` of our formalisation.

Since we have $\eta = 2^{-113.7}$, a 128-bit computed value of f is enough to make the exact comparison. In order to build the algorithm, we first build the table with 1504 entries that contains $\lceil f(h) \cdot 2^{127} \rceil$.

```
Definition f_tbl h : Z :=
  match h with
  | -787 => 155090275327330814032632713427604519407
  | -786 =>  77545137663665407016316356713802259704
```

```
| -785 =>  387725688318327035081581783569011298 52
...
...
...
|  714 =>  75715339914673581502256102241153698026
|  715 =>  37857669957336790751128051120576849013
|  716 =>  94644174893341976877820127801442122533
| _    =>  0
end.
```

Proving that this table is correct (for each entry i, the value represents $\lceil f(h) \cdot 2^{127} \rceil$) is done by brute and takes 40 minutes. The algorithm then just needs to look up in the table and compares with $m * 2^{217}$.

```
Definition direct_method_alg :=
  let v := n * (f_tbl h) - m * 2 ^ 127 in
  if v < - 2 ^ 57 then Gt
  else if v > 2 ^ 57 then Lt
  else Eq.
```

As we have $2^{124} \leq f(h) \cdot 2^{127} < 2^{128}$, our epsilon is equal to 2^{-124}. This explains the 57 in the algorithm, $57 = -124 + 127 + 54$. The proof of correctness is a direct consequence of these observations.

```
Theorem direct_method_alg_correct :
  (direct_method_alg = Lt ↔ F2R x2 < F2R x10) ∧
  (direct_method_alg = Eq ↔ F2R x2 = F2R x10) ∧
  (direct_method_alg = Gt ↔ F2R x2 > F2R x10).
```

We have seen in the previous section with the execution of the `easycomp` algorithm that we have $g = \varphi(h)$ for the example in the introduction. Now, running the `direct_method_alg` algorithm returns the correct value for the comparison.

```
Compute ineq_alg {| Fnum := 7205759403792794; Fexp := -56 |}
                 {| Fnum :=                1; Fexp :=  -1 |}.
  = Gt
  : comparison
```

The main drawbak of this algorithm is that the table may be too large for some realistic implementation. The next section explains how we can alleviate the problem by using a bipartite table.

5.3 Bipartite table method

In this method, we use:

$$q = \left\lfloor \frac{\varphi(h)}{16} + 1 \right\rfloor \quad r = 16q - \varphi(h)$$

such that $f(h) = 5^{16q} \cdot 5^{-r} \cdot 2^{-h}$. We can easily derive the bounds for q and r from the bound of $\varphi(h)$.

$$\begin{aligned} -21 \leq q \leq 20 \\ 1 \leq r \leq 16 \end{aligned}$$

In this method, we store 5^{16q} and 5^r after a phase of normalisation. For this, we define the function ψ as follows.

$$\psi(g) = \lfloor g \cdot \log_2 5 \rfloor$$

Similarly to what has been done for $\varphi(h)$, it is easy to compute $\psi(q)$ and $\psi(16q)$ using integer arithmetic:

$$\begin{aligned} \psi(g) &= \left\lfloor \lceil 2^{12} \cdot \log_2 5 \rfloor \cdot g \cdot 2^{-12} \right\rfloor && \text{for } |g| \leq 204 \\ \psi(16q) &= \left\lfloor \lceil 2^{12} \cdot \log_2 5 \rfloor \cdot q \cdot 2^{-8} \right\rfloor && \text{for } |q| \leq 32 \end{aligned}$$

These equalities correspond to the theorems `prop_2_2` and `prop_2_3` in our formalisation. They are proved by brute force in 5 minutes and 40 seconds respectively.

Now, we introduce the two functions θ_1 and θ_1 that are defined as follows.

$$\begin{aligned} \theta_1(q) &= 5^{16q} \cdot 2^{-\psi(16q)+127} \\ \theta_2(r) &= 5^r \cdot 2^{-\psi(r)+63} \end{aligned}$$

The following bounds are given by the theorems `theta1_bound` an `theta2_bound`.

$$\begin{aligned} 2^{127} \leq \theta_1(q) < 2^{128} - 1 \\ 2^{63} < \theta_2(r) < 2^{64} \end{aligned}$$

Furthermore, we have

$$f(h) = \frac{\theta_1(q)}{\theta_2(r)} 2^{-64-\sigma(h)} \qquad \text{with} \quad \sigma(h) = \psi(r) - \psi(16q) + h$$

If we define Δ as

$$\Delta = \theta_1(q) \cdot n \cdot 2^{-64+8} - \theta_2(r) \cdot m \cdot 2^{8+\sigma(h)}$$

comparing x_2 and x_{10} gives the same result as comparing 0 and Δ (theorem `delta_ineq`). We also easily get that if $x_2 \neq x_{10}$, then $|\Delta| \geq 2^{124}\eta$. This allows us to derive an approximated version of Δ:

$$\tilde{\Delta} = \left\lfloor \lceil \theta_1(q) \rceil \cdot n \cdot 2^{8-64} \right\rfloor - \theta_2(r) \cdot m \cdot 2^{8+\sigma(h)}$$

and proves that comparing 0 with $\tilde{\Delta}$ gives the same result as comparing x_2 with x_{10} (theorem `delta'_ineq`).

In order to build the algorithm, we first need to create the two tables for $\lceil \theta_1(q) \rceil$ and $\theta_2(r)$. Our tables are given in Figure 1. The first one contains integers that fit in 128 bits and the second one in 64 bits. The algorithm first computes q and r, then reads the values of $\lceil \theta_1(q) \rceil$ and $\theta_2(r)$, and then checks the sign of the computed $\tilde{\Delta}$.

```
Definition theta1_tbl q : Z :=
  match q with
  | -21 => 30291069399869299615748576841329007696 6
  | -20 => 33629842688253419175912847062602803678 9
  | -19 => 18668312833510458212900510778566200808 5
  | -18 => 20725990738668607319295523504017132241 9
  | -17 => 23010472126237643618935106442099516590 4
  | -16 => 25546755962044413589201570726871533645 7
  | -15 => 28362596673541699653588533662014114405
  | -14 => 31488807865122869393368946606905258090 5
  | -13 => 17479799754928565188243886678268334039 8
  | -12 => 19406476153758861689362243605781281940 8
  | -11 => 21545516652742137885659094560277007014 1
  | -10 => 23920328665319054867909425788093943381 5
  |  -9 => 26556899640838354934479410327623431366 5
  |  -8 => 29484081443918291814387145163970850710 3
  |  -7 => 32733906078961418700131896968275991522 2
  |  -6 => 18170968107390172263733095197200113358 9
  |  -5 => 20173827172553973356686868531273530268 3
  |  -4 => 22397447421778042105574422805684442781 3
  |  -3 => 24866161820489332107769112407341042005 1
  |  -2 => 27606985387162255149739023449108101809 9
  |  -1 => 30649910817317777167166940543006183672 4
  |   0 => 17014118346046923173168730371588410572 8
  |   1 => 18889465931478580854784000000000000000
  |   2 => 20971520000000000000000000000000000000
  |   3 => 23283064365386962890625000000000000000
  |   4 => 25849394142282114839731521627186339174 0
  |   5 => 28698592549372253612517981865774823687
  |   6 => 31861838222649045540577607955354236111 9
  |   7 => 17686873200833422592791248615015218321 7
  |   8 => 19636373861190906212383087819945102572 0
  |   9 => 21800754380841731685939475027186221303 1
  |  10 => 24203699467808239205112691458039699047 4
  |  11 => 26871504430268355007163862355800908518 3
  |  12 => 29833362924800826973163861261851735349 6
  |  13 => 33121686421112380675117871377923490061 1
  |  14 => 18386229439566681806493759420108863345 6
  |  15 => 20412815259847818312725919365334518557 8
  |  16 => 22662777498902795595110325882853306642 4
  |  17 => 25160737381238801985261861341284580098 5
  |  18 => 27934029957198183141977422640250350414 6
  |  19 => 31013003229050298983324099477976554711 4
  |  20 => 17215675123832984696095104991662469280 1
  |  _  => 0
  end.
```

```
Definition theta2_tbl r : Z :=
  match r with
  |  1 => 11529215046068469760
  |  2 => 14411518807585587200
  |  3 => 18014398509481984000
  |  4 => 11258999068426240000
  |  5 => 14073748835532800000
  |  6 => 17592186044416000000
  |  7 => 10995116277760000000
  |  8 => 13743895347200000000
  |  9 => 17179869184000000000
  | 10 => 10737418240000000000
  | 11 => 13421772800000000000
  | 12 => 16777216000000000000
  | 13 => 10485760000000000000
  | 14 => 13107200000000000000
  | 15 => 16384000000000000000
  | 16 => 10240000000000000000
  | _ => 0
  end.
```

Fig. 1. Tables for $\lceil \theta_1(q) \rceil$ and $\theta_2(r)$

```
Definition ineq_alg : comparison :=
  let q := g / 16 + 1 in
  let r := 16 * q - g in
  let psir := (9511 * r) / 2 ^ 12 in
  let psiq := (9511 * q) / 2 ^ 8 in
  let s := psir - psiq + h in
  let a := ((theta1_tbl q) * (n * 2 ^ 8)) / (2 ^ 64) in
  let b := (theta2_tbl r) * (m * 2 ^ (8 + s)) in
  let D := a - b in (0 =?= D)
```

The value 9511 corresponds to $\lceil 2^{12} \cdot \log_2 5 \rfloor$ and `=?=` to the comparison function for integer. Its associated correctness theorem is the following.

```
Theorem ineq_alg_correct :
  (ineq_alg = Lt ↔ F2R x2 < F2R x10) ∧
  (ineq_alg = Eq ↔ F2R x2 = F2R x10) ∧
  (ineq_alg = Gt ↔ F2R x2 > F2R x10).
```

Running the `ineq_alg` algorithm gives the same value as the one using the direct method.

```
Compute ineq_alg {| Fnum := 7205759403792794; Fexp := -56 |}
                 {| Fnum :=                1; Fexp :=  -1 |}.
  = Gt
  : comparison
```

6 Equality case

If we only want to test the equality between x_2 and x_{10}, an even simpler algorithm can be used. We already know that $x_2 = x_{10}$ is equivalent to $m \cdot 2^h = n \cdot 5^g$. As 2 and 5 are relatively prime, only two situations can occur :

- either $5^g \mid m$ and $2^h \mid n$ with $0 \leq g \leq 22$ and $0 \leq h \leq 53$;
- or $2^{-h} \mid m$ and $5^{-g} \mid n$ with $-22 \leq g \leq 0$ and $-51 \leq h \leq 0$.

The algorithm is a direct encoding of this property. It checks that $5^g \cdot (n2^{-h}) = m$ if we are in the first case or that $5^{-g} \cdot (m2^h) = n$ if we are in the second case.

```
Definition eq_alg : bool :=
  if (0 ≤ h) && (h ≤ 53) && (0 ≤ g) &&
     (g ≤ 22) && (n mod (2 ^ h) == 0) then
    let m' := 5 ^ g * (n / (2 ^ h)) in m' == m
  else if (h >= -51) && (-22 ≤ g) &&
          (g ≤ 0) && (m mod (2 ^ (-h)) = 0) then
    let n' := 5 ^ (- g) * (m / (2 ^ (- h))) in n' == n
  else
    false.
```

Its correctness theorem proves the equivalence between running the algorithm and testing the two values.

```
Theorem eq_alg_correct : eq_alg = true ↔ (F2R x2 = F2R x10).
```

On our favorite example, it returns the expected result.

```
Compute eq_alg {| Fnum := 7205759403792794; Fexp := -56 |}
               {| Fnum :=                1; Fexp :=  -1 |}.
  = false
  : comparison
```

7 Conclusion and Future Works

Our formalisation contains four algorithms for comparing a binary64 floating-point number and a decimal64 floating-point number and their proof of correctness. The code is available at:

`https://gitlab.com/artart78/compbindec`

It is composed of five files: `util.v` and `rfrac.v` for general results (most of which are about real numbers or continued fractions), `frac.v` for the result of the continued fraction problem solution, `compformula.v` for the computation of some formulas requiring tabulation of values, and `compbindec.v` for the main result. Altogether, this amounts to about 5000 lines of code that Coq checks in about 1 hour and 40 minutes. We have been using intensively SSReflect [6] for its set of tactics and syntax, the Flocq [2] library for manipulating floating-point numbers, and the Interval [8] tactic for the computation over the real numbers. We had to develop a dedicated library for continued fraction in order to tackle some aspects of the proof. This library is available in the `rfrac.v` file.

The main contribution of this work is a carefully check of the algorithms and the proofs presented in [4]. In particular, we have connected in a formal way the paper proofs with the computation that is required in order to get the accuracy at which $f(h) \cdot n$ needs to be computed. During the formalisation, we have found some minor mistakes in the original paper and we have also departed at some places from what was presented in the original paper. For example, the original paper states that $-20 \leq q \leq 21$ instead of $-21 \leq q \leq 20$. Fortunately, this mistake has no consequence. More values are tabulated in the actual implementation (the bounds are slightly relaxed). The original paper mentions the bound $2^{-113.67}$ while our formalisation only makes use of $2^{-113.7}$. The statement of the `direct_method_correct` omits the scaling factor of $f(h)$ in the assumption. It makes its proof easier and simplifies its application. Finally, the statements of the theorems `pos_correct` and `neg_correct` do not mention the extra condition ($\text{if } h \geq 680, \text{ then } \nu' = h + \varphi(h) - 971 > 0 \text{ and } 2^{\nu'} \mid n$) that is present in the original paper. It does not change the lower-bound and we believe that this omission makes the statements more readable at the cost of a negligible extra computing.

The capability of computing expressions over the real numbers thanks to the `interval` tactic has been a key ingredient on this work. These computations are often hidden in the original paper. It is the case for example for the bounds for θ_1 and θ_2. The bounds that are obtained without taking into account the bound on h are actually less strict than the ones specified in the paper. For example, only $\theta_1 < 2^{128}$ could be proved, but it is actually important to get the $2^{128} - 1$ bound in order to know we can embed the value into a 128-bit value. The 2^{128} bound was obtained by noticing that $16q \log_2 5 - \lfloor 16q \log_2 5 \rfloor < 1$, but the $2^{128} - 1$ bound requires that it is less than $1 + \log_2(1 - \frac{1}{2^{128}}) \simeq 1 - \frac{1}{2^{128}}$. We successfully manage to formalise all these proofs but clearly life would have been easier if the tactic `interval` would accommodate the floor and the ceiling functions. It could in particular greatly improve our checking time. 40 minutes

out of the hour and 40 minus of our checking time are spent by our brute-force method in trying to validate the integer equivalent for the function $\varphi(h)$.

There are several ways to extend this work. First, we could formalise the counting argument that is present in the original paper that quantifies the percentage of floating point numbers that are comparable using the first partiall comparison method (`easycomp`). It would require to develop the notion of cardinal of sets of floating point numbers in the Coq standard library. Second, what we have proven are only algorithms, it would be very interesting to try to prove a realistic software implementation using a tool like Why3 [5]. Finally a sequel of our reference [4] for this work has been written by the same authors in [3]. The general idea of the algorithm remains the same, but the generalisation of the result also requires a generalisation of all the intermediate results, and computation may become a lot harder since the numbers can then take up to 128 bits. Formalising it would be a real challenge.

References

1. S. Boldo, C. Lelay, and G. Melquiond. Coquelicot: A user-friendly library of real analysis for Coq. *Mathematics in Computer Science*, 9(1):41–62, 2015.
2. S. Boldo and G. Melquiond. Flocq: A unified library for proving floating-point algorithms in coq. In E. Antelo, D. Hough, and P. Ienne, editors, *20th IEEE Symposium on Computer Arithmetic, ARITH 2011, Tübingen, Germany, 25-27 July 2011*, pages 243–252. IEEE Computer Society, 2011.
3. N. Brisebarre, C. Q. Lauter, M. Mezzarobba, and J. Muller. Comparison between binary and decimal floating-point numbers. *IEEE Trans. Computers*, 65(7):2032–2044, 2016.
4. N. Brisebarre, M. Mezzarobba, J. Muller, and C. Q. Lauter. Comparison between binary64 and decimal64 floating-point numbers. In A. Nannarelli, P. Seidel, and P. T. P. Tang, editors, *21st IEEE Symposium on Computer Arithmetic, ARITH 2013, Austin, TX, USA, April 7-10, 2013*, pages 145–152. IEEE Computer Society, 2013.
5. J.-C. Filliâtre and A. Paskevich. Why3 - where programs meet provers. In M. Felleisen and P. Gardner, editors, *ESOP*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, 2013.
6. G. Gonthier and A. Mahboubi. A small scale reflection extension for the Coq system. Technical Report RR-6455, INRIA, 2008.
7. A. Khinchin and H. Eagle. *Continued Fractions*. Dover books on mathematics. Dover Publications, 1964.
8. E. Martin-Dorel and G. Melquiond. Proving Tight Bounds on Univariate Expressions with Elementary Functions in Coq. *Journal of Automated Reasoning*, 57(3):187–217, octobre 2016.
9. The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.6*, Dec. 2016. http://coq.inria.fr.