Floating-point arithmetic in the Coq system

Guillaume Melquiond[a,1]

[a] *INRIA Saclay – Île-de-France, PCRI, bât. 650, Université Paris 11, F-91405 Orsay Cedex, France*

Abstract

The process of proving some mathematical theorems can be greatly reduced by relying on numerically-intensive computations with a certified arithmetic. This article presents a formalization of floating-point arithmetic that makes it possible to efficiently compute inside the proofs of the Coq system. This certified library is a multi-radix and multi-precision implementation free from underflow and overflow. It provides the basic arithmetic operators and a few elementary functions.

Keywords: floating-point arithmetic, formal proofs, Coq system

1. Introduction

Some mathematical theorems have recently been proved by performing huge amounts of numerical computations (e.g. Hales' proof of Kepler's conjecture), hence casting some doubts on the validity of their proofs. By performing these numerical computations inside a formal system, a much higher confidence in the theorems would have been achieved. Indeed, these systems ruthlessly track unproven assumptions and incorrectly applied theorems, hence helping the user to perform foolproof reasoning. Fortunately, in some of these systems, especially in those based on type-theoretical formalisms, the ability to perform calculations is steadily increasing. It then becomes possible to use these calculations in place of some traditional deductive proofs, hence freeing the user from this burden [1].

One of these formal systems is the Coq proof assistant [2], which is based on the calculus of inductive constructions. Its formalism makes it possible to evaluate functions and to use their results inside proofs. This system is therefore a good candidate for implementing certified yet efficient arithmetics, and hence for using numerical computations to formally prove mathematical results [3].

Irrespective of these considerations, floating-point arithmetic is widely used in computer programs as a fast approximation of the traditional arithmetic on

Email address: `guillaume.melquiond@inria.fr` (Guillaume Melquiond)

[1] This work was partly done while at the Microsoft Research – INRIA Joint Laboratory.

real numbers. By design (limited range, limited precision), it is efficient for performing numerically-intensive calculations. While these calculations are most often encountered in numerical simulations of physical phenomenons, Thomas Hales has shown that they could also be a great help for proving some mathematical theorems [4].

This paper presents a formalization of an effective floating-point arithmetic inside the Coq system. The incentive is to provide a library that can be used to prove theorems by performing numerical computations. For instance, interval arithmetic is a simple method for computing enclosures of real-valued expressions and therefore to prove bounds on such expressions. Moreover, verifying such bounds is usually no harder than computing them directly, so a formal system would end up having to do these computations, even when helped by an external oracle. The floating-point library presented in this paper provides the definitions and the corresponding correctness lemmas that make it easy to write tools that performs proofs by numerical computations [5].

A formal system is an unusual environment for developing a floating-point arithmetic, so Section 2 presents some design decisions and how this library compares to other floating-point formalizations. Section 3 then details the implementation of the basic operators: addition, multiplication, division, and square root. The library also encompasses some elementary functions which are described in Section 4. Finally, some recent realizations and future works are presented in Section 5.

2. Floating-point arithmetic

Floating-point numbers are usually a subset of rational numbers, with some additional values for handling exceptions (e.g. infinities). A radix β is associated to a floating-point arithmetic, and its finite numbers can be represented as $m \cdot \beta^e$, with m and e two integers. Most common radixes are $\beta = 2$, widely available in general-purpose processors, and $\beta = 10$, often found in financial applications. One can also find $\beta = 16$ in some older hardware, and $\beta = 3$ in exotic computers.

For the sake of realizability and/or space efficiency, a precision is usually set, that is an integer p such that the floating-point numbers are restricted to mantissas m that are bounded: $|m| < \beta^p$. For the same reasons, exponents e are also constrained to a range $[e_{\min}, e_{\max}]$. For instance, the double precision arithmetic described in the IEEE-754 standard [6] is a radix-2 arithmetic with $p = 53$, $e_{\min} = -1074$, and $e_{\max} = 971$. A multi-precision library like MPFR [7] works with any precision but still has bounded exponents, though the bounds are sufficiently big so that they do not matter usually.

2.1. Number format

The floating-point formalization presented in this article supports any radix $\beta \geq 2$. Indeed, because of their simplicity, the operations described in Section 3 do not rely on the properties of a specific radix. While any radix can be used, the library does not handle mixed-radix operations. One cannot add a radix-2 number with a radix-3 number and obtain a radix-10 number. So the radix can be seen as a global setting for the floating-point arithmetic. This is not the case for the precision of the numbers. It can be set independently for each operation, and an operation will return a number with the given precision.

Be it in Coq or in other systems, integers are mathematical objects with no immediate notion of limited range. In particular, the integers m and e representing a number $m \cdot \beta^e$ can be arbitrarily big. So there is no need for an upper bound on the precision available to computations. Similarly, constraining the exponents to a bounded range would be artificial. So the formalization has neither $e_{\min}$ nor $e_{\max}$. As a consequence, underflow and overflow no longer occur during computations. It means that exceptional values like signed zeros, subnormal numbers, and infinities, are a lot less useful, and have hence been discarded from the formalization.

The unbounded range of exponents has some immediate properties. First, a floating-point operation will only return zero when the ideal mathematical result is zero too. Second, all the results are normal numbers, so bounds on the relative errors are always verified, which makes it easier to write floating-point algorithms and to perform their error analysis. One no longer has to deal with the traditional sentence: "The result is correct assuming that no underflow nor overflow occurred in the course of the computations."

The formalization nonetheless supports an exceptional value: *Not-a-Number*. It is returned by floating-point operators, when the corresponding mathematical function on real numbers is not defined on the inputs. For instance, a NaN will be returned when computing $\frac{1}{0}$ and $\sqrt{-1}$. As usual, this exceptional value is an absorbing element for all the floating-point operations. So the final result of a sequence of computations will be NaN, if any of the intermediate computations had "invalid" inputs. This is especially useful, since the pure functional programing language of Coq does not offer native exceptions and traps, so delaying the handling of the invalid cases simplifies the use of floating-point operators inside proofs.

2.2. Data sets and functions

Since neither the precision nor the exponent are bounded, any number $m \cdot \beta^e$ can be represented and handled within this formalization. Let us note $\mathbb{F}_\beta$ this subset $\{ \ m \cdot \beta^e \mid (m, e) \in \mathbb{Z}^2 \ \}$ of the real numbers. These numbers will be represented as pairs of integers (m, e). Notice that these pairs are not normalized *a priori*, so (m, e) and $(m \cdot \beta, e - 1)$ are two different representations of the same real number. The set $\overline{\mathbb{F}_\beta}$ will denote the whole set of floating-point numbers, that is $\mathbb{F}_\beta$ extended with a NaN value.

To summarize, a floating-point operation like the division will be a function with a signature depending on its (implicit) first parameter β:

$$\texttt{Fdiv} : \forall \beta : \text{radix},\ \text{rounding} \to \text{precision} \to \overline{\mathbb{F}_\beta} \to \overline{\mathbb{F}_\beta} \to \overline{\mathbb{F}_\beta}$$

The "rounding" type contains modes for selecting a floating-point result when the ideal mathematical value is either outside of $\overline{\mathbb{F}_\beta}$ or not representable with the required precision. The supported modes are detailed in Section 3.1. The "precision" type denotes all the positive integers, though the results will not be specified when precision is 1.

So that the floating-point division can be used in a meaningful way, the library contains the following Coq theorem: For any two floating-point numbers x and y in radix β (bigger than 1) and for any rounding *mode* and precision *prec*, the result of the algorithm represents the same real number as the real quotient $\frac{x}{y}$ once rounded to a floating-point number.

```
Theorem Fdiv_correct :
  forall radix mode prec (x y : float radix), 1 < radix ->
  Fdiv mode prec x y = round radix mode prec (x / y).
```

This theorem matches the IEEE-754 requirement: "Every operation shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then rounded that result according to..." Note that the real quotient $\frac{x}{y}$ and its rounded value are defined in a purely axiomatic way in Coq: Standard real numbers are an abstract type, so they are unusable for computing. All the computations are performed on integers and they are proved afterwards to match the abstract result.

2.3. Computability

Several formalizations of floating-point arithmetic exist, for various proof assistants, and they have been successful in proving numerous facts on floating-point algorithms. For the sake of conciseness, only three of them will be cited here [8, 9, 10]. The main difference between these libraries and the one described in this paper is in the rationale. They were designed for proving floating-point code, but not for performing actual computations. For instance, the following predicate is the usual characterization of rounding to $-\infty$ in their formalization: A real number x is rounded to f in a floating-point *format* (radix, precision, and exponent range) if

$$f \in \mathit{format} \ \wedge \ f \leq x \ \wedge \ \forall g \in \mathit{format}, \ g \leq x \Rightarrow g \leq f.$$

While this is the best definition for describing the behavior of a floating-point rounding, this predicate does not provide any computational content, since there is no direct way for computing f given this definition and x. For their verification purpose, these libraries do not need to provide explicit algorithms for rounding values. If they do (either directly or through an oracle), the motivation is to evaluate floating-point operations only "for particular explicit values" [9]. They were not designed to perform numerically-intensive computations involving millions of floating-point operations, as is done in the example of Section 5.1 [5].

Note that the theorems provided by the existing Coq libraries [10] cannot be easily reused, unfortunately. Indeed, since their goal is the verification of code relying on actual floating-point arithmetic, e.g. IEEE-754, the bounded exponent range (especially the minimal exponent) is intimately embedded in their formalization.

2.4. Modules

The first part of the formalization defines and proves algorithms that work for any radix. While usable to perform floating-point computations, the purpose of this generic implementation is more of a reference implementation. For instance, counting the number of digits of a mantissa m is performed by increasing n until β^n is bigger than $|m|$. This is correct yet inefficient. The issue is similar when shifting mantissas – left $m \cdot \beta^n$ and right $\lfloor m/\beta^n \rfloor$ – since the shifts are performed by powering followed by a multiplication or a division.

The `Z` integers provided by Coq's standard library are represented as list of bits. As a consequence, when the radix is $\beta = 2$, mantissa operations can be performed much more efficiently. For instance, shifts can be computed by adding zeros at the head of the list or removing the first elements, since the least-significant bits are stored first. So the library has been specialized for specific radixes in order to improve performances.

This specialization is achieved thanks to the module system of Coq. First of all, an interface (*module type*) `FloatOps` contains the signature and the specification of all the floating-point operations. A module that implements this interface must therefore provide a set of floating-point operators and a proof that these functions return the exact same results as the functions of the reference implementation. For instance, if the module `M` has type `FloatOps`, the `M.add` function is a floating-point adder and the `M.add_correct` theorem proves its correctness. An important point is that the radix is no longer a parameter of the floating-point operations.

The first implementation of this interface is the module functor `GenericOps`. Given a radix, it generates a module of type `FloatOps` which is a simple wrapper around the reference implementation. For instance, `Module M := GenericOps Radix10` generates a module `M` which provides decimal floating-point arithmetic. The second implementation is provided by the functor `SpecificOps`. Given a set of functions for performing special operations on mantissas (e.g. shifts), this functor generates an improved module that directly manipulates integers at the digit level. For instance, `SpecificOps StdZRadix2` takes advantage of the representation of Coq's standard integers as bit lists in order to speed up radix-2 floating-point computations.

The implementation can be sped up even further by replacing the bit lists with binary trees of fixed-size integers [3]. The binary-tree structure allows for divide-and-conquer algorithms, e.g. Karatsuba's multiplication. Moreover, Coq can use 31-bit machine integers for representing the leaves of the trees. This considerably reduces the memory footprint of the integers, and makes it possible to delegate arithmetic computations on the leaves to the processor. Arithmetic operations are no longer performed one bit after the other, but by blocks of 31 bits at once. As a consequence, the module `SpecificOps BigIntRadix2` is currently the fastest radix-2 implementation, when evaluating expressions with Coq's virtual machine.

3. Basic operators

This section presents the implementation of the basic arithmetic operators. These operators have been developed with a simple design on purpose, so that they can be easily proved correct. As a consequence, this Coq library does not follow the laudable philosophy of the multi-precision floating-point library MPFR: "The complexity should, while preserving correct rounding, depend on the precision required on the result rather than on the precision given on the operands" [7]. When working at constant precision, this issue fortunately does not matter.

3.1. Rounding

If the result x of an exact operation can be represented as a pair (m, e) with $|m| < \beta^p$, then this floating-point value should be returned by the associated operator running at precision p. Otherwise, a floating-point number near x is chosen according to a rounding direction and is returned by the operator.

Let us assume that x is a positive real number. Let $e = \lfloor \log_\beta x \rfloor - p + 1$ and $m = \lfloor x \cdot \beta^{-e} \rfloor$. Both $m \cdot \beta^e$ and $(m+1) \cdot \beta^e$ can be represented by floating-point numbers with mantissas of p digits. Moreover, they are respectively the biggest number smaller than x and the smallest number bigger than x. These are the two candidates toward which x can be rounded.

We first need to have the position of x relatively to these two numbers. Let us pose $d = x \cdot \beta^{-e} - m$. By construction, d is in the range $[0, 1)$. If d is 0, then x is equal to $m \cdot \beta^e$. This position is called `pos_Eq` in the library. Otherwise, d is either smaller than $\frac{1}{2}$ (`pos_Lo`), equal to $\frac{1}{2}$ (`pos_Mi`, x is half-way between $m \cdot \beta^e$ and $(m+1) \cdot \beta^e$), or bigger (`pos_Hi`).

Choosing the correctly-rounded value is therefore a simple case study. For instance, when rounding to nearest the number x at position `pos_Hi`, the floating-point value $(m+1, e)$ is returned. This four-position system is strictly equivalent to the usual hardware approach of using two bits: the rounding bit and the sticky bit [11]. The four rounding directions of the IEEE-754 standard are supported: toward zero, toward $-\infty$, toward $+\infty$, and to nearest (with tie breaking to numbers with even mantissa). New rounding modes can easily be supported, as long the discontinuity points are either floating-point numbers or half-way between consecutive ones.

The position system does not have to be restricted to powers of the radix though. Its generalization will give the correctness proof of the division operator for free (Section 3.3). The library defines a `correctly_located` predicate to express that the mantissa m and the position *pos* are correctly chosen for a given *scale*. This predicate holds when:

$$\begin{cases} x = m \times \mathit{scale} & \text{if } \mathit{pos} \text{ is } \texttt{pos\_Eq} \\ m \times \mathit{scale} < x < (m + \frac{1}{2}) \times \mathit{scale} & \text{if } \mathit{pos} \text{ is } \texttt{pos\_Lo} \\ \ldots \end{cases}$$

For arbitrary numbers x and *scale*, the position and m are not computable.[2] But they are when we already know an integer m' and a position pos' for a scale $scale/n$, with n a positive integer. Indeed, m is then $\lfloor m'/n \rfloor$, and the position is given by the `adjust_pos` function of the library, which compares the remainder $m' - n \times \lfloor m'/n \rfloor$ with $\lfloor \frac{n}{2} \rfloor$. For instance, the new position is `pos_Mi` if the remainder is $\frac{n-1}{2}$ and the old position is `pos_Mi` too.

3.2. Addition and multiplication

The set $\mathbb{F}_\beta$ is actually a ring for the addition and the multiplication inherited from the real numbers. For instance, given two numbers (m_1, e_1) and (m_2, e_2), the pair $(m_1 \cdot m_2, e_1 + e_2)$ represents their product. For the addition, the mantissas have first to be aligned so that the two numbers are represented with the same exponent. The mantissas can then be added. Functions for exact addition and exact multiplication are therefore provided. This ring structure also helps in defining and proving the rounded operators.

Indeed, if the mantissa of the exact result (m, e) has less than p digits, this is also the correctly-rounded result. Otherwise, $(m, \texttt{pos\_Eq})$ correctly locates the number $m \cdot \beta^e$ with the scale β^e. So it can be scaled down to the lower precision p by using the algorithm of Section 3.1. Once done, the new position is used to decide which number to return.

Notice that, for the floating-point rounded addition, this approach is especially inefficient when the exponents e_1 and e_2 are far from each other, as it requires a huge shift followed by a long addition. For instance, the sum of $1 \cdot \beta^{10000}$ and $1 \cdot \beta^0$ involves a 10001-digit addition, while the result is more or less trivial for small precisions, depending on the rounding direction. A better approach would be to extract from the inputs the mantissa parts that actually matter in the rounded result.

3.3. Division and square root

For division and square root, one cannot expect to compute the exact values first. It is nonetheless possible to perform divisions and square roots on integer mantissas and to obtain an exact integer remainder.

Given two positive numbers (m_1, e_1) and (m_2, e_2), the division operator first computes the lengths of the mantissas $l_1 = 1 + \lfloor \log_\beta m_1 \rfloor$ and $l_2 = 1 + \lfloor \log_\beta m_2 \rfloor$. The integer $n = \max(l_2 + p - l_1, 0)$ is such that the integer quotient $q = \lfloor m_1 \cdot \beta^n / m_2 \rfloor$ has at least p digits. The exact result $m_1/m_2 \cdot \beta^{e_1 - e_2}$ of the division is correctly located by $(m_1 \cdot \beta^n, \texttt{pos\_Eq})$ with the scale $\beta^{e_1 - e_2 - n}/m_2$. So it can first be scaled down by a factor m_2, which gives a new location with q as a mantissa. It can then be scaled again by a factor β^k if q has $p + k$ digits. In the end, we get a location of x whose scale is a power of β and whose mantissa has p digits exactly. This is sufficient to get the correctly-rounded value of x.

[2] More generally, rounding operators are functions that are not continuous, so they are not computable on the set of real numbers.

The square root algorithm cannot rely on these changes of scales. It is nonetheless similar. Indeed, in Section 3.1, the rounding algorithm relies on the remainder of the euclidean division. So it can be adapted so that it uses the remainder of the integer square root. In order to compute the rounded result of the square root of (m, e), the operator first computes the length l of m. It then chooses the first integer n bigger than $\max(2p - l - 1, 0)$ and such that $e - n$ is an even integer. The integer square root $s = \lfloor \sqrt{m \cdot \beta^n} \rfloor$ has at least p digits. The remainder is $r = m \cdot \beta^n - s^2$. If r is zero, the exact result is at the $(s, \texttt{pos\_Eq})$ location with a scale $\beta^{(e-n)/2}$. Otherwise, the position is obtained by comparing r to $s + \frac{1}{2}$, which is the half-distance between s^2 and $(s+1)^2$. Since r and s are both integer, they can actually be compared directly. Finally, the location is scaled down again so that the mantissa has exactly p digits.

4. Elementary functions

For the basic operations, the exact real result can either be represented directly as a floating-point number, or with the help of a representable remainder. This is no longer the case for elementary functions. Except for a few trivial inputs, e.g. 0, one can only compute non-singleton ranges enclosing the exact result. This is nonetheless sufficient in order to get correct rounding, as shown by Ziv's iterative process [12].

Formalizing this process in Coq, however, depends on theorems that are currently out of scope. So the elementary functions do not return the correctly-rounded result. Instead, they return an interval enclosing the exact mathematical result. Fortunately, this interval is sufficient when proving inequalities on real-valued expressions. So the Coq tactics [5] that depend on this floating-point library do not need the correct rounding.

The library currently supports the functions cos, sin, tan, arctan, and exp. The implementation of other elementary functions like log is a work in progress. The library relies on an exact division by 2 for floating-point numbers, so the elementary functions cannot be used when the radix is odd.

4.1. Series evaluation and interval arithmetic

Elementary functions are evaluated thanks to simple power series whose terms happen to be alternating and decreasing for small inputs (say $0 \le x^2 \le \frac{1}{4}$). For instance,

$$\arctan x = x \cdot \sum_{i=0}^{\infty} (-1)^i \cdot \frac{(x^2)^i}{2i+1}$$

From $|x| \le \frac{1}{2}$, we can decide how many terms are needed so that the remainder of the series is guaranteed to be smaller than a threshold β^{-k}. For instance, when $\beta \le 4$, k terms are sufficient. This does not mean that the function will compute that many terms, it just means that k is the explicitly-constructed decreasing argument needed to define the recursive summation. Actually, the function tests the current term against the threshold and stops as soon as it is smaller.

A careful error analysis would then permit us to define and to prove an algorithm for evaluating truncated summation with an absolute error less than β^{-k} too. The relative error for arctan would be similar when computing it this way. Unfortunately, this error analysis is currently out of scope. As a replacement, a small kernel of floating-point interval arithmetic was implemented and proved in Coq. So the summation of the series is instead performed with intervals at precision k. This takes into account both the rounding errors and the series remainder, and it trivially ensures that $\arctan x$ is contained in the computed range. But the relative width of the range is no longer guaranteed to be smaller than β^{-k} and hence to converge toward zero when k increases. This prevents the completion of a proof that Ziv's process actually terminates for this implementation.

4.2. Argument reduction

4.2.1. Forward trigonometric functions and exponential

In order to get an input x with an absolute value smaller than $\frac{1}{2}$, an argument reduction is performed. For the three direct trigonometric functions, angle-halving formulas are used:

$$\begin{aligned} \cos(2 \cdot x) &= 2 \cdot (\cos x)^2 - 1 \\ \operatorname{sign}(\sin(2 \cdot x)) &= \operatorname{sign}(\sin x) \cdot \operatorname{sign}(\cos x) \end{aligned}$$

These formulas give $\cos x$ and the sign of $\sin x$ for any x. The values $\sin x$ and $\tan x$ are then reconstructed thanks to the following formulas:

$$\begin{aligned} \sin x &= \operatorname{sign}(\sin x) \cdot \sqrt{1 - (\cos x)^2} \\ \tan x &= \operatorname{sign}(\sin x) \cdot \operatorname{sign}(\cos x) \cdot \sqrt{(\cos x)^{-2} - 1} \end{aligned}$$

When no argument reduction is needed for sin and tan, the library does not rely on the power series of cos. Instead, it relies directly on the series of sin.

For the exp function, the argument is first reduced to a negative number by using $\exp x = (\exp(-x))^{-1}$ if needed. The argument is then brought in the domain $[-\frac{1}{2}, 0]$ by squaring the result: $\exp x = (\exp \frac{x}{2})^2$. On this reduced domain, the power series of exp is alternated, hence computable the same way than the other functions.

These argument reductions were chosen because they provide exact reduced arguments and are simple to prove. It would be interesting to compare their performances against the more traditional approach of using a trivial reconstruction and inexact reduced arguments:

$$\begin{aligned} \cos x &= \cos(x - k \cdot 2\pi) \quad \text{with } k = \left\lfloor \frac{x}{2\pi} \right\rceil \\ \exp x &= \exp(x - k \cdot \log \beta) \cdot \beta^k \quad \text{with } k = \left\lfloor \frac{x}{\log \beta} \right\rceil \end{aligned}$$

4.2.2. arctan *and* π

The argument of the arctan function is first reduced to $x > 0$ by using the parity of the function. The result is then computed thanks to the power series on the domain $[-\frac{1}{3}, \frac{1}{2}]$, after a potential reduction with the following formulas:

$$\arctan x = \begin{cases} \frac{\pi}{4} + \arctan \frac{x-1}{x+1} & \text{for } x \in [\frac{1}{2}, 2] \\ \frac{\pi}{2} - \arctan \frac{1}{x} & \text{for } x \geq 2 \end{cases}$$

Notice that the result reconstruction of arctan involves the constant $\frac{\pi}{4}$. It is computed thanks to Machin's formula $4 \cdot \arctan \frac{1}{5} - \arctan \frac{1}{239}$, which needs the computation of arctan for small arguments only. In order not to recompute this constant for every arctan evaluation, it is stored inside a co-inductive object. The i-th element of the co-inductive object is defined as an interval computed at precision $31 \cdot i$ with Machin's formula and hence containing $\frac{\pi}{4}$. A co-inductive object can be seen as an infinite list. One can destroy it into two elements: its head and its tail which is again an infinite list. Whenever a co-inductive object is destructed into these two elements, Coq's virtual machine has to compute the value of its head.[3] Moreover, Coq's virtual machine remembers this head value, so that it can be instantly reused the next time a function destructs the co-inductive object. Therefore, co-inductive objects can be used as a cache (that would never be flushed though). In particular, the interval constant for π at a given precision (with a granularity of 31 digits) is computed only once per Coq session.

4.2.3. Reconstruction and accuracy

The library performs all the intermediate computations at the same precision. The reconstruction process may, however, incur a loss of accuracy. For instance, in the case of exp, one bit (or $\log_\beta 2$ digits) of accuracy is lost for each squaring, as long as second-order rounding errors are negligible. Figure 1 shows the number of bits lost (relative width of the output interval on a $\log_2$ scale) depending on the magnitude of the input (again on a $\log_2$ scale).

For each input exponent, the relative width was averaged on 128 values. The precision used for the computations was 30 bits, but the overall figure does not change with other precisions: Be it 50 or 100 bits, the plot looks similar. Figure 1 shows that the loss is constant as long as the input is less than 2^{-1}. Then one additional bit is lost each time the input gets twice as big, except for the arctan function, whose reconstruction process has a fixed number of operations.

As a consequence, the functions have been modified to take into account the loss of accuracy caused by the argument reduction: The internal precision is increased depending on the magnitude of the input, so that the resulting interval has an average width of one unit in the last place. Therefore, even if

[3] Since the co-inductive object is an infinite list, its elements cannot be computed at the time the object is created; they are lazily evaluated when the object is destructed.

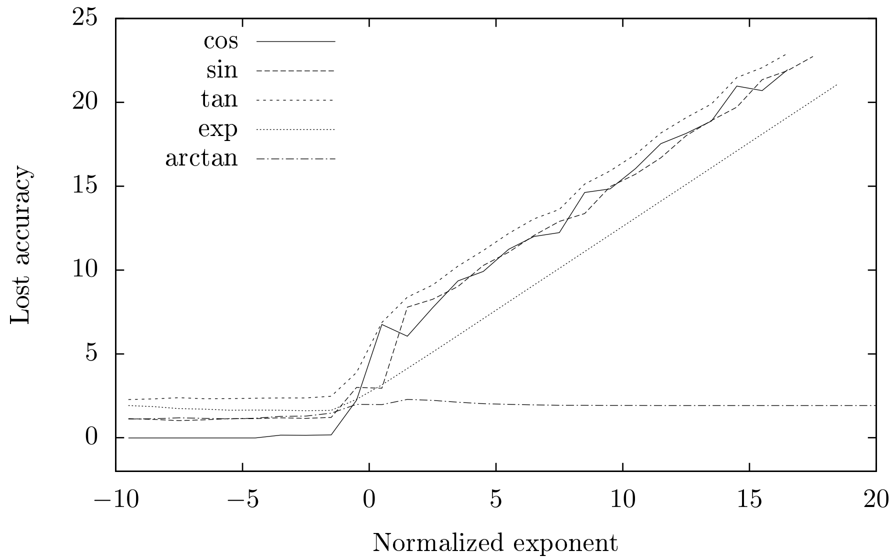

Figure 1: Relative width of the result depending on the normalized exponent of the input.

the elementary functions do not provide correct rounding, they still have an accuracy similar to the basic arithmetic operators.

4.3. Time complexity

Figure 2 shows the time it takes to Coq for computing the result of an elementary function ($\beta = 2$) depending on the internal precision. Note that the input value is reduced to 0.4 which will be used while evaluating the power series. The cos plot is identical to the sin one: The cost of the square root while rebuilding sin is negligible with respect to the other operations.

The last three plots are for the arctan function. Indeed, since the input value is big, the computations will have to access the cache containing enclosures of $\frac{\pi}{4}$. The plots correspond to various cache behaviors. The first plot happens with a hot cache: $\frac{\pi}{4}$ is already available. The second plot happens with a cold cache: Coq has to fill it. As shown on the figure, initializing the cache takes a huge time at higher precisions. Indeed, since the cache is stored inside a co-inductive list, accessing the value at a given precision of the constant requires to initialize all the values at lower precisions. This issue can be avoided by adding an indirection: Instead of directly storing the constants in the list, one can store trivial co-inductive objects that do compute the constants. This method gives the third plot. It is the one finally implemented.

Some regular steps are visible on the plots, especially for arctan at higher precisions. These steps are most certainly related to the way fast integers are

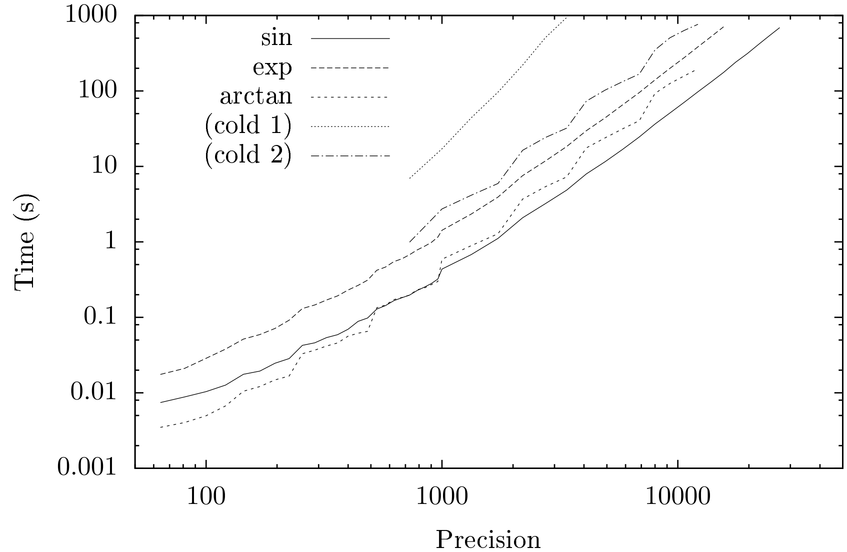

Figure 2: Time for computing a function at point $111111111111 \cdot 2^{-30} \simeq 103.5$.

stored: binary trees with 31-bit native integers at the leaves. Indeed, the steps occur around precisions $31 \cdot 2^5 = 992$, $31 \cdot 2^6 = 1984$, $31 \cdot 2^7$, and so on.

The slope of the exp and sin plots is about 2.4, which means that the time complexity of these functions is asymptotically $p^{2.4}$.

4.4. MPFR test data

The elementary functions of this library have been compared to the corresponding MPFR radix-2 functions. The MPFR library is shipped with test files containing input values and the expected results for these functions at a precision of 53 bits. All these input values were tested with the Coq implementation in order to compare it with MPFR performance-wise.

The strategy for computing the results in Coq is as follows. An interval enclosing the exact result is computed with an internal precision p_i. If both bounds round to the same floating-point number at precision 53, then this number is the correctly-rounded result. Otherwise the interval is too wide and the computation starts again at precision $p_{i+1} = \lfloor p_i \times \frac{3}{2} \rfloor$. And so on until a result is returned. The starting precision $p_1 = 63$ is arbitrarily chosen, as is the $\frac{3}{2}$ multiplier. Note that this Ziv strategy does not require any termination proof: Since the input set is fixed, a suitable upper bound on i can be found by experimenting.

Table 1 gives the average slowdown caused by using Coq instead of MPFR to compute correctly-rounded results for the test data. Note that these results are only meant to give a rough idea of the relative performance of the implementations, since the evaluation strategy is arbitrary and the input values are unusually large. The table shows the slowdown depending on the integer type the library is instantiated with. The first column is obtained for fast binary-tree integers, while the second column is obtained for standard bit-list integers.

Function	Coq fast ints	Coq std ints
cos	1700×	12000×
sin	2000×	40000×
tan	1900×	37000×
exp	3300×	21000×
arctan	470×	2000×

Table 1: Slowdown of Coq functions with respect to MPFR functions.

First noticeable point: Computing a value with Coq is about 2000 times slower than computing it with an optimized C program. This is the expected slowdown magnitude when programming inside the formal system. It is hopefully acceptable for proving some theorems with numerical computations.

Second point: Using standard integers only incurs an additional slowdown of 7× for arctan and cos with respect to fast integers. This one is a bit puzzling: Since integer computations are much faster on the processor, the slowdown should be bigger. This may be explained by the small size of the integers. Moreover, some common operations are trivial on standard integers, e.g. shifts.

Third point: There is yet another slowdown when Coq computes sin and tan with standard integers. It is due to the use of a square root when reconstructing the final result. The square root on standard integers does not only return the integer result, but it also returns a proof that this integer is the correct result. In the type-theoretical setting of Coq, this proof is an explicit λ-term whose body and type depend on the integer input. Coq wastes a long time (65% of the total time) building the whole proof term, while it is irrelevant from a computational point of view.[4] This explains why the binary-tree integers are much faster there: The correctness of their square root is guaranteed by a separate theorem instead of being carried by the code.

These tests also helped to detect an unexpectedly unproven (and wrong!) theorem in the Coq library that formalizes native 31-bit integer computations.[5] They also showed that the test values of MPFR were not as difficult as they could be, since the Coq functions could almost always get the correct result at

[4]By using a lazy evaluation scheme, the proof term would never be computed, since it is not used. It would, however, slow down all the other computations since none of them would benefit from lazy computations.

[5]The correctness proof for the 62-bit square root algorithm was not proved in the standard library of Coq, only admitted. In fact, it would have been impossible to prove it, as the Newton iteration used in the algorithm was actually incorrect. This was detected thanks to discrepancies with MPFR results for large arguments to the sin function.

the second iteration, hence with an internal precision of 94 bits.

4.5. Reducing arguments further

The functions of the library only perform an argument reduction until the input is smaller than $\frac{1}{2}$. This is unusual [13], since a smaller argument will speed up the power series evaluation. Figure 3 shows the time variation depending on the threshold chosen for the reduction of exp. To account for the longer reconstruction, the precision of the series evaluation was increased by one whenever the threshold was divided by two. The plot shows that reducing an argument until between 2^{-7} and 2^{-13} is the best choice for MPFR data. When reducing the argument even further, the cost of the reconstruction steps becomes too large, hence negating the speedup of the power series.

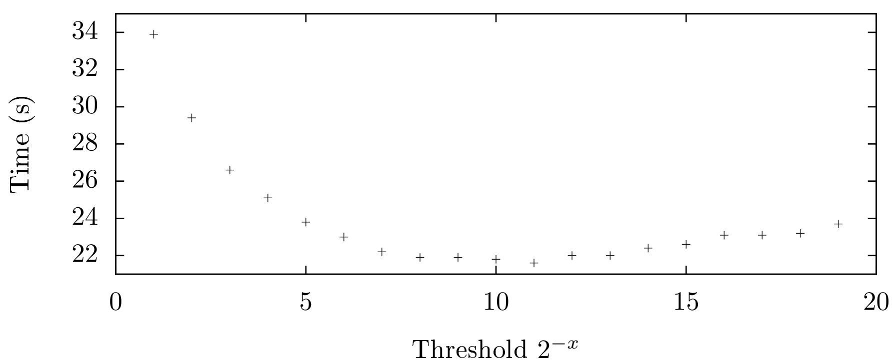

Figure 3: Time needed for computing exp on MPFR data depending on the argument-reduction threshold.

Therefore, changing the threshold brings the evaluation of exp in the same speed range as the trigonometric functions, about $\times 2000$ slower than MPFR. Note that this optimal threshold cannot be generalized to other precisions, since the number of operations during the evaluation of the power series depends on the precision, while the reconstruction depends only on the magnitude of the input.

For the trigonometric functions, the speedup when reducing the threshold is less noticeable. This may be due to the small amount of information available in an approximation of $\cos x$ for x small. So one could hope to achieve some speedup by evaluating instead a function $\operatorname{umc} x = 1 - \cos x$ and using the argument reduction

$$\operatorname{umc}(2 \cdot x) = 2 \cdot (\operatorname{umc} x) \cdot (2 - \operatorname{umc} x).$$

Unfortunately, it does not.

5. Conclusion

5.1. Proving theorems

Implementing floating-point arithmetic inside a formal system is hardly useful on its own. This arithmetic is only a means to efficiently prove mathematical theorems. So some Coq tactics have been developed. They rely on floating-point computations in order to automatically prove bounds on real-valued expressions [5]. They perform interval arithmetic and their usage is similar to those of some existing PVS strategies [14]. In addition, these tactics support bisection search, and they rely on a bit of automatic differentiation for doing interval evaluation with Taylor's order-1 decomposition. As a consequence, these Coq tactics are able to automatically handle a theorem originally proved in PVS [15]. The exact same formal methods are used for proving this theorem, but the Coq tactics do not need to rely on an external oracle.

The proved theorem states a tight bound on the relative error between Earth local radius

$$r_p(\phi) = \frac{a}{\sqrt{1 + (1 - f)^2 \times \tan^2 \phi}}$$

and a degree-10 polynomial with single-precision floating-point coefficients that is approximating it. The original PVS proof was composed of about 10000 generated scripts. It took several hours to check all of them on a 48-core machine in 2005. The Coq proof, a 25-line script, took a few minutes to check on a single core in 2008.

While the Coq tactics are performing too many computations because there is no oracle, they benefit from the use of floating-point arithmetic. Indeed, PVS' strategies are using rational numbers as interval bounds. As a consequence, the numbers that appear in the intermediate computations of the PVS proof carry thousands of digits, since all the bounds are computed exactly.[6] On the contrary, all the floating-point bounds in the Coq proof are rounded outwards to a precision of 30 bits. So the computations are not slowed down by the size of the numbers, which explains the tremendous speedup that is achieved on this example.

Coq was also used to check the correctness of a single-precision polynomial approximation of exp [16]. Again, it only took a few minutes for the formal system to automatically prove that the approximation is accurate enough.

5.2. Future works

The floating-point formalization described in this article is part of the Coq library available at

`http://www.lri.fr/~melquion/soft/coq-interval/`

[6]Note that the PVS computations are only exact for truncated power series, not for the complete elementary functions.

The long-term goal is to develop and adapt numerical methods for a proof assistant, so that computational proofs like Hales' one can be completely checked inside a formal system. Implementing a floating-point arithmetic inside Coq is a step toward this goal, and there is still space for improvements.

Obviously, new elementary functions should be added, so that the usual ones at least are available. Fortunately, with a suitable argument reduction, functions like log can also be expressed as alternating series. So their implementation and formal proof should closely match the ones for the existing functions, hence making them straightforward.

More importantly, the interval evaluation of elementary functions should be replaced by a static error analysis. There are currently no formal methods for doing this kind of analysis for multi-precision algorithms, so this will first require to build a comprehensive formalism. Not counting the ability to actually certify multi-precision algorithms, there are two benefits to a formalized static analysis. First, removing intervals will speed up the functions a bit. Second, it will allow us to implement Ziv's strategy [12] and get correctly-rounded results without relying too much on axioms.

While correct rounding is not needed for proofs, it would be a great help in writing MPFR-based oracles. Indeed, since correct rounding allows for portable results, a numerical computation that succeeds in the oracle would also succeed in Coq. As a consequence, it would become possible to carefully craft the oracle so that the proofs it generates need as few computations as possible to prove a given theorem in Coq.

The other way around, having portable results makes it possible to directly perform the extraction of a high-level numerical algorithm written in Coq to a compilable language (e.g. Ocaml) with bindings to the arithmetic operators and elementary functions of MPFR. That way, both the development and the certification of a numerical application could be done in Coq, while its execution would be as fast as currently possible for a multi-precision code.

References

[1] G. Gonthier, Formal proof – the four-color theorem, Notices of the AMS (2008) 1382–1393.

[2] Y. Bertot, P. Castéran, Interactive Theorem Proving and Program Development. Coq'Art: the Calculus of Inductive Constructions, Texts in Theoretical Computer Science, Springer-Verlag, 2004.

[3] B. Grégoire, L. Théry, A purely functional library for modular arithmetic and its application to certifying large prime numbers, in: U. Furbach, N. Shankar (Eds.), Proceedings of the 3rd International Joint Conference on Automated Reasoning, Vol. 4130 of Lectures Notes in Artificial Intelligence, Seattle, WA, USA, 2006, pp. 423–437.

[4] T. C. Hales, A proof of the Kepler conjecture, Annals of Mathematics 162 (2005) 1065–1185.

[5] G. Melquiond, Proving bounds on real-valued functions with computations, in: A. Armando, P. Baumgartner, G. Dowek (Eds.), Proceedings of the 4th International Joint Conference on Automated Reasoning, Vol. 5195 of Lectures Notes in Artificial Intelligence, Sydney, Australia, 2008, pp. 2–17.

[6] D. Stevenson, et al., An American national standard: IEEE standard for binary floating point arithmetic, ACM SIGPLAN Notices 22 (2) (1987) 9–25.

[7] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, P. Zimmermann, MPFR: A multiple-precision binary floating-point library with correct rounding, ACM Transactions on Mathematical Software 33 (2).

[8] D. M. Russinoff, A mechanically checked proof of IEEE compliance of the AMD-K7 floating point multiplication, division, and square root instructions, LMS Journal of Computation and Mathematics 1 (1998) 148–200.

[9] J. Harrison, A machine-checked theory of floating-point arithmetic, in: Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, L. Théry (Eds.), Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics, Vol. 1690 of Lecture Notes in Computer Science, Nice, France, 1999, pp. 113–130.

[10] M. Daumas, L. Rideau, L. Théry, A generic library of floating-point numbers and its application to exact computing, in: Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics, Edinburgh, Scotland, 2001, pp. 169–184.

[11] M. D. Ercegovac, T. Lang, Digital Arithmetic, Morgan Kaufmann Publishers, 2004.

[12] A. Ziv, Fast evaluation of elementary mathematical functions with correctly rounded last bit, ACM Transactions on Mathematical Software 17 (3) (1991) 410–423.

[13] J.-M. Muller, Elementary Functions, Algorithms and Implementation, Birkhauser, Boston, 1997.

[14] C. Muñoz, D. Lester, Real number calculations and theorem proving, in: J. Hurd, T. Melham (Eds.), Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics, Vol. 3603 of Lecture Notes in Computer Science, Oxford, UK, 2005, pp. 195–210.

[15] M. Daumas, G. Melquiond, C. Muñoz, Guaranteed proofs using interval arithmetic, in: P. Montuschi, E. Schwarz (Eds.), Proceedings of the 17th IEEE Symposium on Computer Arithmetic, Cape Cod, MA, USA, 2005, pp. 188–195.

[16] P.-T. P. Tang, Table-driven implementation of the logarithm function in IEEE floating-point arithmetic, ACM Transactions on Mathematical Software 16 (4) (1990) 378–400.