Floating-point arithmetic in the Coq system

Guillaume Melquiond
INRIA – Microsoft Research
guillaume.melquiond@inria.fr

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.

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 these systems, especially in those based on type-theoretical formalisms, the ability to efficiently perform calculations is steadily increasing. As a consequence, the process of proving mathematical theorems is slowly shifting from a deductive approach to a computational approach.

One of these formal systems is the Coq proof assistant [1], 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 powerful mathematical results.

Irrespective of these considerations, floating-point arithmetic is widely used in computer programs as a fast approximation of the traditional arithmetic on 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. This hinted at formalizing an effective floating-point arithmetic inside the Coq system.

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 radices 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 realisability 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 [9] is a radix-2 arithmetic with $p = 53$, $e_{\min} = -1049$, and $e_{\max} = 971$. A multi-precision library like MPFR [5] 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 of 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, 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 phenomenons can no longer occur during computations. In particular, 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 nevertheless 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:

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

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.

The library contains the following Coq theorem. It states that the floating-point division algorithm is correct: 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 than the real quotient $\frac{x}{y}$ once rounded to a floating-point number. Note that this real quotient and its rounding are defined in a purely axiomatic way in Coq: Standard real numbers are an abstract type useless for computations.

```
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 ..."

2.3 Computability

Several formalizations exist, for various proof assistants, and they have been successful in proving numerous facts on floating-point arithmetic. For the sake of conciseness, only two of them will be cited here [3, 7]. 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 actually computing. 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

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

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.

Note that the theorems provided by these libraries 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 radices 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 correction. 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 the integers as 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 [6]. 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 leafs of the trees. This considerably reduces the memory footprint of the integers, and makes it possible to delegate arithmetic computations on the leafs 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 MPFR's laudable philosophy: "The complexity should, while preserving correct rounding, depend on the precision required on the result rather than on the precision given on the operands." 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 floating-point number.

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 [4]. 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:

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

For arbitrary numbers x and $scale$, the position and m are not computable.[1] 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

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

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 $(m_1 \cdot \beta^n, \mathtt{pos\_Eq})$ 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.

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 $(s, \mathtt{pos\_Eq})$ 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 [10].

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, 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 that depend on this floating-point library do not need the correct rounding.

The library currently supports the functions $\cos$, $\sin$, $\tan$, $\arctan$. The implementation of other elementary functions like $\exp$ and $\log$ 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

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

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

From $|x| \leq \frac{1}{2}$, 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 $\beta \leq 4$, 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 that permits 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 to define and to prove an algorithm for evaluating truncated summation with an absolute error less than β^{-k} too. As a consequence, the relative error is less than β^{k-2} when computing $\arctan$ this way, since $|x| \leq \frac{1}{2}$. We therefore have a way to compute arbitrarily small ranges enclosing $\arctan x$.

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-2} 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

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:

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

These functions have been arbitrarily restricted to values smaller than 2^{20}. Outside this domain, they return NaN. This should hopefully be sufficient for the purpose of proving theorems with numerical computations.

The $\arctan$ function, however, is defined on the whole set of floating-point numbers. The argument 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:

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

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 destroyed, Coq computes the value of its head. Moreover, Coq remembers this value, so that it can be instantly reused the next time a function destroys the co-inductive object. Therefore the interval constant for π at a given precision (with a granularity of 31 digits) is computed only once per Coq session.

5 Conclusion

5.1 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. Except for $\arctan$, the test files were trimmed to inputs smaller than 2^{20} due to the argument reduction described in Section 4.2.

The strategy for computing the values 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 is arbitrarily set to 63, as is the $\frac{3}{2}$ multiplier. Note that this Ziv strategy does not require any termination proof, since the input set is fixed, so a suitable upper bound on i can be found by experimenting.

The following table gives average values for the number of processor cycles it takes to compute the correctly-rounded result for every test input on an Intel Core 2 processor. Note that these results are only meant to give a rough idea of the relative performance of the implementations, since the evaluation strategy is plain arbitrary and the input values are unusually large. The second column contains the number of input values, and the ratio with respect to the non-trimmed files. The third and fourth columns show the speed of Coq, when using fast binary-tree integers and standard $\mathtt{Z}$ integers respectively.

Function	Tested values	Coq speed	Coq Z speed	MPFR speed
$\arctan$	870 (100%)	$20 \cdot 10^6$	$10 \cdot 10^7$	$57 \cdot 10^3$
$\cos$	1289 (91%)	$27 \cdot 10^6$	$20 \cdot 10^7$	$21 \cdot 10^3$
$\sin$	1120 (89%)	$32 \cdot 10^6$	$81 \cdot 10^7$	$21 \cdot 10^3$
$\tan$	1297 (81%)	$36 \cdot 10^6$	$83 \cdot 10^7$	$22 \cdot 10^3$

First noticeable point: Computing a value with Coq is about 1000 times slower than computing it with an optimized C program. This is the expected magnitude and it may be acceptable for proving theorems with numerical computations. Second, using the standard integers only incur an additional slowdown of 7 for $\arctan$ and $\cos$. This one is a bit surprising, since integer computations are much faster on the processor, so the slowdown should be bigger. This may be explained by the small size of the integers and by the fact that some common operations are trivial on standard integers, e.g. shifts.

Third, there is yet another slowdown when Coq computes $\sin$ and $\tan$ with standard integers. This is due to the square root when reconstructing the final result. The square root on standard integers

not only returns the integer result, it also returns a proof that this integer is the correct result. In the type-theoretical setting of Coq, this proof is an explicit function whose type depends on the integer input. Therefore Coq wastes a long time building this term, while it is irrelevant from a computational point of view. This explains why the binary-tree integers are much faster there, as the correctness of their square root is not embedded in its computation but it is guaranteed by a separate theorem.

These tests also helped to detect an unexpectedly unproven (and wrong!) theorem in the Coq library that formalizes native 31-bit integer computations. 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 the second iteration, hence with an internal precision of 94 bits.

5.2 Proving theorems

Implementing floating-point arithmetic in Coq is only a way to efficiently prove mathematical theorems. So some tactics have been developed in order to automatically prove bounds on real-valued expressions. These tactics are based on interval arithmetic and are similar to some existing PVS strategies [8]. They also support bisection search, and 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 [2] by using the exact same formal methods but without relying on an external oracle. This theorem was stating 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 was approximating it. The original PVS proof was composed of about 10000 generated scripts and 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' interval strategies are using rational numbers as 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. On the contrary, the floating-point bounds in the Coq proof were all rounded outwards to a precision of 30 bits. So the computations were not slowed down by the size of the numbers, which explains the tremendous speed up that was achieved on this example.

5.3 Future works

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

`http://www.msr-inria.inria.fr/~gmelquio/soft/coq-interval/`

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 $\exp$ and $\log$ 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 to implement Ziv's strategy 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] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: the Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer-Verlag, 2004.

[2] M. Daumas, G. Melquiond, and C. Muñoz. Guaranteed proofs using interval arithmetic. In P. Montuschi and E. Schwarz, editors, *Proceedings of the 17th IEEE Symposium on Computer Arithmetic*, pages 188–195, Cape Cod, MA, USA, 2005.

[3] M. Daumas, L. Rideau, and 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*, pages 169–184, Edinburgh, Scotland, 2001.

[4] M. D. Ercegovac and T. Lang. *Digital Arithmetic*. Morgan Kaufmann Publishers, 2004.

[5] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software*, 33(2), 2007.

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

[7] J. Harrison. A machine-checked theory of floating-point arithmetic. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics*, volume 1690 of *Lecture Notes in Computer Science*, pages 113–130, Nice, France, 1999.

[8] C. Muñoz and D. Lester. Real number calculations and theorem proving. In J. Hurd and T. Melham, editors, *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics*, volume 3603 of *Lecture Notes in Computer Science*, pages 195–210, Oxford, UK, 2005.

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

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