End-to-End Formal Verification of a Fast and Accurate Floating-Point Approximation

Florian Faissole
Mitsubishi Electric R&D Centre Europe, 35700 Rennes, France

Paul Geneau de Lamarlière
Mitsubishi Electric R&D Centre Europe, 35700 Rennes, France
Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, LMF, 91190 Gif-sur-Yvette, France

Guillaume Melquiond
Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, LMF, 91190 Gif-sur-Yvette, France

Abstract

Designing an efficient yet accurate floating-point approximation of a mathematical function is an intricate and error-prone process. This warrants the use of formal methods, especially formal proof, to achieve some degree of confidence in the implementation. Unfortunately, the lack of automation or its poor interplay with the more manual parts of the proof makes it way too costly in practice. This article revisits the issue by proposing a methodology and some dedicated automation, and applies them to the use case of a faithful *binary64* approximation of exponential. The peculiarity of this use case is that the target of the formal verification is not a simple modeling of an external code, it is an actual floating-point function defined in the logic of the Coq proof assistant, which is thus usable inside proofs once its correctness has been fully verified. This function presents all the attributes of a state-of-the-art implementation: bit-level manipulations, large tables of constants, obscure floating-point transformations, exceptional values, etc. This function has been integrated into the proof strategies of the CoqInterval library, bringing a 20× speedup with respect to the previous implementation.

2012 ACM Subject Classification Software and its engineering → Formal software verification; Theory of computation → Interactive proof systems; Theory of computation → Automated reasoning; Mathematics of computing → Mathematical software performance; Mathematics of computing → Interval arithmetic

Keywords and phrases Program verification, floating-point arithmetic, formal proof, automated reasoning, mathematical library

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

The CoqInterval library[1] provides a set of strategies for the Coq proof assistant that automatize the formal verification of enclosures of real-valued expressions. It is based on a formalization of rigorous polynomial approximations that are computed using an interval arithmetic with floating-point bounds [9]. Originally, the floating-point computations were performed one bit at a time in the logic of the Coq system. But now that hardware floating-point computations can be performed inside Coq proofs, CoqInterval's strategies can delegate some computations to the floating-point unit of the processor, thus greatly speeding up the proof checking [10]. This makes it possible to formally verify the following approximation of Siegfried Rump's integral, which is known to cause computer algebra systems to struggle

[1] `https://coqinterval.gitlabpages.inria.fr/`

due to the large number of oscillations of the integrand, in a handful of seconds:

$$\int_0^8 \sin(x + \exp x)dx = 0.3474 \pm 10^{-6}.$$

Among numerous other things, this proof requires the ability to compute a floating-point enclosure of the mathematical value $\exp x$ for x some floating-point number. Thanks to a suitable abstraction of floating-point arithmetic [10], CoqInterval uses the same algorithm for both floating-point numbers computed in hardware and floating-point numbers slowly emulated in the logic of Coq.[2] Having a single algorithm, and thus a single proof of correctness, for both implementations of floating-point arithmetic made the large formalization effort that went into adding hardware computations to CoqInterval much less tedious. The algorithm for computing an enclosure of $\exp x$ goes as follows. First, using the following mathematical identities, an argument reduction brings the input x into the interval $[-2^{-8}; 0]$:

$$\begin{aligned} \exp x &= (\exp(-x))^{-1} \\ \exp x &= (\exp(x/2))^2 \end{aligned} \tag{1}$$

Second, the alternating series $\exp(-x) = \sum(-x)^n/n!$ is computed using interval arithmetic to a high enough order. Thanks to the use of interval arithmetic and an alternating series, the algorithm is guaranteed to compute an enclosure of the real number $\exp x$. Third, the argument reduction is inverted to reconstruct the final interval result.

By suitably choosing the order of truncation of the series, one can obtain arbitrarily tight enclosures of $\exp x$, assuming that the precision of the floating-point arithmetic used to compute the interval bounds can be made accordingly large. This property is invaluable when used in conjunction with the original multi-precision floating-point arithmetic of CoqInterval. But for hardware floating-point numbers and their fixed precision of 53 bits, the property is pointless. The inadequacies of the implementation of exp then become prominent. First, Equation (1) means that computing an enclosure of $\exp x$ is not constant time, but proportional to the magnitude of x. Second, an alternating series is the worst way of approximating a value, as part of the computations performed at order i are immediately canceled by those at order $i + 1$ and thus have been performed in vain. Third, while interval arithmetic is correct by construction, hence very proof-friendly, it performs twice as many floating-point operations as needed.

Outside of a formal system, a state-of-the-art approximation of the exponential function using hardware floating-point numbers would be implemented along the following guidelines [12, §6.2]. It would first perform a constant-time argument reduction using the following mathematical identity:

$$\exp x = \exp(x - k \cdot \ln 2) \cdot 2^k \quad \text{with } k = \lceil x/\ln 2 \rfloor \in \mathbb{Z}.$$

Then, a low-degree polynomial approximation of exp around 0 would be evaluated. Finally, the result reconstruction is trivial, as it is a multiplication by a power of two. The whole algorithm amounts to just a few tens of operations; it is thus extremely fast. The issue now is that one needs to formally verify the adequacy of the result, as it no longer derives from the use of interval arithmetic. In particular, if one wants to use such a state-of-the-art implementation of exp in place of the one currently in CoqInterval, the proof effort needs to be sufficiently light so that it is worth replacing a feature that is already good enough for most use cases.

[2] This emulation is still useful for proofs that require more than the 53 bits of precision provided by the *binary64* format.

```
let iexp x =
  if x < -0x1.74385446d71c4p9 then (0., 0x1.p-1074) else
  if x > 0x1.62e42fefa39efp9 then (0x1.fffffffffff2ap1023, infinity) else
  let k' = x *. invLog2_64 +. 0x1.8p52 in
  let k = k' -. 0x1.8p52 in
  let t = (x -. k *. log2_64h) -. k *. log2_64l in
  let y = t *. (p1 +. t *. (p2 +. t *. (... + t *. p5))) in
  let ki = int_of_float k' - 0x18000000000000 in
  let p0 = cst.(ki land 63) in
  let lb = p0 +. (p0 *. y -. d) in
  let ub = p0 +. (p0 *. y +. d) in
  next_down (ldexp lb (ki asr 6)), next_up (ldexp ub (ki asr 6))
```

Figure 1 Guaranteed approximation of exponential in OCaml. The output is a pair of floating-point numbers that enclose $\exp x$. Symbols `invLog2_64`, `log2_64h`, `log2_64l`, `d`, `cst`, `p1`, `p2`, etc, are predefined floating-point literals; in particular, `d` is about $3 \cdot 2^{-58}$.

Figure 1 shows the implementation we have devised, represented as an OCaml function for readability. (Its translation to Coq's λ-calculus is straightforward.) Given a finite floating-point number x, the code computes a pair of floating-point numbers that enclose $\exp x$. In particular, it uses the functions `next_down` and `next_up` to compute the predecessor and successor of a floating-point number.

While the code seems to contain useless, if not adverse, floating-point computations, this is not the case. For example, `k` looks like it could be directly computed as `x *. invLog2_64` by canceling `0x1.8p52 -. 0x1.8p52`. This optimization would completely break the function, causing it to no longer approximate the exponential, not even roughly. Similarly, `t` should not be rewritten as `x -. k *. (log2_64h +. log2_64l)`, and `d` should not be moved outside of the parentheses in the computations of `lb` and `ub`. So, not only does floating-point arithmetic not respect the usual algebraic laws of associativity and distributivity, floating-point experts actively rely on the lack of these laws to compute more accurate approximations.

As for the accuracy of the code, one can get an intuitive feel of it by considering the distance between both components of the returned pair. Ideally, the distance should be one unit in the last place (*ulp*), as the components should be consecutive floating-point numbers for $x \neq 0$. This property, called *correct rounding* [12, §12.3], is still an open research question for floating-point formats larger than *binary32* and completely out of reach of a formal proof, as of today. So, the best we can hope to achieve is a distance of up to two ulps, that is, one component is optimal, while the other is off-by-one. If the constant `d` was zero, this would be the case. As it is not quite zero here, when $\exp x$ is close to the midpoint between two consecutive floating-point numbers, the distance might end up being three ulps. The proportion of inputs that cause a 3-ulp interval output is $d \cdot 2^{51} \simeq 1/40$.

There have been several attempts at formally verifying this kind of state-of-the-art implementation using the Coq proof assistant, but they all have suffered from various shortcomings. It might have been that the floating-point arithmetic was modeled without any exceptional value [3, §6.2.3]. Indeed, when a computer-assisted proof is meant to complement a pen-and-paper proof, it is acceptable that it only focuses on the most intricate parts of the proof, which the absence of exceptional behavior is hardly ever. But, since this idealized arithmetic does not match the behavior of hardware floating-point numbers, it cannot be used here. Some later attempt solved the issue of the exceptional values [6], but it was still

targeting the verification of some code meant to run outside of Coq and thus did not need to cover all of its facets. For example, it was ignoring the first and last few steps of the algorithm, so as to focus on the important part. It was also assuming the existence of some higher-level functions like `nearbyint`, which are either missing or too slow in our setting. Moreover, it was ignoring the issue of accuracy on subnormal outputs, where it is ill-defined, but that we cannot just disregard. Finally, the code was not as intricate as the one presented in Figure 1: no array accesses, no mixed integer/floating-point operations, etc.

The novelty of the work presented in this article is thus the full verification of a state-of-the-art floating-point implementation of a mathematical function. This verification really covers all the facets, since the algorithm is not just modeled, but it is an actual code that will effectively be run when checking subsequent Coq proofs, so absolutely no shortcuts can be taken. This article also presents all the automated strategies that were added to make this verification as painless as possible. Indeed, the exponential function is just the first step; the goal is to optimize all the mathematical functions of CoqInterval using the presented methodology.

Section 2 reminds both the arithmetic language and the notion of well-behaved expression that were introduced in a previous work [6]. Section 3 explains how some strategies of CoqInterval have been improved to automatically verify properties involving tight bounds on rounding errors. Section 4 details the new features added to the arithmetic language and associated tools to tackle the algorithm of Figure 1: hardware floating-point numbers, conversions, macro-operations, array accesses, etc. Section 5 describes the methodology used to formally verify the correctness of exponential, as well as some unusual properties of floating-point arithmetic we ended up with. Section 6 explains how this work relates to some other works. Finally, Section 7 concludes with some benchmarks and some perspectives.

2 Preliminaries

This work is partly built on top of a framework for modeling floating-point expressions [6]. In particular, that framework provides some facilities to automate the proof of the absence of exceptional behaviors, thus making it possible for the user to focus on a modeling of floating-point expressions as real numbers. This section reminds the features of that framework that are the most relevant to the presented work. Section 2.1 focuses on the expressions and their various interpretations, while Section 2.2 shows how one can jump between interpretations to ease the proof process.

In the following, the unary operator $\circ(\cdot)$ designates a rounding operator from $\mathbb{R}$ to $\mathbb{R}$; it returns the real number the nearest to the input that fits in the target floating-point format (with unlimited range) [3, §3.2.2]. This theoretical operator is at the core of the IEEE-754 standard for floating-point arithmetic.

2.1 Arithmetic expressions

An arithmetic expression e is represented as the value of an inductive type corresponding to a typed abstract syntax tree, namely an expression tree [6]. The nodes of an expression tree correspond to arithmetic expressions, including floating-point operations, integer operations, and some functions such as `nearbyint`.

The expression e can then be interpreted in several ways, two of which are relevant here. First, it can be interpreted as the floating-point number $[\![e]\!]_{\mathsf{flt}}$ that would be obtained according to the IEEE-754 standard. Second, e can be interpreted as the value $[\![e]\!]_{\mathsf{rnd}}$ obtained by performing all the operations on real numbers and rounding their results. For example, in

the case of the floating-point addition, we have $[\![u+v]\!]_{\mathsf{rnd}} = \circ([\![u]\!]_{\mathsf{rnd}} + [\![v]\!]_{\mathsf{rnd}})$. In the case of integer operations, $[\![e]\!]_{\mathsf{flt}}$ performs computations modulo a power of two, while $[\![e]\!]_{\mathsf{rnd}}$ performs operations on unbounded integers, *e.g*, $[\![u+v]\!]_{\mathsf{rnd}} = [\![u]\!]_{\mathsf{rnd}} + [\![v]\!]_{\mathsf{rnd}}$. The first interpretation corresponds to the value actually computed by an implementation, and therefore the value on which we need to prove a correctness theorem. The second interpretation, however, is the one that is the more amenable to formal reasoning, as it is not susceptible to exceptional behaviors such as overflows.

There are two features of expression trees that are of interest to us. The first one is the support for let-binding operators, with binders represented by their de Bruijn indices, to express sharing between sub-expressions and to guide proofs. The second one is the availability of exact arithmetic operations, as they are commonly encountered in implementation of mathematical functions. As far as $[\![\cdot]\!]_{\mathsf{flt}}$ is concerned, there is no difference in interpretation between exact and inexact operations over floating-point numbers; they are performed as mandated by the IEEE-754 standard. For $[\![\cdot]\!]_{\mathsf{rnd}}$, exact arithmetic operations, however, are not rounded, which makes formal proofs, both manual and automatic, much simpler. This rises the concern of whether such a proof about $[\![e]\!]_{\mathsf{rnd}}$ is meaningful, which Theorem 1 below will tackle.

Let us illustrate these two features on the example of the argument reduction of the Cody-Waite exponential [4], variants of which are still widely used in modern implementations, including in the code shown in Figure 1:

$$k \leftarrow \texttt{nearbyint}(x \cdot C),$$
$$t \leftarrow x - k \cdot c_1 - k \cdot c_2,$$

with $c_1 + c_2 \simeq 1/C$ and $c_2 \ll c_1$. Below is the formulation of this argument reduction as an expression tree.

```
Let (NearbyInt (Op MUL (Var 0) (BinFl C)))
(Op SUB
  (OpExact SUB (Var 1) (OpExact MUL (Var 0) (BinFl c1)))
  (Op MUL (Var 0) (BinFl c2)))
```

Notice that both floating-point operations in $x - k \cdot c_1$ are annotated as exact operations by using the `OpExact` constructor. All the other operations are marked as potentially inexact (`Op` constructor). This gives the following value for $[\![t]\!]_{\mathsf{rnd}}$:

$$\circ(x - k \cdot c_1 - \circ(k \cdot c_2)) \quad \text{with } k = \lfloor \circ(x \cdot C) \rceil.$$

2.2 Relation between interpretations

As mentioned earlier, a correctness statement is about $[\![e]\!]_{\mathsf{flt}}$, while a user only wants to have to deal with $[\![e]\!]_{\mathsf{rnd}}$, as it is free of exceptional behaviors and contains fewer rounding operations. In order to bridge the gap between both interpretations, a predicate WB (for *well-behaved*) is defined recursively over expressions. For example, the proposition WB(`Op DIV` u v) is defined as

$$\mathrm{WB}(u) \wedge \mathrm{WB}(v) \wedge [\![v]\!]_{\mathsf{rnd}} \neq 0 \wedge |\circ([\![u]\!]_{\mathsf{rnd}} / [\![v]\!]_{\mathsf{rnd}})| \leq \Omega$$

with Ω the value of the largest finite floating-point number. In other words, for the floating-point division u/v to be well-behaved, it is sufficient that u and v are well-behaved, that the interpretation of v as a real number is non-zero, and that the division over real numbers,

once rounded, does not overflow the floating-point format. The predicate WB is defined in a similar way for the other inexact operations over floating-point numbers. For exact operations, the formula contains an additional conjunct that states that the result is exactly representable. For example, the proposition WB(`OpExact ADD` u v) is defined as

$$\mathrm{WB}(u) \wedge \mathrm{WB}(v) \wedge \circ(\llbracket u \rrbracket_{\mathsf{rnd}} + \llbracket v \rrbracket_{\mathsf{rnd}}) = \llbracket u \rrbracket_{\mathsf{rnd}} + \llbracket v \rrbracket_{\mathsf{rnd}} \wedge |\llbracket u \rrbracket_{\mathsf{rnd}} + \llbracket v \rrbracket_{\mathsf{rnd}}| \leq \Omega.$$

The key result is that, if an expression e is well-behaved, then $\llbracket e \rrbracket_{\mathsf{flt}}$ is a finite floating-point number and it represents the real number $\llbracket e \rrbracket_{\mathsf{rnd}}$. This is expressed by the following theorem:

▶ **Theorem 1.** *Given an expression* e, $\mathrm{WB}(e) \Rightarrow \llbracket e \rrbracket_{\mathsf{flt}}$ *finite* $\wedge\ \llbracket e \rrbracket_{\mathsf{flt}} = \llbracket e \rrbracket_{\mathsf{rnd}}$.

When applying Theorem 1, the user is left with a subgoal WB(e), which is painful to prove by hand. So, to ease the proof process, the framework proposes a proof strategy called `simplify_wb`, which tackles this subgoal by applying a procedure similar to CoqInterval's `interval` strategy to every conjunct of WB(e) individually. In practice, one can expect all the conjuncts related to the absence of exceptional behaviors to be automatically proved. Conjuncts related to exact operations, *e.g.*, $\circ(\llbracket u \rrbracket_{\mathsf{rnd}} + \llbracket v \rrbracket_{\mathsf{rnd}}) = \llbracket u \rrbracket_{\mathsf{rnd}} + \llbracket v \rrbracket_{\mathsf{rnd}}$, are however out of the scope of CoqInterval. So, the user will have to prove them either manually or using a dedicated tool like Gappa [3, §4.3].

Note that, in order for `simplify_wb` to make use of the `interval` strategy of CoqInterval, the latter had to be enhanced with some support for rounding operators, as they appear in almost all the conjuncts of WB(e). This support was based on the so-called *standard model* of floating-point arithmetic. (Section 3.2 will propose a better approach.) For instance, given an enclosure $u \in [\underline{u}; \overline{u}]$, CoqInterval would compute an enclosure of $\circ(u)$ as follows, assuming a *binary64* format:

$$\circ(u) \in [\underline{u} - \varepsilon; \overline{u} + \varepsilon] \quad \text{with } \varepsilon = \max\left(2^{-1075}, -2^{-53} \cdot \underline{u}, +2^{-53} \cdot \overline{u}\right). \tag{2}$$

3 Automated tools and rounding errors

Since support for rounding operators had been added to CoqInterval to suit `simplify_wb` [6], it seemed like it could be used for more than just automatically proving some conjuncts of WB(e). In particular, it should be possible to automatically prove a bound on the error between a rounded expression and the ideal expression it supposedly approximates. Consider the code of Figure 1. An intermediate property that is needed to prove its correctness is the following one:

$$\forall t \in \mathbb{R},\ |t| \leq 355 \cdot 2^{-16} \Rightarrow |1 + y - \exp t| \leq 2^{-58} \quad \text{with } y = \circ(t \cdot \circ(p_1 + \circ(t \cdot \circ(p_2 + \ldots)))). \tag{3}$$

Prior to this work, the methodology to prove the bound would have been to decompose the expression $1 + y - \exp t$ into two parts $e_1 + e_2$ [3, §6.2.3]: $e_1 = y - t \cdot (p_1 + t \cdot (p_2 + \ldots)))$ and $e_2 = (1 + t \cdot (p_1 + t \cdot (p_2 + \ldots))) - \exp t$. Expression e_2 (no rounding operator) would then be bounded using the rigorous polynomial approximations of CoqInterval, while expression e_1 (no exponential) would be bounded using the Gappa tool [3, §4.3]. This would give a proof of the expected bound 2^{-58}. Our goal is to perform this proof directly using CoqInterval, without any need for such algebraic manipulations nor the use of an external tool.

Unfortunately, several issues arise when using the `interval` strategy on Equation (3). First of all, both sides of the subtraction are strongly correlated by definition, since the left-hand side was chosen among the best possible floating-point approximations of the right-hand side $\exp t$. This means that naive interval arithmetic, as used in Equation (2) to

define the enclosure of a rounding operator, will cause an overestimation of the final enclosure that is so large that it becomes useless for proving anything interesting. On this example, the strategy would only be able to prove that the error is bounded by 10^{-2}, very far from the expected bound of 2^{-58}. So, the first step is to define rigorous polynomial approximations for rounding operators (Section 3.1).

This is not sufficient though, as the strategy would only succeed in proving a bound of about $1.5 \cdot 2^{-58}$, which is already quite good, but not sufficient to prove the correctness of the whole code. This overestimation is a consequence of the simplicity of the standard model of floating-point arithmetic, as it often causes bounds on rounding errors to be overestimated. So, the second step is to prove tighter bounds (Section 3.2). In the context of `simplify_wb`, both issues were insignificant, since the goal was to automatically prove that some expression $[\![e]\!]_{\mathsf{rnd}}$ is bounded by $\Omega \simeq 2^{1024}$, which is usually several hundreds of orders of magnitude larger than e.

3.1 Rigorous polynomial approximations

The correlation issue of naive interval arithmetic is well-known, and it is independent of rounding errors. In fact, even the interval evaluation of $(a + x) \cdot (b - x)$ would suffer from it, as $a + x$ and $b - x$ vary in opposite directions with respect to x. A first solution to this issue is to split the domain of x into smaller sub-intervals and to take the union of the enclosures of the whole expression on all these sub-intervals. This approach is very simple proof-wise, but it scales poorly computation-wise, so it should only be used as a last resort. A second solution is to compute enclosures whose bounds symbolically depend on x rather than being just numerical values. This approach scales better, but it requires a much larger formalization effort.

CoqInterval provides both approaches [9]. In particular, the second approach is implemented using rigorous polynomial approximations. Instead of just computing a single interval $[\underline{e}; \overline{e}]$ that encloses an expression $e(x)$ for any $x \in X$, it computes a polynomial P and an interval Δ such that, for any $x \in X$, we have $e(x) - P(x) \in \Delta$, which we denote by $e \in (P, \Delta)_X$. Those polynomial enclosures can then be composed. For example, if we have $f \in (P_f, \Delta_f)_X$ and $g \in (P_g, \Delta_g)_X$, we also have $f + g \in (P_f + P_g, \Delta_f + \Delta_g)_X$. This makes it possible to compute the polynomial enclosure of an arbitrary expression, by induction on its structure.

Therefore, to benefit from the rigorous polynomial approximations of CoqInterval, we need to be able to compute a polynomial enclosure of $\circ(u)$, given an enclosure $u \in (P, \Delta)_X$. To do so, we rewrite $\circ(u(x))$ into the sum $[\circ(u(x)) - u(x)] + u(x)$. For the left-hand side, we use the degree-0 enclosure $\circ(u) - u \in (0, [-\varepsilon; \varepsilon])_X$ with ε computed as in Equation (2). Then, by adding the original enclosure $(P, \Delta)_X$, we get a polynomial enclosure of $\circ(u)$.

This change to CoqInterval was straightforward, but it has shifted the perspective on rounding operators in the library. Indeed, the original implementation, which was designed for `simplify_wb`, computed an enclosure of $\circ(u)$ given an enclosure of u. Then, the user could ask for an enclosure of $\circ(u) - u$, which would be correct but overestimated. The new implementation computes a tight enclosure of $\circ(u) - u$ from an enclosure of u, from which it derives an enclosure of $\circ(u)$. This change has been propagated up to the surface language, that is, CoqInterval now recognizes the expression $\circ(u) - u$ as an atomic error for an expression u rather than a subtraction between two sub-expressions involving u.

3.2 Tighter error bounds

By adding support for rounding operators, CoqInterval is now able to automatically prove Equation (3), but only if the rightmost bound is changed to $1.5 \cdot 2^{-58}$. It fails for any tighter bound, especially for 2^{-58}, which we need to prove the correctness of the implementation of Figure 1. As mentioned earlier, the issue comes from the simplicity of the standard model of floating-point arithmetic, which states that the absolute error between $\circ(u)$ and u is bounded by $2^{-53} \cdot |u|$, assuming that u is in the normal range. While this is sensibly true for values of u slightly larger than a power of two, this is off by a factor two for values of u that are slightly smaller than a power of two. A better model of floating-point errors is to bound the absolute error between $\circ(u)$ and u by $\frac{1}{2}\mathrm{ulp}(u)$. In other words, given an enclosure $u \in [\underline{u}; \overline{u}]$, we have the following enclosure of the absolute error:

$$\circ(u) - u \in [-\tfrac{\varepsilon}{2}; \tfrac{\varepsilon}{2}] \quad \text{with } \varepsilon = \mathrm{ulp}(\max(-\underline{u}, \overline{u})).$$

This has required us to implement in CoqInterval an overestimation of ulp and to prove that it was sound with respect to the non-computational definition found in the Flocq library. Not only is this new enclosure tighter, but it separates the concerns about the target format and the rounding direction. Regarding the target format, one just has to chose the corresponding definition for ulp. As a consequence, CoqInterval now supports not only the floating-point formats of Flocq, but also its fixed-point formats. As for the rounding direction, it is a matter of choosing the enclosing interval: $[-\frac{\varepsilon}{2}; \frac{\varepsilon}{2}]$ for rounding to nearest, $[0; \varepsilon]$ for rounding toward $+\infty$, and so on.

Thanks to these improvements, the `interval` strategy can now directly prove Equation (3). This proof only takes a tenth of a second using degree-10 polynomials (default degree for `interval`). Note that the use of polynomial approximations, rather than the use of more naive variants of interval arithmetic, is critical for this proof, as can be experienced by reducing the degree. With degree 3, it takes about one second; with degree 2, it takes about one minute; and with degree 1, it does not seem to terminate.

4 New features

Since the goal of our work is to formally prove the function shown in Figure 1, we need several new features that were missing from the earlier work on the Cody-Waite algorithm [6]. First of all, since our implementation relies on hardware support for both integer and floating-point numbers, we need an interpretation of the expressions from Section 2 into the corresponding types (Section 4.1). Since the implementation also uses an array of pre-calculated values to reduce the degree of the polynomial approximation, expressions have been extended with support for array accesses (Section 4.2). Finally, as we mentioned earlier, the algorithm takes advantage of the inaccuracies and "flaws" of floating-point arithmetic to implement optimized versions of `nearbyint` and `int_of_float`. Hence, to ease the proof of this algorithm, we have added support for these optimized operations (Section 4.3).

4.1 Hardware operations

Similarly to $[\![e]\!]_{\mathsf{flt}}$, we would like to define another interpretation $[\![e]\!]_{\mathsf{prim}}$ which represents the computation of e using the hardware types provided by Coq's standard library. The `PrimFloat` module offers support for hardware floating-point numbers [10], while the `PrimInt63` module offers support for OCaml's 63-bit integers [5]. Both modules provide constants, basic operations ($+$, $-$, $\times$, $/$, etc.), comparisons ($=$, $<$, $\leq$), conversions, and some

miscellaneous functions (*e.g.*, floating-point predecessor and successor functions). They also provide axiomatized specifications for these hardware operations.

Since hardware floating-point numbers are just an instance of Flocq's generic floating-point numbers, we have derived the following variant of Theorem 1:

▶ **Theorem 2.** *Given an expression* e, $\mathrm{WB}(e) \Rightarrow [\![e]\!]_{\mathsf{prim}}$ *finite* $\wedge\ [\![e]\!]_{\mathsf{prim}} = [\![e]\!]_{\mathsf{rnd}}$.

There are two things to note about the definition of $[\![e]\!]_{\mathsf{prim}}$. First, not all operations can be performed directly on hardware types. Fused multiply-add (FMA), for example, is not yet provided by the `PrimFloat` module. Therefore, to complete the definition of $[\![e]\!]_{\mathsf{prim}}$, we emulate these missing operations using the Flocq library (*i.e.*, convert the operands to the formalized Flocq types, compute the result using Flocq's operations, and convert it back to the hardware type).

Second, we have made our integers 32-bit wide, so that we can use Coq's 63-bit hardware integers to compute $[\![e]\!]_{\mathsf{prim}}$ while maintaining our ability to export verified algorithms as C programs. For 32-bit integer expressions, $\mathrm{WB}(e)$ hence requires $[\![e]\!]_{\mathsf{rnd}}$ to remain inside the $[-2^{31};\, 2^{31} - 1]$ range.

4.2 Array accesses

The implementation of Figure 1 starts with an argument reduction that is very similar to Cody & Waite's, but based on a slightly different identity:

$$\exp(x) = \exp\left(x - k \cdot \frac{\ln 2}{64}\right) \cdot 2^{k/64} \quad \text{with} \quad k = \left\lceil x \cdot \frac{64}{\ln 2} \right\rfloor. \tag{4}$$

This makes the reduced argument much smaller, but it also means that the reconstruction is not a simple multiplication by an integer power of 2 anymore. To multiply by $2^{k/64}$ for some integer k, we first compute the Euclidean division of k by 64, in other words find k_q and k_r such that $k = k_q \cdot 64 + k_r$ and $0 \leq k_r \leq 63$. Since there are only finitely many different values of k_r, we pre-compute a correct rounding of $2^{k_r/64}$ for each value of k_r and store the results in a table `cst`. Therefore, to multiply by $2^{k/64}$, we first multiply by `cst.[`k_r`]` and then by 2^{k_q} (see Figure 1).

We have defined `cst` using the Coq standard library `PArray`, which provides persistent arrays [5]. To ease proofs, we have added a constructor `ArrayAcc` to the type of expressions to represent accesses to tables of constants. The constructor takes as argument an array a of hardware floating-point numbers and an integer expression i and is interpreted as follows:

$$[\![\texttt{ArrayAcc}\ a\ i]\!]_{\mathsf{prim}} := a.[[\![i]\!]_{\mathsf{prim}}].$$

For an access to be well-behaved, we need the index to be well-behaved and smaller than the length of the array, and all the entries of the array to be finite floating-point numbers.

4.3 Macro-operations

As we can see in Equation (4), the argument reduction also requires the `nearbyint` function. This poses a problem as the latter is not provided by the `PrimFloat` module. Since we only need to compute the exponential on inputs in the $[-746;\, 710]$ range, we can use the following trick[3] to compute the integer part:

$$\lceil f \rfloor = \circ(\circ(f + 1.5 \cdot 2^{52}) - 1.5 \cdot 2^{52}).$$

[3] If $|f| \leq 2^{51}$ then $f + 1.5 \cdot 2^{52}$ is between 2^{52} and 2^{53} with $ulp(2^{52}) = 1$, which means the result of the addition is rounded to the nearest integer.

Using the language of abstract expressions, we could simply represent this sequence of operations as `Op SUB (Op ADD` t `(BinFl 0x1.8p52)) (BinFl 0x1.8p52)`. However, that would not be very helpful in proofs because it leaves us the tedious work of showing that those operations behave as `nearbyint`. Instead, we want to treat those operations as if they were one single `nearbyint` operation. For this, we define a new constructor `FastNearbyint` in the language whose interpretation as a floating-point expression is the sequence of operations above, but whose interpretation as a rounded expression is the integer part:

$$\begin{aligned} [\![\texttt{FastNearbyint}\ e]\!]_{\mathsf{flt/prim}} &:= [\![e]\!]_{\mathsf{flt/prim}} \oplus \texttt{0x1.8p52} \ominus \texttt{0x1.8p52}, \\ [\![\texttt{FastNearbyint}\ e]\!]_{\mathsf{rnd}} &:= \lceil [\![e]\!]_{\mathsf{rnd}} \rfloor. \end{aligned}$$

Since these interpretations are no longer in one-to-one correspondence, proving Theorems 1 and 2 for these constructors required significantly more work on our part. This, however, saves the user from having to do the work themselves. Note that the macro-operation only works for inputs $|t| \leq 2^{51}$, so WB(`FastNearbyint` e) must contain a conjunct $|[\![e]\!]_{\mathsf{rnd}}| \leq 2^{51}$ for the theorems to hold.

The macro-operation we have just defined computes the integer part as a floating-point number, but the algorithm in Figure 1 also needs it as an integer. Hence, we define another constructor `FastNearbyintToInt` which extracts the mantissa[4] after adding $1.5 \cdot 2^{52}$:

$$\lceil f \rfloor_{\mathbb{Z}} = \mathrm{mantissa}(\circ(f + \texttt{0x1.8p52})) - 3 \cdot 2^{51}.$$

5 Application

We now have all the ingredients to state and prove the correctness of the algorithm shown in Figure 1. To simplify notation in this section, whenever a floating-point value `f` is finite (*i.e.*, neither $\pm\infty$ nor NaN), we denote by f the real value it represents.

We state the correctness of the algorithm as follows, with `x` the input, and with `flb` and `fub` respectively the lower and upper bounds of the output:

▶ **Theorem 3.** *If* `x` *is finite, then* `flb` $\leq \exp x \leq$ `fub`.

Proof of this theorem for large positive and negative values of `x` is straightforward, as those are the cases where the exponential either overflows or degenerates to 0. This section presents the methodology we have followed for proving the correctness theorem for $x \in [-745.13; 709.78]$.

For a given input `x`, to find an enclosure of $\exp x$, our algorithm performs only one approximation `y`, but then subtracts (resp. adds) an error term `d` to find the lower (resp. upper) bound of the enclosure. Correctness of the algorithm therefore relies on whether `d` is big enough to cancel out the inaccuracy of the approximation. For this reason, an essential step in our proof is to find some ε such that any choice of d with $|d| > \varepsilon$ makes $\circ(\circ(p_0 \cdot y) + d)$ an upper bound of $\exp(x - k \ln 2/64) - 2^{k_r/64}$ (and similarly for the lower bound). The value we have experimentally found for ε is characterized by the following lemma:

▶ **Lemma 4.** *Let* `d` *a finite floating-point number such that* $|d| \leq 2^{-52}$. *Then*

$$\left| 2^{k_r/64} + \circ(\circ(p_0 \cdot y) + d) - (\exp(x - k_q \ln 2) + d) \right| < \varepsilon \simeq 3 \cdot 2^{-58}.$$

[4] For hardware numbers, we have implemented it as `normfr_mantissa (fst (frshiftexp` f`))`.

Since the proof of this lemma is rather intricate, we will just illustrate our methodology on the piece of code that performs the argument reduction (Section 5.1). We will also show part of the proof for the reconstruction as it involves some unusual facts about floating-point arithmetic (Section 5.2).

5.1 Illustration of the methodology

Among other facts about the argument reduction, we need to prove that the computation of `ki` causes no exceptional behaviors and that it is indeed an integer part equal to k despite its convoluted code. To do so in the Coq proof, we have defined an abstract expression `ki'` whose interpretation in the hardware numbers—namely, $[\![\texttt{ki'}]\!]_{\mathsf{prim}}$—is the value stored in `ki`, and whose interpretation in the rounded real numbers—namely, $[\![\texttt{ki'}]\!]_{\mathsf{rnd}}$—is k. Then we can use Theorem 2 to transform a goal about `ki` into a goal about k.

In practice, we not only want to transform the goal, but we also want to assert some property on the transformed subexpression. Hence, we have implemented a strategy `assert_float` which takes as argument a predicate Q, looks for an expression of the form $[\![\mathsf{e}]\!]_{\mathsf{prim}}$, and applies the following corollary of Theorem 2:

$$\mathrm{WB}(\mathsf{e}) \implies Q([\![\mathsf{e}]\!]_{\mathsf{rnd}}) \implies (\forall x,\, x = [\![\mathsf{e}]\!]_{\mathsf{rnd}} \wedge Q(x) \implies G(x)) \implies G([\![\mathsf{e}]\!]_{\mathsf{prim}}).$$

The strategy also invokes `simplify_wb` to discharge as many conjuncts of WB(`e`) as possible. When using this strategy, the Coq proof usually looks as follows:

```
set (ki' := FastNearbyintToInt (Op MUL (Var 0) InvLog2_64)).
change (normfr_mantissa _ - _)
  with (evalPrim ki' [:x:]). (* Var 0 is mapped to x *)
assert_float (fun ki => -68736 <= ki <= 65536).
{ ... proof of the assertion ... }
```

We have used the `assert_float` strategy 8 times in the proof. Here is another example of its usage to state the main property about the reduced argument `t`, which contains exact operations just like in the original Cody-Waite algorithm (see Section 2.1):

```
set (t' := Op SUB (OpExact SUB (Var 1) ...) ...).
change (x - _ - _) with (evalPrim t' [:k, x:])).
assert_float (fun t => abs t <= 355 / 65536
                    /\ abs (t - (x - k * ln 2) <= 65537 * pow2 (-77)).
```

Contrary to the other uses of `assert_float` in the proof, `simplify_wb` is not able to completely discharge the subgoal WB(`t`). So we have to manually prove the remaining conjuncts, which are the proof obligations of exact operations:

$$\circ(k \cdot c_1) = k \cdot c_1 \quad \wedge \quad \circ(x - k \cdot c_1) = x - k \cdot c_1.$$

The proof of both equalities relies on bit-counting reasoning, which Gappa is specifically designed for [3, §4.3.4]. But since we do not want to introduce a dependency over Gappa in CoqInterval, we have performed this reasoning by hand.

5.2 Correctness of reconstruction

At this point in the proof, thanks to Lemma 4, we know $2^{k_r/64} + y_\ell \leq \exp x \cdot 2^{-k_q} \leq 2^{k_r/64} + y_u$, with y_ℓ and y_u some intermediate floating-point results. To complete the proof, we need to

deduce the following enclosure:

$$\mathit{flb} = \mathrm{pred}(\circ(\circ(p_0 + y_\ell) \cdot 2^{k_q})) \leq \exp x \leq \mathrm{succ}(\circ(\circ(p_0 + y_u) \cdot 2^{k_q})) = \mathit{fub}.$$

To do so, we want to factor out the multiplication by 2^{k_q}, but the possibility that the result might fall into the subnormal range makes this factorization impossible. So we have proved the following lemma:

▶ **Lemma 5.** *Let y be a* binary64 *floating-point number greater than 2^{-1021}. Then, for any integer k,* $\mathrm{pred}(\circ(y \cdot 2^k)) \leq \mathrm{pred}(y) \cdot 2^k$ *and* $\mathrm{succ}(\circ(y \cdot 2^k)) \geq \mathrm{succ}(y) \cdot 2^k$.

By transitivity, we are thus left to prove the following inequalities:

$$\mathrm{pred}(\circ(p_0 + y_\ell)) \leq 2^{k_r/64} + y_\ell \quad \wedge \quad 2^{k_r/64} + y_u \leq \mathrm{succ}(\circ(p_0 + y_u)).$$

These inequalities hold because the predecessor and successor functions are enough to compensate both the error between p_0 and $2^{k_r/64}$ and the rounding error of the final addition. Indeed, there are two cases, depending on the value of k_r:

- If $k_r = 0$, then $p_0 = 1$. So, the first inequality reduces to $\mathrm{pred}(\circ(1 + y_\ell)) \leq 1 + y_\ell$, which is a general property of the predecessor function. Proof of the upper bound is similar.
- If $k_r \neq 0$, then we have $1.01 < p_0 < 1.99$. Therefore, $1.001 < \circ(p_0 + y_\ell) < 1.999$ and hence $\mathrm{pred}(\circ(p_0 + y_\ell)) = \circ(p_0 + y_\ell) - 2^{-52}$. Since p_0 is a correct rounding of $2^{k_r/64}$, we have $|p_0 - 2^{k_r/64}| \leq 2^{-53}$. Similarly, $|\circ(p_0 + y_\ell) - (p_0 + y_\ell)| \leq 2^{-53}$. As a consequence,

$$\begin{aligned} \mathrm{pred}(\circ(p_0 + y_\ell)) &= 2^{k_r/64} + y_\ell + (p_0 - 2^{k_r/64}) + (\circ(p_0 + y_\ell) - (p_0 + y_\ell)) - 2^{-52} \\ &\leq 2^{k_r/64} + y_\ell + 2^{-53} + 2^{-53} - 2^{-52} = 2^{k_r/64} + y_\ell \end{aligned}$$

which completes the proof for the lower bound. Proof of the upper bound is similar.

6 Related works

While there had been some earlier works to formalize hardware arithmetic operators [15], formal verification of mathematical libraries really started with the impressive work by John Harrison. Among other things, he used the HOL Light system to prove the correctness of the IA-64 implementation of sin and cos for floating-point inputs smaller than 2^{63}. This implementation relies on 80-bit floating-point arithmetic and is accurate to 0.574 ulp [7]. It uses an intricate argument reduction: first a pre-reduction, followed by a 3-term Cody-Waite reduction, resulting in a double-*binary80* reduced argument. Then a degree-17 polynomial is evaluated, followed by a simple reconstruction of the result so as to take the lower part of the reduced argument into account. During both the argument reduction and the reconstruction, several floating-operations are actually exact and need to be considered as such, in order to be able to prove anything interesting about the result. Our methodology could be used to automate various parts of this proof, but the representation of the reduced argument as a non-evaluated sum of two floating-point numbers would presumably warrant adding a few more macro-operations to our expression language, *e.g.*, a FastTwoSum operator [13, §1.3].

A more recent work is the large verification using the Coq proof assistant of the power function of the CORE-MATH library by Laurence Rideau and Laurent Théry [8]. This includes the correctness of an exponential function whose implementation shares some similarities with ours, but it is a lot more subtle, since both input and output are double-*binary64* numbers. Their formalization, however, ignores the issue of exceptional behaviors

and just assumes that numbers can be arbitrarily large, as is traditionally the case in pen-and-paper proofs. Again, our methodology could help transition to a complete proof, especially since they are already making heavy use of CoqInterval.

Regarding the use of hardware floating-point numbers in the Coq proof assistant, Érik Martin-Dorel and Pierre Roux have implemented and verified a checker for semi-definite positive matrices [10]. The algorithm performs a Choleski decomposition using floating-point arithmetic on a slightly perturbed input matrix. The correctness theorem states, that if this decomposition succeeds, then a Choleski decomposition using exact arithmetic would have succeeded on the original input matrix, which guarantees that it was indeed semi-definite positive. The perturbation, and hence the correctness proof, depends on the ability to compute a bound on the rounding error of the floating-point decomposition [14]. Our approach would have been of little help for that use case, as the algorithm and the error bound highly depend on the dimension of the matrix.

Regarding the automation of proofs of bounds on rounding errors in a proof assistant, one can cite the FPTaylor tool, which can generate proofs for HOL Light [16]. Given a floating-point expression, it computes an affine form that encloses it, using elementary rounding errors as variables of the affine form. The strength of that tool is that the coefficients of the affine form are kept as symbolic expressions rather than intervals. This approach separates the concerns between the global optimizer used for computing enclosures and the formalization of affine forms for floating-point arithmetic. Indeed, enclosures of the symbolic expressions are only needed when they occur in terms of order 2 or more, as these terms cannot be represented as part of the affine form. Therefore, the verification of these enclosures can be done in a rather naive way, since they are only used for higher-order error term and thus do not have to be tight. It should be noted that FPTaylor supports both the standard model of floating-point arithmetic and a tighter model (see Section 3.2), but it can only generate proofs for the former. Moreover, the global optimizer used to compute the enclosures in FPTaylor is not the same as the one used to verify them in HOL Light, which might cause difficulties if the latter procedure is not strong enough or too slow.

A similar tool is PRECiSA, which targets the PVS proof assistant [11]. As with FPTaylor, errors are kept as symbolic expressions, and a global optimizer is used to compute their enclosures. Higher-order error terms, however, are not eliminated as computations progress, which might cause some performance issues compared to FPTaylor. PRECiSA, however, uses a tight formal model of floating-point errors, and the tool can detect exact subtractions (Sterbenz' lemma). Moreover, it supports conditional expressions, including the cases where rounding causes a different branch to be taken.

Finally, one should mention the VCFloat2 tool, which targets the Coq proof assistant [1]. As with the previous two tools, the error is kept as a symbolic expression. Before being fed to a global optimizer (namely CoqInterval), this expression is first simplified by expanding it through distributivity and discarding the sub-expressions that cancel. This expansion might cause some performance issues, due to combinatorial explosion. A user-provided threshold is used to further discard negligible terms, at the expense of a potentially worse error bound. It can also use a technique similar to Gappa to reduce the correlation between sub-expressions, and thus improve the tightness of the computed enclosures. The tool uses the standard model of floating-point errors, but the user can annotate operations that are supposed to be exact and the tool will verify that the conditions hold (Sterbenz' lemma). Moreover, the tool supports user-defined operations, which means that it can easily be extended with double-word arithmetic, as long as the user has formalized it beforehand.

7 Conclusion

In this article, we have presented a floating-point approximation of the exponential function, its mechanized proof of correctness, and the tools we have developed to ease the verification work. One peculiarity of this work is that the verified code is not just modeled using the Coq proof assistant, it can actually run in the logic of the system and therefore be used to perform proofs by computations. Indeed, the correctness theorem tells how the result of the approximation can be used as a lower/upper bound of the mathematical exponential. The specification of the code of Figure 1 and its correctness proof take about 600 lines of Coq script;[5] Lemma 5 is about 130 lines; extending the proof of Theorem 1 to support macro-operations and arrays takes about 500 lines; the tighter bounds on rounding errors take about 200 lines. This work was integrated in release 4.10.0 of CoqInterval.

7.1 Integration to CoqInterval and performances

As explained in the introduction, the CoqInterval library provides an interval extension of exponential that can use the floating-point unit of the processor to speed up proof checking [10]. Its implementation, however, is based on a truncated power series, which is effective but rather naive, compared with the implementations that can be found in mathematical libraries targeting hardware floating-point formats. We have thus plugged our verified implementation in place of the original one. Consider the following Coq script.

```
Goal forall x, 10 <= x <= 11 -> Rabs (exp x - exp x) <= 0x1p-6.
Proof. intros x Hx. interval with (i_bisect x, i_depth 30). Qed.
```

It states that, for any real number x between 10 and 11, the difference between $\exp x$ (mathematical exponential) and itself is less than 2^{-6}. From a mathematical point of view, this statement is useless, since it could be trivially proved by rewriting $\exp x - \exp x$ to zero, but it is a good way to exercise the computations performed by CoqInterval. Indeed, the way the `interval` strategy is invoked, it will not try to use anything fancier than naive interval arithmetic. As a consequence, because $\exp x$ is strongly correlated with itself, the formal proof generated by the tactic ends up considering around 6.7 million sub-intervals of the input enclosure $x \in [10; 11]$ (and as many interval evaluations of exponential).

Using the original implementation of exponential, this computationally intensive proof takes about 160 seconds to be checked by the Coq proof assistant on an Intel 13th-generation 4GHz processor. With the implementation verified in this work, the proof is checked in less than 8 seconds. Taking the average of three runs, the speedup is 20.5×. Since the argument reduction of the original implementation is more costly the further away from zero the input is, the speedup can grow even larger, up to 24×.

7.2 Real-life performances

Being able to perform about one million faithful interval evaluations of exponential per second inside the logic of Coq is impressive, but it is nowhere near the actual throughput of the floating-point unit of the processor. Indeed, disregarding any concern about the guaranteed accuracy of a mathematical library, one should expect a state-of-the-art implementation

[5] `https://gitlab.inria.fr/coqinterval/interval/-/blob/master/src/Interval/Float_full_primfloat.v`

```
let fexp x =
  if x < -0x1.74385446d71c4p9 then 0. else
  if x > 0x1.62e42fefa39efp9 then infinity else
  if x <> x then nan else
  ...
  ldexp (p0 +. p0 *. y) (ki asr 6)
```

Figure 2 Floating-point exponential in OCaml.

to take 25–50 cycles to compute two floating-point approximations of exponential[6] (and thus one interval enclosure), so about 100× faster than what we currently achieve in the logic of Coq. There are several reasons for the remaining gap. First of all, the code of our implementation is not directly run by the processor, but interpreted by a virtual machine. Second, this bytecode interpreter boxes floating-point numbers, and thus performs a large amount of memory allocations. Third, while our code only performs computations on values, the interpreter still needs to account for the possibility of open terms (*e.g.*, free variables) appearing as operands to the floating-point computations.

The first issue can be worked around by using the `native_compute` machinery of the Coq system, which compiles the code using the OCaml compiler and then executes it directly [2]. This machinery also partially avoids the second issue, since the compiler can optimize away the boxing of some intermediate floating-point results. But the third issue is still present and makes it hard to avoid pessimization in the generated code. As a consequence, this only improves proof checking by a factor 3× to 4× for the longer proofs.

To get a better feel of the actual performances of our implementation, we can instead implement the function directly in OCaml, as shown in Figure 2. This is roughly the same code as Figure 1, except that the original last three lines, which were computing an enclosure of $\exp x$, have been replaced by a single floating-point value: `ldexp (p0 +. p0 *. y) (ki asr 6)`. Accordingly, the first few lines return a single value for the exceptional cases. The code is run on about $1.5 \cdot 10^9$ inputs uniformly distributed among those that lead to a finite output. Compiling the code with OCaml 5.1.1, we get that the floating-point exponential from the GNU C Library is about 1.45× faster than our implementation.

Even if the GNU C Library has been heavily tuned, this is still a rather large gap. Part of the reason is its use of the FMA operation. This ternary operation computes $\circ(x \cdot y + z)$ at once, which halves the number of operations performed during the argument reduction and the polynomial evaluation. Modifying our code accordingly, this reduces its slowdown to 1.32×. When translating the code to C and compiling it with GCC, the slowdown is brought down to 1.24×. Obviously, using FMAs in place of multiplications and additions invalidates the correctness proof, since they do not compute the same values (notice the lack of rounding operator around the product). Fortunately, the proof can be easily adapted. Indeed, exact operations during the argument reduction are still exact when performed with an FMA, and having a more accurate polynomial evaluation only makes the proof simpler. Note that, while our framework supports reasoning about the FMA operation, it is not one of the native floating-point operations provided by the Coq system, so it cannot be used to speed up the implementation of CoqInterval. One would instead have to use larger tables, as does the GNU C Library, so as to reduce the degree of the polynomial approximation.

[6] `https://core-math.gitlabpages.inria.fr/`

7.3 Future works

First, it should be noted that, while the GNU C Library does not implement correct rounding either, it is nonetheless slightly more accurate than our implementation. In about 20% of cases, the code of Figure 2 returns a floating-point result that is off by one, while for the GNU C Library, the probability is 10^{-5}. In the context of CoqInterval, this hardly matters, since we want to compute an enclosure of the mathematical result rather than the nearest floating-point number. But for a mathematical library, people might prefer a code that is experimentally a bit more accurate to a code whose correctness has been formally verified. Most of the inaccuracy comes from the factor `p0`. There are two ways to improve it, both of which require adding a new table along `cst`. In the first approach, the new table contains the error on `p0`, which can then be reintroduced in the computation. In the second approach, the new table tells how to shift the input, such that the error on `p0` becomes negligible.

A natural extension of this work is to convert all the other mathematical functions of CoqInterval to use some state-of-the-art implementation when hardware floating-point numbers are used as interval bounds. For functions such as log and arctan, our approach should work without difficulty, as they are quite similar to exp. For trigonometric functions such as sin and cos, the situation is slightly different. First of all, they are not monotone, so considering the lower and upper bounds of the input interval separately might be counter-productive; it might be better to perform a simultaneous argument reduction on both bounds. Second, the Cody-Waite approach to argument reduction does not scale well to extremely large inputs, while some other algorithms for argument reduction take advantage of the periodicity of the trigonometric functions [12, §11.4].

References

1 Andrew Appel and Ariel Kellison. VCFloat2: Floating-point error analysis in Coq. In *13th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 14–29, London, United Kigdom, 2024. doi:10.1145/3636501.3636953.

2 Mathieu Boespflug, Maxime Dénès, and Benjamin Grégoire. Full reduction at full throttle. In *1st International Conference on Certified Programs and Proofs*, pages 362–377, Kenting, Taiwan, December 2011. doi:10.1007/978-3-642-25379-9_26.

3 Sylvie Boldo and Guillaume Melquiond. *Computer Arithmetic and Formal Proofs*. ISTE Press – Elsevier, 2017.

4 William J. Cody, Jr. and William Waite. *Software Manual for the Elementary Functions*. Prentice-Hall, Englewood Cliffs, NJ, 1980.

5 Maxime Dénès. Towards primitive data types for Coq 63-bits integers and persistent arrays. In *5th Coq Workshop*, Rennes, France, July 2013. URL: https://coq.inria.fr/files/coq5_submission_2.pdf.

6 Paul Geneau de Lamarlière, Guillaume Melquiond, and Florian Faissole. Slimmer formal proofs for mathematical libraries. In Theo Drane and Anastasia Volkova, editors, *30th IEEE International Symposium on Computer Arithmetic*, Portland, OR, USA, September 2023.

7 John Harrison. Formal verification of floating point trigonometric functions. In Warren A. Hunt and Steven D. Johnson, editors, *3rd International Conference on Formal Methods in Computer-Aided Design*, volume 1954 of *Lecture Notes in Computer Science*, pages 217–233, 2000.

8 Tom Hubrecht, Claude-Pierre Jeannerod, Paul Zimmermann, Laurence Rideau, and Laurent Théry. Towards a correctly-rounded and fast power function in binary64 arithmetic. 2024. URL: https://inria.hal.science/hal-04159652.

9 Érik Martin-Dorel and Guillaume Melquiond. Proving tight bounds on univariate expressions with elementary functions in Coq. *Journal of Automated Reasoning*, 57(3):187–217, 2016. doi:10.1007/s10817-015-9350-4.

10 Érik Martin-Dorel, Guillaume Melquiond, and Pierre Roux. Enabling floating-point arithmetic in the Coq proof assistant. *Journal of Automated Reasoning*, 67, 2023. doi:10.1007/s10817-023-09679-x.

11 Mariano Moscato, Laura Titolo, Aaron Dutle, and César A. Muñoz. Automatic estimation of verified floating-point round-off errors via static analysis. In Stefano Tonetta, Erwin Schoitsch, and Friedemann Bitsch, editors, *36th International Conference on Computer Safety, Reliability, and Security*, volume 10488 of *Lecture Notes in Computer Science*, pages 213–229, Trento, Italy, 2017. doi:10.1007/978-3-319-66266-4_14.

12 Jean-Michel Muller. *Elementary Functions, Algorithms and Implementation*. Birkhäuser, Boston, MA, 3rd edition, 2016. doi:10.1007/978-1-4899-7983-4.

13 Jean-Michel Muller, Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, Mioara Joldes, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Basel, 2nd edition, 2018. doi:10.1007/978-3-319-76526-6.

14 Pierre Roux. Formal proofs of rounding error bounds – with application to an automatic positive definiteness check. *Journal of Automated Reasoning*, 57(2):135–156, 2016. doi:10.1007/s10817-015-9339-z.

15 David M. Russinoff. *Formal Verification of Floating-Point Hardware Design*. Springer Cham, 2nd edition, 2022. doi:10.1007/978-3-030-87181-9.

16 Alexey Solovyev, Marek S. Baranowski, Ian Briggs, Charles Jacobsen, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic Taylor expansions. *ACM Transactions on Programming Languages and Systems*, 41(1), December 2018. doi:10.1145/3230733.