JAR manuscript No.
(will be inserted by the editor)

Proving Tight Bounds on Univariate Expressions with Elementary Functions in Coq

Érik Martin-Dorel · Guillaume Melquiond

The final publication is available at Springer *via*:
http://dx.doi.org/10.1007/s10817-015-9350-4

Abstract The verification of floating-point mathematical libraries requires computing numerical bounds on approximation errors. Due to the tightness of these bounds and the peculiar structure of approximation errors, such a verification is out of the reach of generic tools such as computer algebra systems. In fact, the inherent difficulty of computing such bounds often mandates a formal proof of them. In this paper, we present a tactic for the Coq proof assistant that is designed to automatically and formally prove bounds on univariate expressions. It is based on a formalization of floating-point and interval arithmetic, associated with an on-the-fly computation of Taylor expansions. All the computations are performed inside Coq's logic, in a reflexive setting. This paper also compares our tactic with various existing tools on a large set of examples.

Keywords Interval arithmetic · Formal proof · Decision procedure · Coq proof assistant · Floating-point arithmetic · Nonlinear arithmetic

1 Introduction

Libraries of mathematical functions (so called `libm`) are pieces of software that provide floating-point approximations of the most common mathematical functions, *e.g.* exp, cos. The use of such functions is so pervasive nowadays that the IEEE 754–2008 standard for floating-point arithmetic gives recommendation on their accuracy. As such, it is critical that libraries document the accuracy of the computed values, so that users of those libraries can rely on them. In fact, some library authors go as far as publishing mathematical proofs showing that the documented accuracy is correct.

This work was funded by the Verasco ANR project (ref. ANR-11-INSE-003). It was partly done while the first author was with Inria Saclay–Île-de-France, in the LRI research laboratory.

Érik Martin-Dorel
Université Toulouse 3–Paul Sabatier, Institut de Recherche en Informatique de Toulouse, UMR 5505 CNRS
IRIT, Université Paul Sabatier, 118 route de Narbonne, 31062 Toulouse Cedex 9, France

Guillaume Melquiond
Inria Saclay–Île-de-France, LRI, UMR 8623 CNRS
PCRI, bât 650, Université Paris-Sud, F-91405 Orsay Cedex, France

For instance, if we take a look at the correctness proof for the implementation of exp in the CRlibm library,[1] we can notice that it relies on the following assumption, among others: For any x such that $|x| \leq 355 \cdot 2^{-22}$, we have the following bound:

$$\left| \frac{x + 0.5 \cdot x^2 + c_3 x^3 + c_4 x^4 - \exp x + 1}{\exp x - 1} \right| \leq 2^{-62} \tag{1}$$

with $c_3 = 6004799504235417 \cdot 2^{-55}$ and $c_4 = 1501199876148417 \cdot 2^{-55}$.

Verifying such an assumption by hand is extremely tedious and error-prone. So we might first want to get a bit more insight on that property. Since the relative error it bounds is a well-behaved univariate function of x, we could first try to plot it with some computer algebra system: Maple, Mathematica, Matlab, and so on. Indeed, if the extremum points of the function graph visually satisfy the property, then the assumption is most probably valid. While not a formal proof in any way, this check would already go a long way in increasing the confidence in the library. Unfortunately, as can be seen on Figure 1, the resulting plot looks so fishy that it might have the opposite effect on the users: a decreased confidence in the library. The reason why standard tools such as Gnuplot fail to plot the relative error of this function should not come as a surprise. Indeed, their plotting procedures perform binary64[2] computations, as it is the floating-point arithmetic natively supported by most processors. Such computations have a precision of 53 bits, while the relative error we are interested in is 2^{-62}. For instance, a precision of at least 90 bits is needed for the plot to look sensible when using Sage.

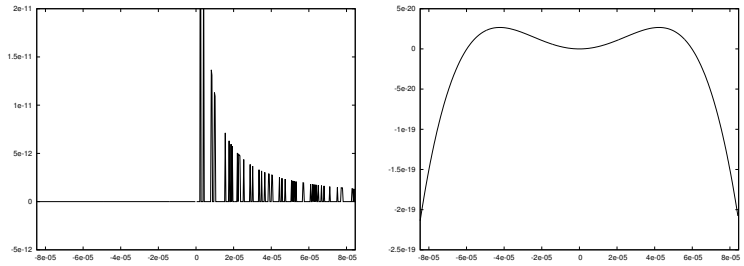

Fig. 1 On the left, the function from Equation (1), as plotted by the Gnuplot tool, which relies on binary64 floating-point numbers. On the right, its actual graph, as plotted by Sollya.[4]

At that point, we have no other choice than to turn to some other tools, if we ever want to trust the correctness proofs of mathematical libraries. The property above is quite representative of the kind of statements one encounters when proving a mathematical library. In fact, the correctness of a modern library might depend on the proof of hundreds of tight bounds on univariate functions, for table-based implementations [22]. Ideally, we should even go all the way to a formal proof, to get the highest confidence in the implementation of these libraries.

[1] `http://lipforge.ens-lyon.fr/www/crlibm/`

[2] Binary64 is the name of the IEEE 754–2008 floating-point format that was formerly known as the "double precision" format.

[4] `http://sollya.gforge.inria.fr/`

1.1 Background and Scope

Historically, the CoqInterval development was at the origin of this effort to verify bounds on univariate functions by using the Coq proof assistant. It provided a formalization of a computable floating-point library [18] and a decision procedure based on interval arithmetic was built on top of it [19]. The floating-point reference algorithms were later moved to the Flocq library [3], while the CoqInterval library focused on formalizing faster versions of the floating-point operators and elementary functions [20]. Independently developed, the CoqApprox library was built on top of CoqInterval's formalization of interval arithmetic. It was designed to compute formally-verified polynomial approximations of univariate functions which can then be fed to programs solving the Table-Maker's Dilemma [4,17].

So, at that point in time, we had, on one side, a decision procedure to verify bounds on univariate functions and, on the other side, a library able to generate accurate polynomial approximations. Since verifying bounds on a univariate polynomial might be faster than on an arbitrary function, it was then natural to try to merge both approaches in order to get a more efficient decision procedure for the Coq proof assistant. This paper gives an overview of how these different libraries are combined together and what features they offer in the context of automatically and formally verifying bounds on univariate functions. The resulting decision procedure is part of the CoqInterval formalization, which is available at

`http://coq-interval.gforge.inria.fr/`

Before going any further, let us describe which kind of expressions we intend to prove bounds on. At first, we can qualify these expressions as being univariate: only one variable can occur in them; all the other arity-0 symbols are numeric constants. This is a bit too restrictive though. Indeed, the approach we present is based on interval arithmetic, so the amount of variables does not really matter, only the number of their occurrences does. A better description would therefore be that, for tight bounds to be computed, among all the variables of the expression, only one (say, x_1) occurs several times. In the sequel, we will say that such an expression is *quasi-multivariate*. Note that this requirement could be relaxed a bit further: if a variable is the only one to appear in a sub-expression, it can be seen as appearing only once in that sub-expression. For instance, the function $f(x,y) = x + \cos(x + (y + \exp(y)))$ fits this interpretation since y only occurs in the $(y + \exp(y))$ sub-expression and x does not occur there. As a consequence, the methods presented in this paper could deal with such a function. Still this hardly covers all the multivariate expressions, hence the usage of "univariate" and "quasi-multivariate" to characterize the expressions the CoqInterval library can handle.

1.2 Related Works

There are various other systems that make it possible to prove inequalities on real-valued expressions. Their purposes are rather diverse. Some are limited to univariate expressions, while others can deal with multivariate expressions. Some are limited to polynomial expressions, while others also support some elementary functions, *e.g.* exp, tan. Finally, some generate proofs that are mechanically checked by proof assistants, while others are standalone tools.

Let us start with Sollya [7,6]. Its interface looks like it comes from a generic computer algebra system but the tool is dedicated to manipulating univariate expressions and computing guaranteed bounds on them. It supports a large range of elementary functions. It

is based on the interval arithmetic paradigm: while its results might be useless because of overestimation, they are never incorrect, by design. Its `infnorm` algorithm for computing approximation bounds is able to generate proofs written in natural language. There exists no tool that can mechanically check them.

MetiTarski is another standalone tool, but it does look like a decision procedure [1]. This is a version of the Metis resolution prover that was extended with a decision procedure solving polynomial systems. Elementary functions are supported thanks to axioms giving some polynomial lower and upper bounds on them. It is complete when it comes to polynomial expressions, but for elementary functions, it is only as strong as the polynomial approximations it uses for them. MetiTarski can generate proofs but, to our knowledge, there exists no tool that can mechanically check them yet. For instance, while PVS provides a `metit` strategy, it blindly trusts the result of MetiTarski [10].

The HOL Light proof system provides a decision procedure `REAL_SOS` for polynomial systems based on sum-of-square certificates [13]. The certificates are generated by an external non-guaranteed solver for semi-definite programming; their correctness is then verified by the HOL Light kernel. Because of round-off errors in this external solver, the certificates might fail the verification step. For this reason, the decision procedure also uses various heuristics to improve the handling of univariate expressions.

In the context of the Flyspeck project,[5] a new HOL Light procedure `verify_ineq` was designed in order to support multivariate expressions involving some elementary functions [25]. An external tool first precomputes a suitable subdivision of the input domain. Then, on each subdomain, the procedure computes an order-1 Taylor–Lagrange polynomial approximation (that is, with a quadratic remainder) in HOL Light and uses it to prove the bound on the expression. Computations are performed using interval arithmetic.

For multivariate polynomial expressions, the PVS proof system uses an approach based on Bernstein polynomials [23]. It first represents the input expression in the Bernstein basis. Then it uses a branch-and-bound procedure to compute tight enclosures of the extremum values. There is also an `interval` strategy for bounding multivariate expressions that involve elementary functions. It relies on interval arithmetic and a generic branch-and-bound algorithm [24].

Finally, another tool was designed to tackle the multivariate inequalities that appear in the Flyspeck project: NLCertify [2]. First, non-algebraic expressions are bounded by sets of quadratic forms. Then, the resulting semialgebraic system is solved using an external non-guaranteed solver for semi-definite programming. All the results are meant to be verified using the Coq proof assistant, but for now, only the certificates for expressing polynomials as sums of square are.[6]

1.3 Content

Interval arithmetic is a well-known tool to compute bounds on real-valued expressions [21]. In this paper, an interval denotes a closed subset of the real numbers which is represented by a pair of bounds $[a, b]$. (Other representations of intervals, *e.g.* midpoint-radius $m \pm r$, will not be used.) The main idea behind interval arithmetic is to extend operations on real numbers to operations on intervals. This extension should satisfy two properties. First, computing the

[5] `https://code.google.com/p/flyspeck/`

[6] The author of NLCertify is considering relying on CoqInterval to check the quadratic forms that bound elementary functions. This would be a step further in getting completely verified results with NLCertify.

result of an interval operator should only involve arithmetic operations on the bounds of the input intervals, so that computing with intervals is both effective and efficient in practice. Second, the interval operators should satisfy the *containment* property: the resulting interval should be large enough so that it contains all the possible results of the operation applied to real numbers. That way, interval arithmetic can be used to formally prove properties on real-valued expressions. Section 2 gives some preliminaries about interval arithmetic and presents the `interval` tactic for proving tight bounds on univariate and quasi-multivariate expressions automatically. It also shows how the CoqInterval library is structured.

For the proofs to be performed automatically, the Coq proof assistant has to be able to effectively compute using intervals. For that purpose, a floating-point arithmetic is formalized in Coq and proved correct with respect to real arithmetic. Then an interval arithmetic is built using floating-point numbers as bounds and proved to preserve the containment property. Section 3 shows how these arithmetics are formalized in Coq.

Thanks to containment, properties that can be deduced from an interval computation are trivially correct. Unfortunately, interval arithmetic is plagued by a major issue called the *dependency effect*. Indeed, the efficiency of interval arithmetic does not come for free: while the resulting intervals are guaranteed to contain any possible real result, the overestimation might be so coarse that it is impossible to deduce any non-trivial fact from them. The archetype of this situation is the interval $(-\infty, +\infty)$, which trivially contains all possible real values for the expression under study, but yields no further information. This issue appears as soon as a variable occurs several times in an expression evaluated by interval computations. Indeed, naive interval arithmetic does not track the dependencies between all these occurrences. Section 4 presents three approaches that we have formalized in Coq to alleviate this issue: bisection, automatic differentiation, and Taylor models.

Finally, Section 5 compares the performance of our implementation with all the tools described in Section 1.2 on a large set of examples. These examples contain various approximation problems, but also some tests taken from MetiTarski, and some well-known multivariate problems.

2 The `interval` and `interval_intro` Tactics

The CoqInterval library provides two tactics that allow one to automatically perform proofs using interval computations: `interval` and `interval_intro`. The `interval` tactic applies to goals that are inequalities over the reals. If the tactic cannot solve the goal, it fails with an error message. The `interval_intro` tactic is useful when doing some forward reasoning: when called on an expression, it computes an enclosure of it, then formally proves it using `interval`, and finally adds it to the proof context. The syntax of these two tactics is as follows:

- `interval` [*options*],
- `interval_intro` *expr* [*mode*] [*options*].

By default, `interval_intro` computes both a lower and an upper bound for *expr*. If only one of them is needed, one can tell the tactic about it by specifying `lower` or `upper` as a mode, so as to speed up the proof process. Both tactics can be configured with the following options:

- "`i_prec` *p*" changes the precision of floating-point computations (default: 30 bits),
- "`i_depth` *n*" changes the maximum depth of the bisection process (default: 15 for `interval`, 5 for `interval_intro`),

- "`i_bisect` x" asks for a bisection along variable x (Section 4.1),
- "`i_bisect_diff` x" asks for a bisection and an automatic differentiation of expressions with respect to variable x (Section 4.2),
- "`i_bisect_taylor` x d" asks for a bisection and the computation of degree-d Taylor models with respect to variable x (Section 4.3).

Obviously, the last three options are mutually exclusive, and if none of them is passed, the tactic does not use any method to reduce the dependency effect.

Below is a toy example showing the usage of the tactic. (`Rabs` denotes the absolute value in Coq; other symbols have their intuitive meaning.)

```
Goal
  forall x, 3/2 <= x <= 2 ->
  forall y, 1 <= y <= 33/32 ->
  Rabs (sqrt(1 + x / sqrt(x + y))
        - 144/1000*x - 118/100) <= 71/32768.
Proof.
  intros.
  interval with (i_prec 19, i_bisect x).
Qed.
```

Notice that the goal inequality involves two variables x and y. In fact, the tactic can cope with an arbitrary number of variables; but the methods for reducing the dependency effect can be applied to only one of those variables, here x. As for performance, the formal verification takes half a second, which is slow with respect to some other Coq tactics, but still tolerable for the user.

The tactic supports any expression composed of the following standard Coq operators: `Ropp` (unary minus), `Rabs` (absolute value), `Rinv` (multiplicative inverse), `Rsqr` (square), `sqrt`, `cos`, `sin`, `tan`, `atan`, `exp`, `ln`, `pow` (power by a nonnegative integer), `powerRZ` (power by any integer), `Rplus`, `Rminus`, `Rmult`, `Rdiv`. The constant `PI` is also supported. There are some restrictions on the domain of a few functions: `pow` and `powerRZ` are only supported if the exponent has a numeric value; inputs of `cos` and `sin` should be between -2π and 2π; inputs of `tan` should be between $-\pi/2$ and $\pi/2$. If not, the tactic might fail to prove anything interesting. These restrictions on the trigonometric functions are related to the naive implementation of their interval versions; any improvement to their code would lift them.

2.1 Preliminaries About Interval Arithmetic

Throughout this article, a boldface variable $\mathbf{x}$ denotes an interval enclosing a real variable x. Its lower and upper bounds are written $\underline{x}$ and $\overline{x}$, so that $\mathbf{x} = [\underline{x}, \overline{x}]$. A function over real numbers is denoted f and its application is $f(x)$. The image of interval $\mathbf{x}$ by f is $f(\mathbf{x})$; it is defined as

$$f(\mathbf{x}) = \{y \mid \exists x \in \mathbf{x},\ y = f(x)\}\,.$$

An interval extension of f is denoted $\mathbf{f}$. It is not uniquely defined, since it just has to satisfy the containment property:

$$\forall \mathbf{x} \subseteq \mathbb{R},\ f(\mathbf{x}) \subseteq \mathbf{f}(\mathbf{x}).$$

Interval operators usually satisfy some other properties, such as *isotonicity*,[7] but they are not required to prove the correctness of the operator itself. As a matter of fact, the isotonicity

[7] An interval function $\mathbf{f}$ is isotone if, for any pair of intervals $(\mathbf{x}, \mathbf{x'})$, we have $\mathbf{x} \subseteq \mathbf{x'} \implies \mathbf{f}(\mathbf{x}) \subseteq \mathbf{f}(\mathbf{x'})$ (see also [11, Definition 4.8.10]).

property does not occur in any of our proofs, but it explains why the bisection technique (presented later on in the paper) can reduce the dependency effect.

The main idea behind interval arithmetic, dubbed the *fundamental theorem of interval analysis*, is that the containment property is preserved by function composition, *e.g.* $\mathbf{f} \circ \mathbf{g}$ is an interval extension of $f \circ g$ [21]. Thus one only needs to build interval extensions of basic operators in order to perform interval computations.

In order to prove that the formula $\forall x_1 \in \mathbf{x_1}, \ldots, \forall x_n \in \mathbf{x_n}, f(x_1, \ldots, x_n) \in \mathbf{y}$ holds for some composite expression f, one first builds an interval extension $\mathbf{f}$ of f using basic interval extensions and composition. Then one computes $\mathbf{f}(\mathbf{x_1}, \ldots, \mathbf{x_n})$ and checks whether it is a subset of $\mathbf{y}$. If it is a subset, the original formula holds. That is the way the tactic uses interval arithmetic to automatically prove properties.

So as to support half-bounded intervals, we allow $\underline{x}$ to be $-\infty$, and $\overline{x}$ to be $+\infty$. (Note that, while we denote intervals with square brackets, in the case of infinite bounds, they are not part of the interval, that is, $[-\infty, +\infty] = \mathbb{R}$.) Infinite bounds are not a mandatory feature of interval arithmetic and one could define a usable one without them. It is helpful to have them though, as we ultimately want to reason about inequalities, which half-bounded intervals make easy to represent. For instance, in order to prove that $\forall x,\ x \leq 0 \Rightarrow f(x) \geq 0$, one can just compute the following interval inclusion: $\mathbf{f}([-\infty, 0]) \subseteq [0, +\infty]$.

Let us now consider the interval subtraction as an example. One way to define this operator **sub** so that it satisfies the containment property is as follows:

$$\forall \mathbf{u}, \mathbf{v} \subseteq \mathbb{R},\ \mathbf{sub}(\mathbf{u}, \mathbf{v}) = [\underline{u} - \overline{v}, \overline{u} - \underline{v}]. \tag{2}$$

Notice that an interval subtraction can be performed at the cost of just two bound subtractions, which makes it an efficient operation. Generally speaking, most interval operations have this kind of complexity. The implementation is rather straightforward, though one has to be careful when performing the operations on bounds, due to the potential presence of infinities. They also tend to make the proof of the containment property a bit cumbersome due to the explosion of cases.

Let us consider another example, the exponential function. Due to the monotonicity of exp, the containment property is trivially satisfied by the following interval extension:

$$\forall \mathbf{u} \subseteq \mathbb{R},\ \mathbf{exp}(\mathbf{u}) = [\exp(\underline{u}), \exp(\overline{u})].$$

It raises a question though: how to represent bounds? As can be seen on that example, they can indeed be almost any real number. We could simply use standard real numbers to represent them, but that would prevent us from using the reduction engine to perform computations on them, so this is not a suitable solution for a Coq implementation. We could restrict the bounds to computable real numbers instead, but we expect that such an implementation would not be tractable, due to performance issues.

This issue is not new and was dealt with by every arithmetician wanting to effectively compute such interval extensions. Since the only property we are interested in is the containment property, an interval result does not have to be the tightest possible one, it can be slightly enlarged. As a consequence, instead of real numbers, one can use any subset of $\mathbb{R}$ as the set of finite bounds. We only need to have some functions $\triangledown$ and $\triangle$ that, given a real number, return a bound that is smaller, respectively larger. The interval extension of exp can then be defined as

$$\forall \mathbf{u} \subseteq \mathbb{R},\ \mathbf{exp}(\mathbf{u}) = [\triangledown(\exp(\underline{u})), \triangle(\exp(\overline{u}))]. \tag{3}$$

Note that these two functions $\triangledown$ and $\triangle$ are only for exposition and proof purpose. We cannot first compute the exponential and then round it to a bound. Both operations happen at once: given a bound b, we compute a bound that is smaller (resp. larger) than $\exp(b)$.

Now, which subset of real numbers to choose? We could use rational numbers as was done in PVS [8]. But numerators and denominators of rational numbers tend to grow during computations, hence making them slower as they progress further. To prevent this growth, we could round rational numbers from time to time. But if we are to round them, we might just as well use only floating-point numbers (a subset of rational numbers) as they are especially suited for rounded operations.

2.2 Architecture of CoqInterval

Our CoqInterval library has been designed with a special focus on modularity, in order to easily switch the implementation of basic building blocks, but also to facilitate further extensions of the formalization. Its architecture is summarized in Figure 2.

The "Tactic" module provides the tactics themselves and their implementation: parsing the expressions and the bounds, creating a suitable formal proof, and causing Coq to check it automatically (see Section 2.3). It relies on several modules that are able to compute bounds of real-valued expressions. One of the approaches is based on automatic differentiation "Auto-diff" (see Section 4.2). Another approach is based on rigorous polynomial approximations using "Taylor models" (see Section 4.3); they are built using the CoqApprox library [4, 17], now part of CoqInterval.

Both approaches perform interval computations and can be parameterized by any implementation of interval arithmetic that satisfies the "IntervalOps" interface. The CoqInterval library comes with one such implementation: "FloatInterval" (see Sections 3.2 and 3.3). It represents intervals as pairs of floating-point bounds. Again, the implementation of floating-point arithmetic is not fixed and any implementation that satisfies the "FloatOps" interface can be used to obtain an interval arithmetic. This interface provides basic arithmetic operators such as $+$, $\times$, $\sqrt{\cdot}$ (see Section 3.1).

Finally, several implementations provide a floating-point arithmetic satisfying "FloatOps". All these implementations are proved correct using a reference implementation based on the Flocq library [3]. The simplest implementation, "GenericFloat", just reflects the reference implementation. The other implementation, "SpecificFloat", provides optimized versions of the operators, given some dedicated operations on integers. These dedicated operations are described by the "FloatCarrier" interface. Two radix-2 implementations of this interface are provided "StdZRadix2" and "BigIntRadix2". The differences between these two implementations are the performance and the surface of the Coq kernel they exercise. "BigIntRadix2" is the fastest one, while "StdZRadix2" uses a smaller part of the kernel.

To summarize, the tactics rely on the following modules. They use the "Auto-diff" and "Taylor models" modules to perform bound computations. The tactics instantiate these modules using an interval arithmetic provided by the "FloatInterval" module, which they instantiate using the "SpecificFloat" implementation of floating-point arithmetic. Finally, the tactics instantiate this last module using "BigIntRadix2" which provides a fast arithmetic over radix-2 integers.

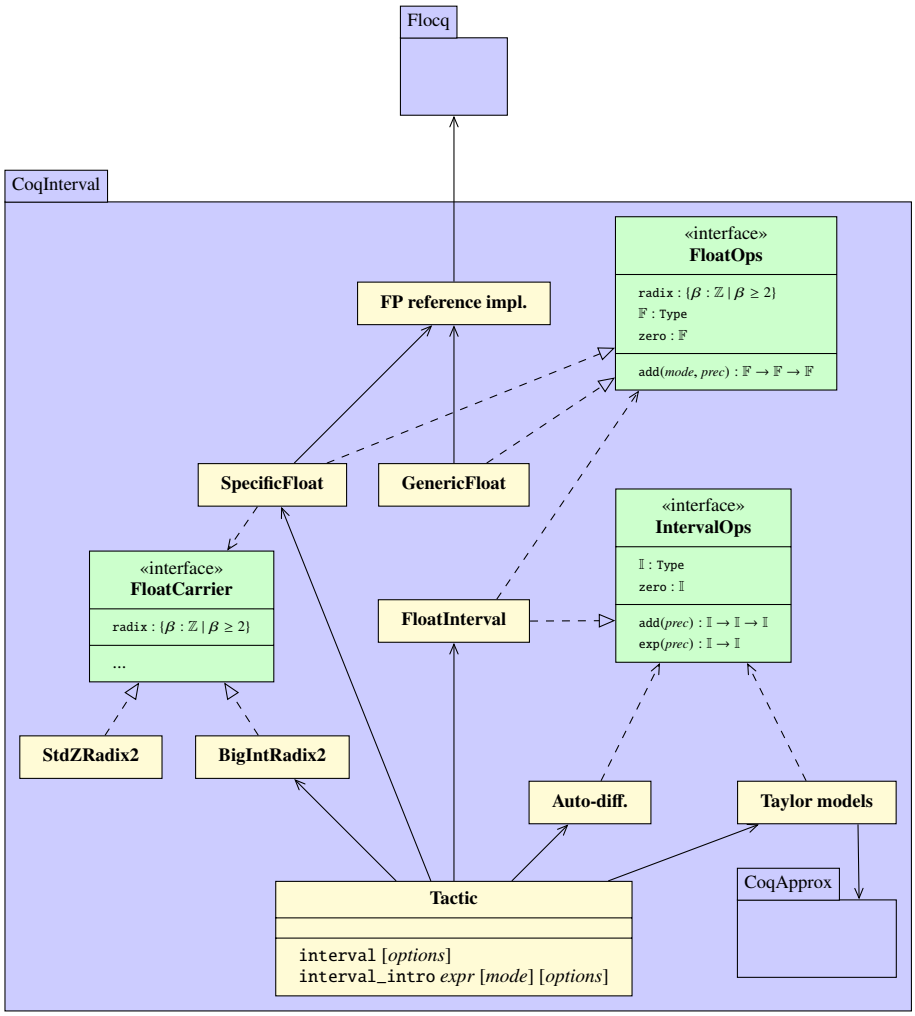

Fig. 2 Diagram (UML) that summarizes the architecture of the CoqInterval package.
The three kinds of arrows involved in the figure are:
A - - - - - - -> I if the module A is parameterized by a module that implements the interface I,
C - - - - - - -▷ I if the module C implements the interface I,
M ——————> C if the module M uses the module C.

2.3 Reification and Reflection

The `interval` tactic works by reflection. Given a goal G to prove, it constructs a Boolean expression b, such that the following implication holds:

$$b = true \Rightarrow G.$$

This general theorem has been proved once and for all; Coq just has to check that b and G are suitable to instantiate it. Once this theorem has been applied to the goal, what is left to prove is just $b = true$. Then the tactic tells Coq that this is just a consequence of the

reflexivity of equality. The formal checker of Coq is thus forced to evaluate b to verify that it is indeed equal to *true* (assuming the goal was provable this way), which concludes the proof. In the context of Coq, this approach has two advantages. First, it leverages the efficient reduction engines for the evaluation of b. Second, it produces proof terms that are tiny: just two deduction steps.

An important point about reflection is that b is not a small expression. Indeed, several libraries about integer arithmetic, a library about lists, a library about floating-point arithmetic, a library about interval arithmetic, a library about Taylor models, etc., were used to design the `interval` tactic. All the computable parts of these libraries might end up in b, depending on which options were passed to the tactic. In fact, if the definitions were to be unfolded, b would amount to several hundreds of lines of ML code. So it should not come as a surprise that the evaluation of b is the expensive stage of the verification process.

Naive interval arithmetic, automatic differentiation, and Taylor models all follow the same approach: they inductively perform computations on the sub-expressions in order to deduce the range of the whole expression. Therefore, to automate this process, we need an inductive representation of expressions. An abstract syntax tree would work, but we wanted to allow for some sharing between common sub-expressions, so as to avoid performing the same computations several times. So, instead, we represent expressions as straight-line programs with static single assignment. An expression is a sequence of statements, each statement being composed of an arithmetic operator and some integers pointing to the location of the inputs. For instance, the expression $(x+y)\cdot x+(x+y)$ is reified into the following program, with comments indicating which values would be contained in the program stack before evaluating each statement.

```
(* initial state: [y, x]                *)     Binary Add 1 0
(* current state: [x+y, y, x]           *)  :: Binary Mul 0 2
(* current state: [(x+y)*x, x+y, y, x]  *)  :: Binary Add 0 1
(* current state: [(x+y)*x+(x+y), ...]  *)  :: nil
```

We have designed a generic evaluator for such programs. The tactic instantiates it in several different ways to prove the goal. First, there is a trivial instance that associates to each operator the corresponding uninterpreted function on real numbers. Given a program, the evaluator thus produces the original real-valued expression; it is used to check that the reification process produced the correct program. Then, there are three specializations of the evaluator to do actual computations, one with single intervals, one with pairs of intervals for automatic differentiation, and one with sequences of intervals for Taylor models. For these three instances, we have proved that the arithmetic operators satisfy the respective containment properties, and by induction, that the program evaluations do, too.

3 Arithmetic Computations Inside Coq

The CoqInterval library defines an interval type with bounds represented by floating-point numbers. It provides several arithmetic operators over intervals and proves that they respect the containment property. The implementation of these operators relies on performing computations on interval bounds using floating-point operators. Section 3.1 gives an overview of how these floating-point operators for addition, multiplication, division, and square root, are defined. Section 3.2 then presents the interval type and how to define interval operators for addition, multiplication, and so on. Finally, Section 3.3 shows how to combine basic

floating-point and interval operators to compute floating-point approximations and interval extensions of elementary functions, *e.g.* exp and cos.

3.1 Floating-point Operators

Floating-point numbers are rational numbers that can be written $m \cdot \beta^e$ with m and e two integers and β a fixed integer larger than or equal to 2. The CoqInterval library does not provide any mixed-radix operation, so let us assume that β is fixed once and for all. The set of floating-point numbers is

$$\mathbb{F} = \{x \in \mathbb{R} \mid \exists m, e \in \mathbb{Z},\ x = m \cdot \beta^e\}.$$

Note that the representation of a real number as a floating-point number $m \cdot \beta^e$, if it exists, is not unique. In practice, this is neither an issue nor a source of inefficiency (floating-point rounding prevents the integers from growing large, as seen below), so we do not have to restrict m to integers that are not multiple of β.

Defining addition and multiplication on this representation is straightforward:

$$(m_1 \cdot \beta^{e_1}) + (m_2 \cdot \beta^{e_2}) = (m_1 \times \beta^{e_1 - e} + m_2 \times \beta^{e_2 - e}) \cdot \beta^e \quad \text{with } e = \min(e_1, e_2),$$
$$(m_1 \cdot \beta^{e_1}) \times (m_2 \cdot \beta^{e_2}) = (m_1 \times m_2) \cdot \beta^{e_1 + e_2}. \tag{4}$$

Obviously, such operations suffer from the same growth that caused us to discard rational numbers in the first place. So let us introduce the operators ∇ and Δ. These operators are parameterized by a precision p. This positive integer specifies how many radix-β digits the resulting mantissa can have at most. Contrarily to β, the value of p can be selected on a per-operation basis. As a matter of fact, the CoqInterval library increases the precision of intermediate computations on-the-fly when approximating elementary functions, if the current precision would lead to grossly inaccurate results; this will be detailed in Section 3.3.

To define ∇ and Δ, let us first restrict $\mathbb{F}$ to the following subset:

$$\mathbb{F}_p = \{m \cdot \beta^e \in \mathbb{F} \mid |m| < \beta^p\}$$

which will be the range of these operators. They can now be defined as

$$\nabla(x) = \max\left\{y \in \mathbb{F}_p \;\middle|\; y \le x\right\} \quad \text{and} \quad \Delta(x) = \min\left\{y \in \mathbb{F}_p \;\middle|\; y \ge x\right\}. \tag{5}$$

By definition, we have $\nabla(x) \le x \le \Delta(x)$ for any real x. So these operators are sufficient to ensure the containment property. The definitions of $\mathbb{F}$, of $\mathbb{F}_p$, and of the rounding operators come from the Flocq library, a multi-radix multi-precision multi-format formalization of floating-point arithmetic in Coq [3].

Notice that these operators return the tightest representable enclosure of x, which is much stricter than what is actually needed for automated proof purpose. Their definitions, however, make it possible to use external tools to test the Coq implementation, and vice-versa. Indeed, these definitions comply with the IEEE-754 standard for floating-point arithmetic and are thus found in most floating-point hardware and in libraries such as SoftFloat[8] or MPFR.[9] Note that the precision p is arbitrary for MPFR, while it is restricted to a few values (*e.g.*, $p = 24$ and $p = 53$) for hardware and SoftFloat. A peculiarity of the floating-point

[8] `http://www.jhauser.us/arithmetic/SoftFloat.html`

[9] `http://www.mpfr.org/`

formalization of CoqInterval is that the exponent range of the numbers in $\mathbb{F}_p$ is unbounded, implying that no arithmetic underflow nor overflow can occur.

The definitions of ∇ and Δ in Equation (5) are suitable for proofs, but they do not offer a way to actually compute the results of the operators. So they are not amenable to automated proof. So the CoqInterval library also provides some functions that, given a value of $\mathbb{F}$ compute a value of $\mathbb{F}_p$. For instance, `Fround_at_prec`$(\beta, mode, p, m \cdot \beta^e)$ returns the element of $\mathbb{F}_p$ the closest to $m \cdot \beta^e$, with the meaning of closest being controlled by *mode* (∇ or Δ). Then one can just compose such a function with + and $\times$ to get rounded values. For instance, the rounded multiplication between two floating-point values is defined in our CoqInterval library as follows.

```
Definition Fmul beta mode prec (x y : float beta) :=
  Fround_at_prec mode prec (Fmul_aux x y).
```

In this definition, `Fmul_aux` is the exact product between two floating-point numbers of type $\mathbb{F}_\beta$, as defined in Equation (4).

For division and square root, however, this approach does not work, since the intermediate results cannot be represented as values of $\mathbb{F}$. So dedicated algorithms are provided for these operators. While originally designed for the CoqInterval library, these algorithms and their proofs have now migrated to Flocq.

That said, even if the algorithms in Flocq are useful as reference implementation, they have not been implemented with a strong concern about performance. In particular, they do not take advantage of the value of β or of the regularity of $\mathbb{F}_p$ (fixed precision, no subnormal numbers). So our library also provides some optimized versions of these algorithms, which are proved equivalent to the ones in Flocq. For instance, when integers use a radix-β representation, to know whether an integer is a multiple of β^k, rather than performing a costly Euclidean division, one can just count the number of less-significant digits that are equal to zero.

In the end, our CoqInterval library proposes several implementations of the basic floating-point operators. For instance, one uses the `Z` type of integers represented as a string of bits, while another one uses the `BigZ` type of integers represented as a balanced binary tree of 31-bit native integers. All these implementations have the same interface though (see Figure 2), so the user can swap one for the other. In particular, the interval operators (presented in Section 3.2) do not care about the actual implementation of the floating-point operators.

In our setting, the floating-point formalization just has to provide an implementation of the following floating-point operations and their correctness proofs: comparison, minimum, maximum, absolute value, opposite, exact addition, exact subtraction, exact multiplication, addition, subtraction, multiplication, division, square root, multiplication by a power of β, multiplication by a power of 2, rounding, conversion from `Z`, and so on. The complete interface can be found in the `Interval_float_sig.v` file. From now on, `F` will denote an optimized implementation of this interface. So, while `Fmul` is a reference floating-point multiplication using non-optimized integer computations, `F.mul` provides an optimized implementation for a fixed radix β.

One last point about our floating-point arithmetic is that it supports a $\bot$ element that is propagated along the computations. So, the domain of all the functions is not $\mathbb{F}$ but $\overline{\mathbb{F}} = \mathbb{F} \cup \{\bot\}$, and floating-point operations are extended accordingly, so $\bot$ is an absorbing element. Note that, in general, having such an absorbing element is not that useful for formalizing most of floating-point arithmetic, *e.g.* rounding. In fact, the reference operators of Flocq do not even support it. Its point will be clearer when it comes to interval arithmetic.

3.2 Interval Operators

As explained before, in order to support half-bounded intervals, one wants to use $-\infty$ as a lower bound and $+\infty$ as an upper bound. Our floating-point arithmetic does not support such infinities, but it supports an absorbing element $\bot$, which we can use to represent infinities. Whenever a floating-point lower bound is $\bot$, the interval extends to $-\infty$, and similarly for the upper bound and $+\infty$. The set $\mathbb{I}$ of intervals with floating-point bounds is thus just $\overline{\mathbb{F}}^2$. These intervals are built with the constructor `Ibnd` in the code snippet below.

Now that we have a type and an arithmetic for the bounds, we can define interval operators. Let us consider the example of interval subtraction. Its Coq implementation is as follows, with `F.sub` being the floating-point subtraction, and `rnd_DN` and `rnd_UP` being the directions ∇ and $\triangle$.

```
Definition sub prec xi yi :=
  match xi, yi with
  | Ibnd xl xu, Ibnd yl yu =>
    Ibnd (F.sub rnd_DN prec xl yu) (F.sub rnd_UP prec xu yl)
  end.
```

Notice that, because $\bot$ is absorbing, no extra care is taken to handle infinite bounds, they just propagate along the computations. Theorem `sub_correct` then states that the implementation above satisfies the containment property, as given in Equation (2).

For interval multiplication, the implementation is not as simple as for subtraction. Indeed, the traditional way of computing it is

$$\mathbf{mul}(\mathbf{u},\mathbf{v}) = [\nabla(\min(\underline{u}\underline{v},\overline{u}\underline{v},\underline{u}\overline{v},\overline{u}\overline{v}),\triangle(\max(\underline{u}\underline{v},\overline{u}\underline{v},\underline{u}\overline{v},\overline{u}\overline{v}))].$$

It suffers from two issues. First, it performs a bit too many multiplications of bounds. Second, the resulting interval will be grossly overestimated, if one of the bound multiplication involves 0 and $\bot$. For instance, such an algorithm would return $[-\infty,+\infty]$ when computing the product between $[0,0]$ and $[-\infty,+\infty]$ while the tightest interval satisfying the containment property is $[0,0]$. So we use a variant of the algorithm that compares the input bounds to retain only the relevant products, hence returning the tightest results while being less costly on average.

Contrarily to the situation for floating-point arithmetic, division and square root on intervals are as simple as multiplication, so we do not detail their implementation. Internally, optimized implementations of multiplication and division are also available when one of the inputs is known to be a point interval.

Finally, the last operation worth mentioning is the interval extension of $x \mapsto x^k$ for k an integer. Indeed, due to the dependency effect, computing $\mathbf{x}^2$ as $\mathbf{mul}(\mathbf{x},\mathbf{x})$ would give grossly overestimated results if $\mathbf{x}$ straddles zero. More generally, computing $\mathbf{x}^{m+n}$ as $\mathbf{mul}(\mathbf{x}^m,\mathbf{x}^n)$ is a poor choice in interval arithmetic. As a consequence, passing intervals to a fast exponentiation algorithm would lead to pointless results.

So we provide a dedicated algorithm for this kind of exponentiation. It checks whether k is even or odd, looks at the signs of the bounds of $\mathbf{x}$, and then computes lower and/or upper bounds on $\underline{x}^k$ and/or $\overline{x}^k$ using a fast exponentiation algorithm. Note that, contrarily to the other operations, the result is guaranteed to be the tightest interval only when $k \in \{-1,0,1,2\}$. For the other powers, the bounds of the result might be off by a few units in the last place[10] (while still satisfying the containment property), for the sake of speed.

[10] The unit in the last place of a real number x is the gap between the two floating-point numbers enclosing x in a given format (see also [22, p. 32]).

As with floating-point arithmetic, our interval arithmetic also supports an absorbing element $\bot_I$. So the actual type of intervals is $\overline{\mathbb{I}} = \mathbb{I} \cup \{\bot_I\}$. Again, there is not much use for such an element and most implementations of interval arithmetic do without it. Its benefits show up when implementing our `interval` tactic (and in particular with the approach based on automatic differentiation), as it makes it possible to keep track of the results of partial functions, *e.g.* $[-1,1]^{-1}$ is defined as $\bot_I$ (due to 0^{-1}) rather than $[-\infty,+\infty]$.

From now on, the interval operators **add**, **sub**, **mul**, **div** will be simply written using infix notation $+, -, \times, /$, while keeping the boldface convention for their interval operands.

3.3 Elementary Functions

At this point, we have some floating-point and interval functions for the basic arithmetic operators: addition, multiplication, division, square root. Yet we want the tactic to support more mathematical functions. In particular, we need some elementary functions, *e.g.* exp and cos, since our goal is to formally verify libraries of such functions.

As before, in order to get interval extensions, the first step is to build floating-point approximations. The first notable point is that we cannot expect to compute the best approximations at a given precision for an elementary function. For instance, we can compute a floating-point number y smaller than $\exp(x)$ for x a floating-point number, but we cannot guarantee that $y = \nabla(\exp(x))$. Indeed, while there are simple algorithms [27] for computing $\nabla(\exp(x))$, proving their termination is next to impossible in Coq. Fortunately, our goal is to perform interval arithmetic, so we are fine even if the results are not always the closest possible floating-point numbers. So let us look for a good enough approximation rather than the best one.

When the precision is known beforehand, the usual way of approximating a function is the evaluation of a precomputed fixed-degree polynomial [22]. Since we want to handle arbitrarily large precisions, we instead evaluate power series which we truncate on the fly. There are two difficulties: knowing at which point to truncate the series and bounding the error due to the part that was truncated away. There are techniques to solve these two issues, as can be seen in MPFR, but the formalization effort they would require did not seem worth the performance gain. If we wanted to approximate arbitrary special functions, *e.g.* Airy or Bessel, we would have to apply such methods, but here we are dealing with elementary functions.

The main difference between special functions and elementary functions is that the latter have *argument reduction* identities. For instance, the identity $\cos(x) = 2\cos^2(x/2) - 1$ makes it possible to transform the input x into an input arbitrarily close to 0. (Note that, for $\beta = 2$, the computation of $x/2$ is trivial and does not introduce any error.) Once x is close enough to 0, the process of evaluating the power series becomes much simpler.

Consider an elementary function f that we want to approximate at point x. Let us assume that it has a converging Taylor series $f(x) = \sum (-1)^k a_k x^k$ and that the sequence $k \mapsto a_k x^k$ is positive decreasing. Then we have the following inequalities:

$$0 \leq (-1)^n \left(f(x) - \sum_{k=0}^{n-1} (-1)^k a_k x^k \right) \leq a_n x^n .$$

They solve both issues at once: they tell us that we can stop summing terms when $a_n x^n$ becomes small enough and that the truncated power series is off from the value of $f(x)$ by at most $a_n x^n$.

As for the evaluation of the power series itself, we perform it using interval arithmetic. Note that this interval computation only succeeds when evaluating floating-point functions, not interval ones. Indeed, due to the alternated signs, the dependency effect is at its worst here. Fortunately, for floating-point functions, the input interval is a point interval $[x, x]$, so the overestimation stays in check, as long as the precision of the interval computations is large enough. But for interval extensions, inputs are usually not point intervals, so this approach would not work.

We now have a way to compute an elementary function when x is close to 0, but we still have to invert the argument reduction process for other inputs. Unfortunately, for a function like cos or ln, our reconstruction of the result takes a number of arithmetic operations proportional to $\ln |x|$, and each of these operations incurs an additional overestimation. To alleviate this issue, when approximating a function with a precision p, the intermediate computations are performed at a precision p' that depends on p and $\ln |x|$.

The process is similar for functions other than cos. First, we look for an interval of $\mathbb{R}$ where the function is approximated by an alternating series. The series should converge fast enough, to ensure good performance. Then, we look for an argument reduction that brings any input into this interval. Finally, we look for a recurrence that computes efficiently and accurately the series. More details on how arguments are reduced and how precision was tweaked can be found in [20].

Once we have floating-point approximations for elementary functions, we can devise interval extensions. Since exp, ln, and arctan, are monotonic, extending them to intervals is straightforward, as can be seen in Equation (3). Note that these extensions are not guaranteed to return the tightest results, since the corresponding floating-point operations do not even have this property.

As for cos, sin, and tan, their interval extensions are quite naive. Indeed, because they are not monotonic, the implementation and the proof of these extensions have been simplified by making their results meaningful only on a small domain around zero. For instance, the interval extension of sin just returns $[-1, 1]$ if its input is not a subset of $[-2\pi, 2\pi]$ (while the floating-point implementation of sin can handle arbitrarily large inputs).

4 Reducing the Dependency Effect

The dependency effect was mentioned several times already, but we were able to avoid it up to now, since the interval extensions we had to compute (*e.g.* x^k, $\sum a_k x^k$) were under our control. But we now want to tackle whatever the user can throw at our automated procedure, so correlations are bound to appear.

First of all, let us explain what the dependency effect is. Consider the expression $x - x$. Thanks to the containment property, we know that the values of this expression are contained in $\mathbf{x} - \mathbf{x}$. Let us perform this interval computation and see what we can deduce about $x - x$. For $\mathbf{x} = [0, 1]$, we have

$$\mathbf{x} - \mathbf{x} = [0, 1] - [0, 1] = [0 - 1, 1 - 0] = [-1, 1].$$

If our intent was to prove that $x - x = 0$ when $x \in [0, 1]$, there is no way we can deduce it from the result $[-1, 1]$.

This seems like an artificial example, but this is exactly what happens in practice. Remember that our objective is to prove inequalities that appear when verifying the correctness of libraries of mathematical functions and that these inequalities have the general form

$|f(x) - \tilde{f}(x)| \leq \varepsilon$ with $\tilde{f}$ an approximation of f. Because of the dependency effect, computing $\mathbf{f}(\mathbf{x}) - \tilde{\mathbf{f}}(\mathbf{x})$ will never produce any interesting result.

Note that, while the dependency effect is specific to interval arithmetic, the theory of real arithmetic, once extended with elementary functions, is nevertheless undecidable. Indeed, while the polynomial fragment is decidable, adding periodic functions such as sin makes it possible to encode integers, thus making this class of problems impossible to solve in general. Yet there is a subclass of problems that we expect to verify by reducing the impact of the dependency effect.

This section presents the various methods we have formalized to reduce the dependency effect, from the simplest one to the most complicated. None of these methods are new and some of them have been used since the early days of interval arithmetic [21]. The most naive approach is the bisection (Section 4.1), then comes an approach based on automatic differentiation (Section 4.2), and finally the most powerful approach using Taylor models (Section 4.3).

4.1 Bisection

The idea behind bisection is simple. If an interval $\mathbf{x}$ can be decomposed into sub-intervals $\mathbf{x} = \mathbf{x}_1 \cup \ldots \cup \mathbf{x}_n$, then we have the following inclusion:

$$f(\mathbf{x}) \subseteq \mathbf{f}(\mathbf{x}_1) \cup \ldots \cup \mathbf{f}(\mathbf{x}_n).$$

This approach only works if the right-hand side is tighter than $\mathbf{f}(\mathbf{x})$ (which it fortunately is, by isotonicity). If we consider the $x-x$ example again, we can see that, by using $\mathbf{x}_1 = [0, 1/2]$ and $\mathbf{x}_2 = [1/2, 1]$, we now obtain $x - x \in [-1/2, 1/2]$ instead of $[-1, 1]$. By doubling the number of input intervals, we have reduced the overestimation of the output interval by a factor 2. This statement is a general guideline when it comes to the dependency effect.

This approach, however, does not avoid the dependency effect in all practical cases. For instance, it is impossible to find a subdivision such that one can prove $x - x = 0$ with this approach. And even if we only wanted to prove $|x - x| \leq 10^{-12}$, it would require to consider 10^{12} sub-intervals and perform as many interval computations, which is out of reach of any proof assistant.

Still, this approach has a property that makes it interesting in practice. It can easily be combined with other approaches to reduce the dependency effect. Assume that some other approach fails because of some singularity in the input interval $\mathbf{x}$. Then the bisection will be able to isolate the singularity in a tiny interval $\mathbf{x}_2$ and let the other approach deals with $\mathbf{x} \setminus \mathbf{x}_2 = \mathbf{x}_1 \cup \mathbf{x}_3$.

Given a function *chk* from intervals to Booleans, the bisection process works as follows:

```
bisect(n,x) =
  if n = 0 then false
  else if chk(x) then true
  else let m be the midpoint of x in
    bisect(n − 1,[x,m]) && bisect(n − 1,[m,x])
```

Integer n is the maximum depth of the process: the width of the sub-intervals will never diminish below 2^{-n} times the width of $\mathbf{x}$. At each step, either $chk(\mathbf{x})$ returns *true*, or $\mathbf{x}$ is split into two sub-intervals and the `bisect` function is recursively called on each sub-intervals. If $\texttt{bisect}(n, \mathbf{x})$ evaluates to *true*, then for any predicate P over real numbers, we have

$$(\forall \mathbf{u},\ chk(\mathbf{u}) = true \Rightarrow \forall u \in \mathbf{u},\ P(u)) \Rightarrow \forall x \in \mathbf{x},\ P(x). \tag{6}$$

So, when trying to prove some property $\forall x \in \mathbf{x},\ P(x)$, we first build the corresponding *chk* function, we then prove the left-hand side of Equation (6), and we finally call `bisect`, hoping that it evaluates to *true*. For instance, let us suppose that we want to prove some bound on an approximation error, that is, $P(x)$ is $|f(x) - \tilde{f}(x)| \leq \varepsilon$. We define $chk(\mathbf{u})$ as $|\mathbf{f}(\mathbf{u}) - \tilde{\mathbf{f}}(\mathbf{u}))| \subseteq [-\infty, \varepsilon]$. The left-hand side of Equation (6) thus becomes a straightforward consequence of the containment property.

As explained above, bisection cannot be expected to always succeed in canceling the dependency effect. Let us characterize when it does though. For a quasi-multivariate expression on bounded input intervals, as long as the bounds the user wants to prove are not attained for some values in the input intervals, there should always be a precision and a bisection depth such that the tactic succeeds. Indeed, the tactic only supports functions that are continuous on their definition domain, so the range of the expression is a closed interval if the input intervals are a subset of its domain. As a consequence, if the bounds to be proved are not attained, there is a gap sufficient to compensate both sub-expression dependencies and floating-point round-off errors. Note that this only tells us that the process will eventually succeed, not how fast.

The previous remark explains how bisection guarantees the semi-decidability of our approach, given enough computing power. Now let us see some of the shortcomings of our implementation. Since bisection is performed inside the logic of Coq rather than being driven by an external source, *e.g.* an oracle that would give the optimal partition, the partition that is actually built might be arbitrarily large. For instance, let us suppose that, for a given problem, the optimal partition is $[0, 2^{-n}] \cup [2^{-n}, 1]$. The bisection has to process $2n + 1$ intervals to complete the proof (n failures and $n + 1$ successes), while an oracle could have immediately provided the two optimal intervals.

4.2 Automatic Differentiation

The main issue that occurs in the archetype $x - x$ of the dependency effect is the fact that x and $-x$ have opposite variations: when one increases, the other decreases. This is the kind of correlations that naive interval arithmetic cannot track. The first idea to reduce the dependency effect is thus to track the way sub-expressions evolve when the input x changes.

Let us consider the function $f(x) = x - x$. Its derivative is $f'(x) = 1 - 1$. The interval extension $\mathbf{f'}(x)$ evaluates to $[1, 1] - [1, 1] = [0, 0]$, which is the optimal enclosure. From that enclosure, we can deduce that f is a constant function, and thus that $x - x = 0$ by performing an evaluation at an arbitrary point.

Mechanizing this reasoning is done as follows. First, let us recall the Taylor–Lagrange formula at order 0 (also known as the mean-value theorem). Assuming that f is differentiable over $\mathbf{x}$ and that $x_0 \in \mathbf{x}$, we have

$$\forall x \in \mathbf{x},\ \exists \xi \in \mathbf{x},\ f(x) = f(x_0) + (x - x_0) \times f'(\xi).$$

Weakening it using the containment property, it becomes

$$\forall x \in \mathbf{x},\ f(x) \in \mathbf{f}([x_0, x_0]) + (\mathbf{x} - [x_0, x_0]) \times \mathbf{f'}(\mathbf{x}).$$

By definition, the right-hand side is an interval extension of f. (In practice, one would choose the midpoint of $\mathbf{x}$ for x_0.)

To automate this approach, we need some way to compute $\mathbf{f'}(\mathbf{x})$. This is done using the ideas of automatic differentiation. Instead of just performing arithmetic on intervals,

we now compute on pairs of intervals. The first member of the pair is the enclosure of the expression, while the second one is the enclosure of its derivative. Here are a few examples of operations:

$$\begin{aligned}(\mathbf{u},\mathbf{u'}) + (\mathbf{v},\mathbf{v'}) &= (\mathbf{u}+\mathbf{v},\mathbf{u'}+\mathbf{v'}),\\ (\mathbf{u},\mathbf{u'}) \times (\mathbf{v},\mathbf{v'}) &= (\mathbf{u}\times\mathbf{v},\mathbf{u'}\times\mathbf{v}+\mathbf{u}\times\mathbf{v'}),\\ \exp(\mathbf{u},\mathbf{u'}) &= (\exp(\mathbf{u}),\mathbf{u'}\times\exp(\mathbf{u})).\end{aligned}$$

There is a containment property that suits these pairs of intervals. As with the containment property on single intervals, it is preserved by composition, so we get $(\mathbf{f}(\mathbf{x}),\mathbf{f'}(\mathbf{x}))$ at the end of the computation. Once we have $\mathbf{f'}(\mathbf{x})$, we compute $\mathbf{f}([x_0,x_0])$ and we apply the interval version of the Taylor–Lagrange formula. The first member of the pair could be discarded, but we use it to further refine the result of the Taylor–Lagrange formula.

This approach was the reason for introducing the absorbing element $\bot_I$. It is used as the second member of the pair $(\mathbf{u},\mathbf{u'})$ to indicate that the sub-expression is not differentiable at some points of $\mathbf{x}$ (or at least not proved to be differentiable) and thus that the Taylor–Lagrange formula cannot be used at the end. In that case, we simply use the first member of the pair. For instance, expressions involving square roots or absolute values have no meaningful derivatives at point 0.

Automatic differentiation nicely integrates with interval arithmetic, but there is still some dependency effect when computing $(\mathbf{x}-[x_0,x_0])\times\mathbf{f'}(\mathbf{x})$. In some cases, we can obtain tight enclosures by relying on some monotonicity argument. Indeed, if the interval $\mathbf{f'}(\mathbf{x})$ happens to be of constant sign, then we can prove that f is monotonic on $\mathbf{x}$. As a consequence, we can just compute $\mathbf{f}([\underline{x},\underline{x}])$ and $\mathbf{f}([\overline{x},\overline{x}])$, take their convex hull, and thus obtain a superset of $f(\mathbf{x})$. Since the input intervals are point intervals, the dependency effect is minimal and the enclosure is tight. When combining automatic differentiation with bisection, the latter will split into sub-intervals at worst until $\mathbf{f'}$ has constant sign on them.

Finally, it should be noted that automatic differentiation also handles unbounded intervals. For instance, evaluating the expression $\exp x-(1+x)$ for $x\in[0,+\infty]$ gives the pair $([-\infty,+\infty],[0,+\infty])$. The first component is useless, but the second one tells us that the function is increasing. So an evaluation at $x=0$ suffices to prove that the expression is nonnegative for $x\in[0,+\infty]$. This ability to handle monotonic functions and unbounded intervals is specific to this approach; it will be lost with Taylor models.

A limitation of automatic differentiation (and of Taylor models as well) is that it requires the expression to be differentiable. If it is not, it does not perform any better than naive interval arithmetic. Yet this approach could be made to work for any Lipschitz-continuous function. In fact, even that would be too restrictive, since a function such as square root at zero could be handled. The idea is that derivatives do not really matter, only slopes do. The Taylor–Lagrange formula could then be replaced by

$$\forall x\in\mathbf{x},\ f(x)\in\mathbf{f}([x_0,x_0])+(\mathbf{x}-[x_0,x_0])\times\mathbf{hull}\left\{\frac{f(u)-f(v)}{u-v}\;\middle|\;u\neq v\in\mathbf{x}\right\}.$$

Interestingly, most of the formulas used for automatic differentiation would work in this new setting, since the set of derivatives is equal to the closure of the set of slopes for C^1 functions.

There is another issue with our implementation of automatic differentiation. For each sub-expression, its range and the range of its derivative are computed. But the knowledge learned about the derivative is not used to further refine the range of the sub-expression. Only the final expression benefits from the Taylor–Lagrange formula. As a consequence,

the intermediate computations suffer from the dependency effect, which in turn inflates the overestimation on the derivatives, thus causing some monotonic functions to not be detected as such. Obviously, applying the Taylor–Lagrange formula at each step would induce an overhead, so there is a trade-off to explore between both implementations.

4.3 Taylor Models

4.3.1 Preliminaries About Taylor models

Borrowing the term chosen in [15,16], we will call *Taylor model* a pair $(P,\mathbf{\Delta})$, where P is a polynomial (in the Taylor polynomial basis[11]) and $\mathbf{\Delta}$ an interval error bound. The intuition of such a data type is that a Taylor model $(P,\mathbf{\Delta})$ represents a whole set of functions, namely the functions $f : D \to \mathbb{R}$ (on a given domain $D \subseteq \mathbb{R}^n$) such that

$$\forall x \in D,\ f(x) - P(x) \in \mathbf{\Delta}, \tag{7}$$

or more concisely $f \in [\![(P,\mathbf{\Delta})]\!]_D$.

To be more specific, we rely on the same setting as Mioara Joldeş's thesis [14, Chap. 2]: we especially focus on Taylor models in the univariate case, but we do not enforce that the polynomials P are actual Taylor expansions of the considered functions. Furthermore, we specifically deal with Taylor models $(\mathbf{P},\mathbf{\Delta})$ where the polynomial $\mathbf{P}$ has tight interval coefficients.

This latter choice simplifies the implementation of the Taylor model algorithms within an approximate arithmetic, as the rounding errors are directly handled by the interval computations. So this specific kind of Taylor models is used in all the intermediate steps of our formalized algorithms. But if need be, it is very easy to turn the "interval Taylor model" $(\mathbf{P},\mathbf{\Delta})$ so obtained into a "standard Taylor model" $(P,\mathbf{\Delta'})$ satisfying Equation (7). One just need to pick some P_i inside the $\mathbf{P}_i$ and accumulating the errors into $\mathbf{\Delta'}$. Even if this algorithm is not required for implementing the approach presented in this paper, it has been formally verified within the CoqApprox library.

4.3.2 Main Features of the CoqApprox Formalization

The CoqApprox library gathers certified algorithms for univariate Taylor models, which can be executed inside the logic of Coq [4,17]. The main data type involved in the formalization is a record that combines a polynomial and an interval; it is parameterized by a type for representing polynomials and another type for intervals. We say that $(\mathbf{P},\mathbf{\Delta})$ is a Taylor model for a given function $f : \mathbb{R} \to \mathbb{R}$ over a given interval $\mathbf{I}$ around $\mathbf{x_0}$ if it satisfies the following predicate (cf. [17, Definition 1]):

$$\begin{aligned} f \in [\![(\mathbf{P},\mathbf{\Delta})]\!]_{\mathbf{I}}^{\mathbf{x_0}} \iff^{\text{def}} \mathbf{x_0} \subseteq \mathbf{I} \ \wedge\ 0 \in \mathbf{\Delta} \\ \wedge \quad \forall x_0 \in \mathbf{x_0},\ \exists Q \in \mathbb{R}[X],\ \begin{cases} \operatorname{size} Q = \operatorname{size} \mathbf{P}, \\ \forall i < \operatorname{size} \mathbf{P}, \quad Q_i \in \mathbf{P}_i, \\ \forall x \in \mathbf{I}, \quad f(x) - \displaystyle\sum_{i<\operatorname{size} Q} Q_i \cdot (x - x_0)^i \in \mathbf{\Delta}. \end{cases} \end{aligned} \tag{8}$$

[11] *i.e.* in the univariate case, P is considered as $P(x) = \sum_{i=0}^{n} P_i \cdot (x - x_0)^i$ for a given expansion point x_0.

Before elaborating on the algorithms that rely on this predicate, its definition deserves some detailed explanation.

First, the "expansion point" $\mathbf{x_0}$ involved in (8) is actually an interval. This might seem surprising, but this specific choice will be a key requirement for the composition algorithm, due to the presence of polynomials with interval coefficients. Besides, it can be useful if we ever want to compute a Taylor model around an irrational expansion point.

Second, as far as univariate polynomials are concerned, our formalization deals with their "size" rather than their degree. Indeed, we do not enforce that the "leading term" of the polynomial part of a Taylor model $(\mathbf{P}, \mathbf{\Delta})$ has a nonzero coefficient. So we need not prove some additional "side-conditions" related to the degree, which eases the formalization.

Regarding the three conditions of the predicate (8) itself, the first two simply assert that all the expansion points $x_0 \in \mathbf{x_0}$ belong in the input interval under study, and that the polynomial part of the Taylor model also belongs to the "set of functions" that the Taylor model represents. The statement of the third condition is just a direct translation of the following condition:

> for any expansion point $x_0 \in \mathbf{x_0}$,
>
> there exists some polynomial Q in the Taylor basis around x_0, such that
>
> Q is "contained" in the interval polynomial $\mathbf{P}$ and $\forall x \in \mathbf{I}, f(x) - Q(x) \in \mathbf{\Delta}$.

Now let us give an overview of the algorithms on Taylor models that have been formalized in the CoqApprox library.

First, we have some Coq functions to compute Taylor models for *basic functions* such as square root, exponential, or sine. For instance, the algorithm for exponential is a function `TM_exp` that takes four arguments: the working precision *prec* of floating-point computations, two intervals $\mathbf{x_0}$ and $\mathbf{I}$, and the order n of the desired Taylor model. It returns a Taylor model $\texttt{TM\_exp}(prec, \mathbf{x_0}, \mathbf{I}, n)$ centered at $\mathbf{x_0}$ that approximates exp over the interval $\mathbf{I}$.

Second, the CoqApprox library makes it possible to "combine" such Taylor models according to some *composite functions*. Each operation $+$, $-$, $\times$, $\div$, and $\circ$ (composition), corresponds to a dedicated algorithm [14]. Note that for division, we currently handle a function $\frac{f}{g}$ as $f \times (\mathrm{inv} \circ g)$ (as in [14]) but a more efficient algorithm, based on Newton's method for example, could be formalized.

All these algorithms are proved correct with respect to the predicate (8). For instance, the correctness lemma for `TM_exp` is as follows:

$$\texttt{TM\_exp\_correct} : \forall prec, \mathbf{x_0}, \mathbf{I}, n, \quad \mathbf{x_0} \subseteq \mathbf{I} \wedge \mathbf{x_0} \neq \emptyset \implies \exp \in [\![\texttt{TM\_exp}(prec, \mathbf{x_0}, \mathbf{I}, n)]\!]_{\mathbf{I}}^{\mathbf{x_0}},$$

and the correctness lemma for the addition of Taylor models is:

$$\texttt{TM\_add\_correct} : \forall prec, \mathbf{x_0}, \mathbf{I}, \mathrm{TM}_f, \mathrm{TM}_g, f, g, \quad \mathrm{size}(\mathrm{TM}_f) = \mathrm{size}(\mathrm{TM}_g) \implies f \in [\![\mathrm{TM}_f]\!]_{\mathbf{I}}^{\mathbf{x_0}} \wedge g \in [\![\mathrm{TM}_g]\!]_{\mathbf{I}}^{\mathbf{x_0}} \implies f + g \in [\![\texttt{TM\_add}(prec, \mathrm{TM}_f, \mathrm{TM}_g)]\!]_{\mathbf{I}}^{\mathbf{x_0}}.$$

The hypothesis $\mathrm{size}(\mathrm{TM}_f) = \mathrm{size}(\mathrm{TM}_g)$ makes it possible to simplify the implementation, but it might seem overly restrictive. In practice though, when computing a univariate Taylor model, all the intermediate polynomials have the same size. But this implementation choice could be relaxed, by considering that the size of the polynomials is a kind of "precision"

parameter, which could be adapted during computation to increase the performance of the Taylor model algorithms.

Regarding the implementation of basic functions in CoqApprox ($\sqrt{\cdot}$, exp, sin, etc.), we aimed at factoring the algorithmic chain as much as possible, in order to increase the extensibility of the library. Roughly speaking, for computing a Taylor model $(\mathbf{P}, \mathbf{\Delta})$ for some function f, we only have to implement two algorithms: an interval extension $\mathbf{f}$ of f (see Section 3.3) and a formula relating the iterated derivatives of f (*e.g.* its differential equation). Typically, this formula is expressed as a recurrence relation F such that, given $t_0 := f(x)$ and $t_k := F(t_{k-1}, k)$ for each integer $k \geq 1$, we get $t_k = f^{(k)}(x)/k!$ for all $k \in \mathbb{N}$.

The Taylor model is then computed as follows. In order to compute $\mathbf{P}$, the recurrence that defines the Taylor coefficients $(t_k)_{k\in\mathbb{N}}$ is "unrolled" by a dedicated function. And this is performed using interval arithmetic (hence the boldface in what follows). To be more specific, let us resume the example above of order-1 recurrence defined by F. Suppose we want to compute interval enclosures of the Taylor coefficients around $\mathbf{x}$ up to order n. To this aim, we pose $\mathbf{t_0} := \mathbf{f}(\mathbf{x})$, then we rely on our higher-order function `trec1` dedicated to order-1 recurrences: to sum up, its type is "$\texttt{trec1} : (\mathbb{I} \to \mathbb{N} \to \mathbb{I}) \to \mathbb{I} \to \mathbb{N} \to \mathbb{I}[X]$" and it suffices to evaluate $\texttt{trec1}(F, \mathbf{t_0}, n)$ to obtain the polynomial $\sum_{k=0}^{n} \mathbf{t_k} X^k$. Finally, in order to compute $\mathbf{\Delta}$, we either use the Taylor–Lagrange inequality directly,[12] or alternatively use the algorithm called Zumkeller's technique [17, Algorithm 1]. Both algorithms are implemented in the form of Coq functions. It can be noted that the latter algorithm leads to sharp bounds, as soon as it detects that the function $x \mapsto f^{(n+1)}(x)$ has a constant sign over the interval $\mathbf{I}$, which is usually the case for the basic functions we are interested in. The formal verification of this algorithm benefited from the pen-and-paper proof of Proposition 2.2.1 in Mioara Joldeş' thesis [14], which relies on Lemma 5.12 in Roland Zumkeller's thesis [28].

4.3.3 Integration of CoqApprox into CoqInterval

Regarding the integration of CoqApprox into CoqInterval, Figure 3 provides a refined view of the architecture that was presented in Figure 2. It is still simplified though, since the CoqApprox package internally relies on several other modules. For more details, see [4].

CoqApprox is integrated into CoqInterval *via* the "UnivariateApprox" interface for approximating univariate expressions. This interface is a parameterized signature that depends on an implementation of intervals (with respect to "IntervalOps", see Section 2.2). It declares an abstract data type `T` as well as a ternary relation $\mathcal{A}(\mathbf{I}, t, f)$ that indicates if an object t of type `T` is an approximation of function f over the interval $\mathbf{I}$. It also declares abstract functions to build approximations, starting from constants or identity, and combining existing approximations by using an operation (addition, subtraction, multiplication, or division), an elementary function ($\sqrt{\cdot}$, exp, sin, etc.), or the absolute value. For example, the correctness claim for $\texttt{exp} : (\texttt{precision} \times \mathbb{N}) \to \mathbb{I} \to \texttt{T} \to \texttt{T}$ is

$$\texttt{exp\_correct} : \forall (u : \texttt{precision} \times \mathbb{N})\ (\mathbf{I} : \mathbb{I})\ (t : \texttt{T})\ (f : \mathbb{R} \to \mathbb{R}),\\ \mathcal{A}(\mathbf{I}, t, f) \implies \mathcal{A}(\mathbf{I}, \texttt{exp}(u, \mathbf{I}, t), \exp \circ f). \quad (9)$$

[12] Namely, we use this simple algorithm when computing a Taylor model for identity or constant functions, as the estimation of the Taylor–Lagrange remainder is already sharp in this case.

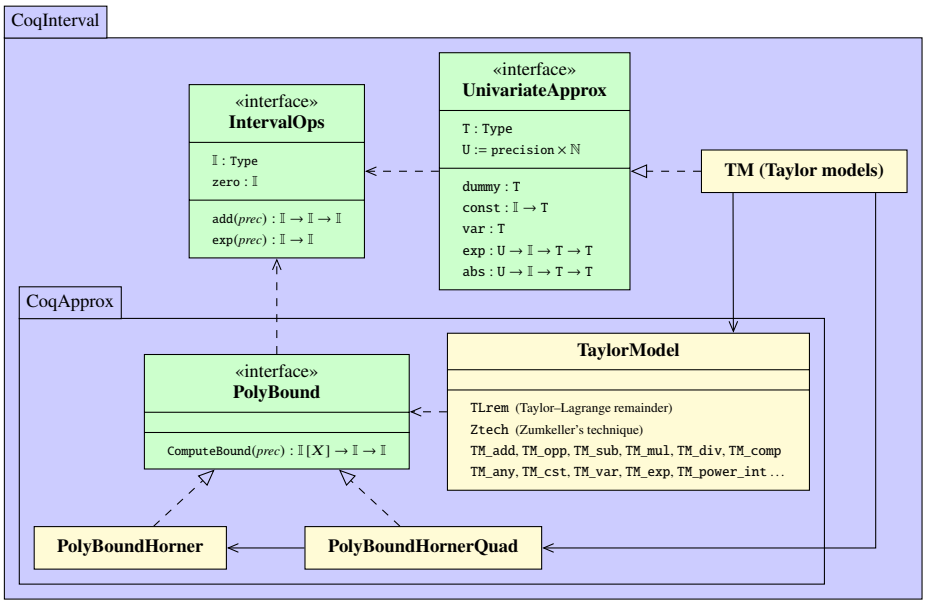

Fig. 3 Diagram (UML) that summarizes the architecture of the CoqApprox package w.r.t. CoqInterval.
The three kinds of arrows involved in the figure are:
A - - - - - - -> I if the module A is parameterized by a module that implements the interface I,
C - - - - - - ▷ I if the module C implements the interface I,
M ——————> C if the module M uses the module C.
For readability, we omit some arrows of the first kind when they simply result from transitivity, *e.g.* the TaylorModel module depends not only on PolyBound, but also on IntervalOps.

The "UnivariateApprox" interface also declares a function `eval` to compute the range of an expression from its approximation, with the following correctness claim:

$$\texttt{eval\_correct} : \forall (u : \texttt{precision} \times \mathbb{N})\ (\mathbf{I} : \mathbb{I})\ (t : \texttt{T})\ (f : \mathbb{R} \to \mathbb{R}),$$
$$\mathcal{A}(\mathbf{I}, t, f) \implies \forall (\mathbf{J} : \mathbb{I})\ (x \in \mathbf{J}),\ f(x) \in \texttt{eval}(u, t, \mathbf{I}, \mathbf{J}). \quad (10)$$

The `eval` function takes two interval arguments $\mathbf{I}$ and $\mathbf{J}$ so that one can evaluate an approximation that is valid on $\mathbf{I}$ over multiple subsets $\mathbf{J} \subseteq \mathbf{I}$. (Note that this constraint does not appear in the type of `eval_correct` on purpose. The `eval` function will return $\bot_I$ in the case where $\mathbf{J} \not\supseteq \mathbf{I}$, so that (10) will also hold in this case.) This additional flexibility, however, is not currently used, since our `interval`/`interval_intro` tactics just call the `eval` function with $\mathbf{J} := \mathbf{I}$.

Finally, an implementation of this interface "UnivariateApprox" is provided in the form of a parameterized module "TM", which takes as input an implementation of "IntervalOps", and which uses the CoqApprox algorithms. Specifically, it relies on the "TaylorModel" module of CoqApprox, which in turn takes as input an implementation of the "IntervalOps" and "PolyBound" interfaces. The function `ComputeBound` specified in "PolyBound" computes an overestimation of the range of a polynomial over an interval. It is used for truncating polynomials (in order to keep the order of Taylor models low when multiplying them) as well as for computing Taylor models for expressions involving the composition of elementary functions.

Regarding the implementation of the abstract data type `T` in the "TM" module, we do not directly use the type of Taylor models for efficiency reasons. Instead, we add three extra constructors to handle some specific families of functions:

`Inductive` `T := Dummy | Const` **`of`** $\mathbb{I}$ `| Var | Tm` **`of`** $\left(\mathbb{I}[X] \times \mathbb{I}\right)$.

The constructor `Dummy` represents any function f. As such, it is a capturing element of the abstract data type `T`. `Const` represents a constant function with a value contained in the interval argument, while `Var` represents the identity function. Both constructors `Const` and `Var` make it possible to compute the Taylor models of some common expressions without going through a full-blown composition of Taylor models. For instance, the expression $\exp(x)$ is handled as $\exp \circ \mathrm{id}$, as shown by property (9), yet it does not need an actual composition. Also, it may be noted that the interval argument $\mathbf{x_0}$ involved in the CoqApprox algorithms does not occur in our "UnivariateApprox" interface. Indeed, the instantiation of this argument is handled transparently to the user: we just take the point interval built upon the midpoint of the interval $\mathbf{I}$.

This does not appear on Figure 3 but the "UnivariateApprox" interface is generic with respect to the way polynomial approximations are computed. So, in the future, it will be possible to swap our Taylor model implementation with another implementation, such as Chebyshev models [14, Chap. 4].

4.3.4 Shortcomings

One of the shortcomings of our implementation of automatic differentiation is that, when computing the range of a sub-expression, we do not use the range of its derivative to refine it. Taylor models suffer less from this specific issue, since the range of a sub-expression only matters when applying an elementary function to a Taylor model. Yet we still need to be able to compute the range of an expression from its Taylor model, if only for bounding the final one when concluding the proof. Unfortunately, evaluating the polynomial part with Horner's rule tends to greatly overestimate the range of the function represented by a Taylor model. Again, the culprit is the dependency effect. For instance, consider the polynomial x^2. Its interval evaluation will be performed as $(\mathbf{1} \times \mathbf{x} + \mathbf{0}) \times \mathbf{x} + \mathbf{0}$, which amounts to computing $\mathbf{x} \times \mathbf{x}$. Let us see what happens for the input interval $\mathbf{x} = [-1, 1]$. (Due to the way Taylor models are formalized, the polynomial is always evaluated on an interval centered at 0.) We get $[-1, 1]$ while the actual range of the polynomial is just $[0, 1]$.

To avoid this issue, we have implemented a slightly better interval evaluation of polynomials. The idea is to rewrite them to reduce the dependency effect [5]. We did not formalize the whole method though; we only applied it to the quadratic part of the polynomials:

$$a_0 + x \cdot (a_1 + x \cdot (a_2 + x \cdot (\ldots))) = a_0 - \frac{a_1^2}{4a_2} + a_2 \cdot \left(x + \frac{a_1}{2a_2}\right)^2 + x^3 \cdot (\ldots).$$

This algorithm has been formalized in the "PolyBoundHornerQuad" module that appears in Figure 3, while "PolyBoundHorner", another implementation of the "PolyBound" interface, corresponds to the Horner evaluation algorithm that was initially formalized in CoqApprox.

If we assume that the higher-degree part of polynomials induces only negligible correlations, this rewriting gives a tight enclosure while requiring a limited amount of extra computation with respect to Horner's rule (namely, the implementation of this polynomial evaluation only requires seven extra interval operations per evaluation).

On the benchmarks of Section 5, this algorithm leads to speed-ups for proving the harrison97, MT8, and MT9 problems, making the tactic up to twice faster. The impact on the cos_cos problems is also noticeable; they are solved about 50% faster. On the other problems, there is still an overall gain, but it is less noticeable. There is an exception though: using the naive Horner scheme, the rel_err_geodesic problem would be solved about 20% faster than shown in Table 1.

5 Benchmarks

We have compared our tactic to the tools described in Section 1.2: Sollya [7], MetiTarski [1] (through PVS' `metit` strategy), NLCertify [2] (both with and without Coq verification), the `verify_ineq` [25] and `REAL_SOS` [13] procedures for HOL Light, and the `bernstein` [23] and `interval` [8] strategies for PVS. To perform the comparison, we have selected several problems and tested whether they successfully verified them, and if so, how long it took to prove them.

5.1 Selection of Some Reference Problems

Our selection of problems includes a few approximation problems taken from the literature and/or from actual implementation:

- CRlibm exponential: $|(x+0.5\cdot x^2+6004799504235417\cdot 2^{-55}\cdot x^3+1501199876148417\cdot 2^{-55}\cdot x^4-\exp x+1)/(\exp x-1)|\leq 2^{-62}$ when $2^{-20}\leq|x|\leq 355\cdot 2^{-22}$;
- square root [19]: $|\sqrt{x}-(((((122/7397\cdot x-1733/13547)\cdot x+529/1274)\cdot x-767/999)\cdot x+407/334)\cdot x+227/925)|\leq 5\cdot 2^{-16}$ when $x\in[0.5,2]$;
- arctangent, with a tighter bound w.r.t. [8, p. 235]: $|\arctan x-(x-11184811/33554432\cdot x^3-13421773/67108864\cdot x^5))|\leq 5\cdot 2^{-28}$ when $|x|\leq 1/30$;
- Earth's radius of curvature [9,19]: $|(r(\phi)-p((715/512)^2-\phi^2))/r(\phi)|\leq 23\cdot 2^{-24}$ when $\phi\in[0,715/512]$, with $r(\phi)=6378137/\sqrt{1+(1-10^9/298257223563)^2\cdot\tan^2\phi}$ and $p(t)=4439091/4+t\cdot(9023647/4+t\cdot(13868737/64+x\cdot(13233647/2048+x\cdot(-1898597/16384+x\cdot(-6661427/131072)))))$;
- Tang's exponential [12,26]: $|(\exp x-1)-(x+8388676\cdot 2^{-24}\cdot x^2+11184876\cdot 2^{-26}\cdot x^3)|\leq(23/27)\cdot 2^{-33}$ when $|x|\leq 10831\cdot 10^{-6}$.

We have also crafted some increasingly-difficult approximation problems using polynomial approximations with binary32 coefficients of $f(x)=\cos(1.5\cdot\cos x)$ for $x\in[-1,1/2]$. The goal is to prove $|(p_n(x)-f(x))/f(x)|\leq C_n$ with n the degree of the approximation between 2 and 8. The bounds are $C_2=57\cdot 2^{-10}$, $C_3=51\cdot 2^{-11}$, $C_4=51\cdot 2^{-14}$, $C_5=3\cdot 2^{-12}$, $C_6=17\cdot 2^{-16}$, $C_7=25\cdot 2^{-19}$, and $C_8=5\cdot 2^{-20}$. The polynomial coefficients are easily obtained by running the following Sollya command, so they will not be reproduced here: `fpminimax(`f`, `n`, [|SG...|], [-1;1/2], relative)`.

We have also selected a few polynomial problems that have been recurrently used for testing tools [23,25]. While multivariate (up to 7 variables), they fit into our definition of quasi-multivariate expressions and thus were a sensible choice:

- 3-variable reaction diffusion: $-36.7126907\leq -x_1+2\cdot x_2-x_3-0.835634534\cdot x_2\cdot(1+x_2)$ when $x_1,x_2,x_3\in[-5,5]$;
- adaptive Lotka-Volterra system: $-20.801\leq x_1\cdot x_2^2+x_1\cdot x_3^2+x_1\cdot x_4^2-1.1\cdot x_1+1$ when $x_1,x_2,x_3,x_4\in[-2,2]$;

- Butcher's problem: $-1.44 \leq x_6 \cdot x_2^2 + x_5 \cdot x_3^2 - x_1 \cdot x_4^2 + x_4^3 + x_4^2 - x_1/3 + 4/3 \cdot x_4$ when $x_1 \in [-1,0]$, $x_2 \in [-0.1,0.9]$, $x_3 \in [-0.1,0.5]$, $x_4 \in [-1,-0.1]$, $x_5 \in [-0.1,-0.05]$, $x_6 \in [-0.1,-0.03]$;
- magnetism: $-0.25001 \leq x_1^2 + 2 \cdot x_2^2 + 2 \cdot x_3^2 + 2 \cdot x_4^2 + 2 \cdot x_5^2 + 2 \cdot x_6^2 + 2 \cdot x_7^2 - x_1$ when $x_1, \ldots, x_7 \in [-1,1]$.

Finally, we have selected some univariate problems that are not approximation problems. They were originally designed for MetiTarski [1]. Note that some problem statements had to be slightly modified so as to accommodate as many systems as possible. First, the bounds of all the inputs have to be finite and closed. So bounds that were originally infinite were replaced by ± 10, while open bounds were modified by $\varepsilon = 2^{-10}$ so that they could be closed. Second, in order for numerical solvers to succeed, there must be a gap large enough between both sides of an inequality. So, whenever both sides could be equal for some inputs, ε was added to one of them to make them disjoint. For instance, the original version of the first problem is $2\,|x|\,/(2+x) \leq |\ln(1+x)|$ when $x > -1$, so it exhibits all of these issues: its input x has an open lower bound, it has an infinite upper bound, and both sides of the inequality are equal when $x = 0$.

- MT1: $2\,|x|\,/(2+x) \leq |\ln(1+x)| + \varepsilon$ when $x \in [-1+\varepsilon, 10]$;
- MT2: $|\ln(1+x)| \leq -\ln(1-|x|) + \varepsilon$ when $x \in [-1+\varepsilon, 1-\varepsilon]$;
- MT3: $|x|\,/(1+|x|) \leq |\ln(1+x)| + \varepsilon$ when $x \in [-1+\varepsilon, 1]$;
- MT4: $|\ln(1+x)| \leq |x|\,(1+|x|)/\,|1+x| + \varepsilon$ when $x \in [-1+\varepsilon, 1]$;
- MT5: $|x|\,/4 < |\exp x - 1|$ when $x \in [-1,1] \setminus (-\varepsilon, \varepsilon)$;
- MT6: $|\exp x - 1| < 7\,|x|\,/4$ when $x \in [-1,1] \setminus (-\varepsilon, \varepsilon)$;
- MT7: $|\exp x - 1| \leq \exp|x| - 1$ when $x \in [-10, -\varepsilon]$;
- MT8: $|\exp x - (1+x)| \leq |\exp|x| - (1+|x|)|$ when $x \in [-10, -\varepsilon]$;
- MT9: $|\exp x - (1+x/2)^2| \leq |\exp|x| - (1+|x|\,/2)^2|$ when $x \in [-10, -\varepsilon]$;
- MT10: $2x/(2+x) \leq \ln(1+x) + \varepsilon$ when $x \in [0,10]$;
- MT11: $x/\sqrt{1+x} \leq \ln(1+x) + \varepsilon$ when $x \in [-1/3, 0]$;
- MT12: $\ln((1+x)/x) \leq (12x^2+12x+1)/(12x^3+18x^2+6x)$ when $x \in [1/3, 10]$;
- MT13: $\ln((1+x)/x) \leq 1/\sqrt{x^2+x}$ when $x \in [1/3, 10]$;
- MT14: $\exp(x - x^2) \leq 1 + x + \varepsilon$ when $x \in [0,1]$;
- MT15: $\exp(-x/(1-x)) \leq 1 - x + \varepsilon$ when $x \in [-10, 1/2]$;
- MT16: $|\sin x| \leq 6/5 \cdot |x| + \varepsilon$ when $x \in [-1,1]$;
- MT17: $1 - 2x < \cos(\pi \cdot x)$ when $x \in [\varepsilon, 100/201]$;
- MT18: $0 \leq \cos x - 1 + x^2/2 + \varepsilon$ when $x \in [-10, 10]$;
- MT19: $8\sqrt{3} \cdot x/(3\sqrt{3} + \sqrt{75+80x^2}) \leq \arctan x + \varepsilon$ when $x \in [0,10]$;
- MT20: $1 < (x + 1/x) \cdot \arctan x$ when $x \in [\varepsilon, 10]$;
- MT21: $3x/(1+2\sqrt{1+x^2}) \leq \arctan x + \varepsilon$ when $x \in [0,10]$;
- MT22: $\cos x \leq \sin x/x$ when $x \in [\varepsilon, \pi]$;
- MT23: $\cos x < (\sin x/x)^2$ when $x \in [\varepsilon, \pi/2]$;
- MT24: $0 < \sin x/3 + \sin(3x)/6$ when $x \in [\pi/3, 2\pi/3 - \varepsilon]$;
- MT25: $12 - 14.2 \cdot \exp(-0.318 \cdot x) + (3.25 \cdot \cos(1.16 \cdot x) - 0.155 \cdot \sin(1.16 \cdot x)) \cdot \exp(-1.34 \cdot x) > 0$ when $x \in [0,2]$.

5.2 Experimental Results

For the experiments conducted within the PVS proof assistant, we have been using the `ProofLite` package by César Muñoz to perform the proofs in batch mode. These PVS

proofs have been performed by using one of the following strategies: `bernstein` (with PVS 5.0), `interval` (with PVS 6.0), and `metit` (with PVS 6.0 and MetiTarski 2.2). See Section 1.2 for more details and references.

As the HOL Light/`verify_ineq` decision procedure handles goals that are strict inequalities and supports sin, cos, but not tan, we manually adapted our reference problems to fit in this setting. We also rephrased some statements to remove the absolute value whenever possible. Still, some of our reference problems cannot be handled by the tool: this includes inequalities involving unsupported functions such as exp. The procedure was configured to perform computations with numbers using 5 radix-200 digits, as in [25]; this amounts to a precision of about 37 bits. The MT23 problem needed a slightly larger precision to be solved though, about 44 bits.

Table 1 shows the CPU time for proving our selection of problems. For these benchmarks, we have used a desktop computer running Ubuntu 14.04.2 LTS on an Intel Core i5-4460S CPU clocked at 2.90 GHz, along with the following tools: Coq 8.4pl6 and CoqInterval 2.0.0, Sollya 4.1, PVS 5.0 (for the Bernstein strategy), PVS 6.0 and nasalib 6.0.9 (for the PVS/interval and metit strategies), MetiTarski 2.2 and Z3 4.3.1 (bundled with nasalib 6.0.9), as well as development versions of HOL Light[13] (rev. 196) as well as of `flyspeck/formal_ineqs`[14] (rev. 3660) and NLCertify[15] (commit 9e85404). Regarding the compilers and the support libraries required for these benchmarks, we have been using the following versions: gcc 4.8.4 for x86_64-linux-gnu, OCaml 4.01.0, Camlp5 6.13 (but HOL Light relied on Camlp5 6.11), Flocq 2.4.0, Math-Comp 1.5.0, CSDP 6.1.1, GMP 5.1.3, MPFR 3.1.2, MPFI 1.5.1, fpLLL 4.0.4, BLAS 1.2.20110419, ATLAS 3.10.1, LAPACK 3.5.0, SDPA 7.3.9, MUMPS 4.10.0, OPAM 1.2.2, Zarith 1.3, LACAML 7.2.6. For systems that have no time limit *per se*, and may not terminate on some examples of our test suite, we have been setting up a timeout. For all systems, a timeout reported in the table means that they did not succeed after 180 seconds of computations.

Let us give a bit of information about the missing results first. Both PVS/Bernstein and HOL Light/`REAL_SOS` only handle polynomial systems, which explains why most of their columns are empty. HOL Light/`verify_ineq` and NLCertify were designed to tackle the inequalities from the Flyspeck project, so they have no support for functions such as exp. This explains the missing results for most of the tests extracted from MetiTarski [1].

The failure of MetiTarski on its own test might seem surprising. Yet in MetiTarski's testsuite,[16] MT19 is documented as "probably not provable", due to the "square root approximation degrading for large inputs". So it is not clear whether it is supposed to pass.

Regarding the parameters of CoqInterval, most tests pass with the default floating-point precision of 30 bits or by increasing it a bit to 40 bits. There are few exceptions though. The most notable one is the first test, which verifies the approximation used by CRlibm. Indeed, since it approximates the function with an accuracy of 62 bits, it is impossible to prove it with a precision lower than that, as shown in Figure 1. As a matter of fact, we had to ask for 90 bits of precision, for the proof to successfully go through.

When using Taylor models, we never had to use degrees higher than 5. As explained in Section 4.3.4, we are not yet able to fully extract the information contained in high-degree polynomials, so they would not speed up the bisection process anyway. The tests that

[13] `https://code.google.com/p/hol-light/`

[14] `https://code.google.com/p/flyspeck/source/browse/#svn/trunk/formal_ineqs`

[15] `http://forge.ocamlcore.org/projects/nl-certify`

[16] `https://metitarski.googlecode.com/hg-history/V2_4/tptp/Problems/atan-problem-1-sqrt.tptp`

Problems	CoqInterval	Sollya	MetiTarski	NLCertify (not verified)	NLCertify (partly verified)	PVS/interval	HOL Light/ `verify_ineq`	PVS/Bernstein	HOL Light/ `REAL_SOS`
crlibm_exp	0.83⋆	0.02	Failed	-	-	Failed	-	-	-
remez_sqrt	0.45	0.02	0.05	15.28⋆	Timeout	Failed	3.60⋆	-	-
abs_err_atan	0.45	0.01	0.07	Failed	Failed	Timeout	2.36⋆	-	-
rel_err_geodesic	3.10	2.24	Timeout	Timeout	Timeout	Failed	229.54⋆	-	-
harrison97	0.42	0.01	0.10	-	-	Failed	-	-	-
cos_cos_d2	0.71	0.05	Timeout	Timeout	Timeout	20.64	5.82⋆	-	-
cos_cos_d3	0.79	0.05	Timeout	Timeout	Timeout	48.87	6.28⋆	-	-
cos_cos_d4	0.91	0.06	Timeout	Timeout	Timeout	Timeout	8.83⋆	-	-
cos_cos_d5	1.44	0.06	Timeout	Timeout	Timeout	Timeout	15.70⋆	-	-
cos_cos_d6	1.54	0.07	Timeout	Timeout	Timeout	Timeout	20.92⋆	-	-
cos_cos_d7	2.21	0.07	Timeout	Timeout	Timeout	Timeout	41.88⋆	-	-
cos_cos_d8	2.79	0.08	Timeout	Timeout	Timeout	Timeout	87.78⋆	-	-
MT1	0.53	-	0.13	-	-	Failed	-	-	-
MT2	1.56	-	0.06	9.99⋆	Timeout	Failed	-	-	-
MT3	0.18	-	0.18	-	-	1.14	-	-	-
MT4	0.23	-	0.17	1.31⋆	18.95⋆	1.19	-	-	-
MT5	0.11⋆	-	0.05	-	-	1.24	-	-	-
MT6	0.15⋆	-	0.07	-	-	1.23	-	-	-
MT7	0.04	-	0.04	-	-	0.69	-	-	-
MT8	0.33	-	0.15	-	-	Timeout	-	-	-
MT9	0.52	-	0.46	-	-	Timeout	-	-	-
MT10	0.19	-	0.04	0.96	14.86	Failed	-	-	-
MT11	0.10	-	0.22	0.40	6.73	1.72	-	-	-
MT12	2.84	-	0.07	Timeout	Timeout	Timeout	-	-	-
MT13	0.98	-	0.07	11.82	137.91	Failed	-	-	-
MT14	0.07	-	0.06	-	-	0.89	-	-	-
MT15	0.15	-	0.07	-	-	0.98	-	-	-
MT16	0.13	-	0.02	0.58⋆	8.23⋆	3.23	0.57⋆	-	-
MT17	0.11	-	0.06	0.22	4.06	1.27	0.23	-	-
MT18	0.16	-	0.02	0.21	2.46	0.69	0.75	-	-
MT19	0.52	-	Failed	5.09	74.55	Failed	1.92	-	-
MT20	3.09	-	0.05	2.63	44.21	Timeout	15.54	-	-
MT21	0.33	-	0.38	3.69	51.94	Failed	1.37	-	-
MT22	0.69	-	0.06	Timeout	Timeout	Failed	113.74	-	-
MT23	1.17	-	0.12	Failed	Failed	Failed	86.90	-	-
MT24	0.10	-	0.36	0.17	2.38	Failed	0.24	-	-
MT25	0.29	-	0.17	-	-	1.78	-	-	-
RD	0.25	-	0.02	1.88	66.01	1.67	0.48	3.26	Timeout
adaptiveLV	0.16	-	0.04	0.23	3.18	1.00	1.26	4.02	3.78
butcher	0.42	-	0.05	0.73	11.08	19.99	2.21	18.23	Timeout
magnetism	0.17	-	0.05	1.35	20.60	Timeout	313.75	Timeout	0.24

Table 1 CPU time (in seconds) for proving the selected problems. Timeout indicates that the prover did not terminate under 180s. Failed indicates that it terminated but did not succeed in proving a problem. When some result is followed by a star, it means that the problem has been split into several sub-problems, and the given timing is the total CPU time for proving them. For example, this is the case when the input domain is not connected (*e.g.* for MT5, it is $[-1, -\varepsilon] \cup [\varepsilon, 1]$), or when some inequality with absolute values has been rephrased into a conjunction of inequalities. Regarding the PVS results, we do not include the proof time for the Type Correctness Conditions (TCCs) that are generated when proving the considered problems.

benefited from using a degree-5 polynomial are crlibm_exp, rel_err_geodesic, MT22, and MT23.

Regarding approximation errors, the only tools able to check them are Sollya, CoqInterval, and HOL Light/`verify_ineq`. Theoretically, PVS/interval should also be able to handle them, but due to the dependency effect, it cannot succeed in a reasonable amount of time. As for MetiTarski, it cannot prove more than what its predefined axioms allow, and thus cannot be used as a general tool for verifying approximation errors. As for performance, CoqInterval is much faster than `verify_ineq` on the most complicated examples, since it is not restricted to order-1 Taylor approximations. It is much slower than Sollya though, but its results are formally verified by Coq. Note that the advanced algorithms of Sollya fail on the rel_err_geodesic test and we had to fall back to `checkinfnorm` for that specific test, which is a rather slow algorithm.

On the tests extracted from MetiTarski, CoqInterval does not perform as well as MetiTarski. The comparison is not that unfavorable though, when one keeps in mind that all the results are formally verified by Coq. The column for PVS/interval gives us some clues as to which problems are simple enough to be handled by naive interval arithmetic.

It should be noted that, while all the problems of this category have bounded inputs, CoqInterval is able to prove some of the original problems with unbounded inputs. For instance, automatic differentiation has no difficulty with the unbounded versions of MT7, MT18, and MT20. For MT8 to go through with an unbounded domain, the user has to manually remove some absolute values beforehand. MT15 can also be proved with an infinite lower bound, as long as the user rewrites $x/(1-x)$ into $1/(1/x-1)$. A similar transformation makes it possible to prove MT19 and MT21. The running time of these seven problems is hardly changed when going from the bounded versions to the unbounded versions.

Finally, the quasi-multivariate problems show that CoqInterval can quite easily handle them. It also shows that some multivariate solvers would benefit from heuristics for detecting quasi-multivariate problems. For instance, the “magnetism” problem can be easily solved, once one has noticed that only x_1 matters.

6 Conclusion

6.1 Summary

This paper has presented a tactic for the Coq proof assistant that is designed to automatically and formally verify numerical bounds on univariate or quasi-multivariate expressions. While the original CoqInterval package was already designed for that purpose, it was not powerful enough to tackle bounds on approximation errors. Indeed, its approach was based on order-0 Taylor–Lagrange approximations, which would cause an explosion of the number of subdomains to verify for the harder cases. Using Taylor models instead, the execution time is sufficiently reduced for the tactic to become usable when verifying mathematical libraries.

Our approach is fully reflexive; starting from a representation of the real-valued expression, it numerically computes some bounds of it. The tactic does not depend on an external oracle that would generate certificates that the prover would check. Instead, all the numerical computations are performed inside the logic of Coq and are proved correct. In fact, one could extract the code from our tactic and build a native tool independent from Coq.

Let us summarize how our approach compares to the other tools. First, it only handles univariate and quasi-multivariate expressions, while Sollya [7] only supports univari-

ate expressions and all the other tools are meant to support truly multivariate expressions. It supports some elementary functions, contrarily to `REAL_SOS` for HOL Light [13] and `bernstein` for PVS [23]. It is restricted to the universally-quantified fragment, while MetiTarski [1] can tackle more intricate problems. (To a lesser extent, `bernstein` can also handle a wider range of problems.) Its computations are performed inside the logic of a proof system, contrarily to MetiTarski, Sollya, and to a lesser extent the PVS strategies. It performs numerical computations using floating-point arithmetic, as do Sollya and the `verify_ineq` procedure for HOL Light [25], while all the other tools use rational arithmetic.

Finally, if we exclude the tools that only support polynomial systems, the main difference between the remaining tools is whether they focus on univariate or multivariate expressions. Indeed, in the multivariate case, the size of Taylor-like approximations grow exponentially with the order, while they grow only linearly in the univariate case. This partly explains why HOL Light/`verify_ineq` and NLCertify manipulate only degree-2 polynomials as approximations. Unfortunately, this low degree is not sufficient to tackle the kind of proofs required when verifying mathematical libraries, since it leads to a large number of subdivisions in that case.

6.2 Perspectives

While CoqInterval has now reached the point where it can formally prove some of the bounds that appear when verifying a floating-point mathematical library, there is still work to be done. First of all, additional floating-point approximations of mathematical functions should be formalized. Unfortunately, implementing efficient approximations is still a costly process that involves lots of custom code and proofs. It would be better to get a more generic way of formalizing floating-point approximations. A promising approach is to automatically derive implementations from the differential equation of a function.

Another part that could be improved in CoqInterval is the interface between the floating-point formalization and the interval one. Indeed, it leaks numerous implementation details about the floating-point operations, *e.g.* the unbounded range of exponent, while the interval operations should only care about inequalities such as $\nabla(u + v) \leq u + v \leq \triangle(u + v)$. As a consequence, one cannot blindly replace a floating-point implementation by another. For instance, using a library like MPFR or the floating-point unit of the processor for improved performance would invalidate all the proofs by breaking some assumptions.

Finally and more importantly, some work has to be done in bridging the gap between the univariate approaches and the multivariate ones. Multivariate methods obviously handle a larger spectrum of problems, but they are quite inefficient when it comes to univariate problems coming from the verification of approximation bounds for mathematical libraries.

Acknowledgements We would like to thank the people from the ANR TaMaDi project for initiating and greatly contributing to the CoqApprox project.

References

1. Akbarpour, B., Paulson, L.C.: MetiTarski: An automatic theorem prover for real-valued special functions. Journal of Automated Reasoning **44**(3), 175–205 (2010). DOI 10.1007/s10817-009-9149-2
2. Allamigeon, X., Gaubert, S., Magron, V., Werner, B.: Certification of bounds of non-linear functions: The templates method. In: J. Carette, D. Aspinall, C. Lange, P. Sojka, W. Windsteiger (eds.) Intelligent Computer Mathematics - MKM, Calculemus, DML, and Systems and Projects, *Lecture Notes in Computer Science*, vol. 7961, pp. 51–65 (2013). DOI 10.1007/978-3-642-39320-4_4

3. Boldo, S., Melquiond, G.: Flocq: A unified library for proving floating-point algorithms in Coq. In: E. Antelo, D. Hough, P. Ienne (eds.) Proceedings of the 20th IEEE Symposium on Computer Arithmetic, pp. 243–252. Tübingen, Germany (2011). DOI 10.1109/ARITH.2011.40
4. Brisebarre, N., Joldeş, M., Martin-Dorel, É., Mayero, M., Muller, J.M., Paşca, I., Rideau, L., Théry, L.: Rigorous polynomial approximation using Taylor models in Coq. In: A. Goodloe, S. Person (eds.) Proceedings of 4th International Symposium on NASA Formal Methods, *Lecture Notes in Computer Science*, vol. 7226, pp. 85–99. Springer, Norfolk, Virginia (2012). DOI 10.1007/978-3-642-28891-3_9
5. Ceberio, M., Granvilliers, L.: Horner's rule for interval evaluation revisited. Computing **69**(1), 51–81 (2002). DOI 10.1007/s00607-002-1448-y
6. Chevillard, S., Harrison, J., Joldeş, M., Lauter, C.: Efficient and accurate computation of upper bounds of approximation errors. Journal of Theoretical Computer Science **412**(16), 1523–1543 (2011). DOI 10.1016/j.tcs.2010.11.052
7. Chevillard, S., Joldeş, M., Lauter, C.: Sollya: An environment for the development of numerical codes. In: K. Fukuda, J. van der Hoeven, M. Joswig, N. Takayama (eds.) Proceedings of the 3rd International Congress on Mathematical Software, *Lecture Notes in Computer Science*, vol. 6327, pp. 28–31. Heidelberg, Germany (2010)
8. Daumas, M., Lester, D., Muñoz, C.: Verified real number calculations: A library for interval arithmetic. IEEE Transactions on Computers **58**(2), 226–237 (2009)
9. Daumas, M., Melquiond, G., Muñoz, C.: Guaranteed proofs using interval arithmetic. In: P. Montuschi, E. Schwarz (eds.) Proceedings of the 17th IEEE Symposium on Computer Arithmetic, pp. 188–195. Cape Cod, MA, USA (2005). DOI 10.1109/ARITH.2005.25
10. Denman, W., Muñoz, C.: Automated real proving in PVS via MetiTarski. In: C.B. Jones, P. Pihlajasaari, J. Sun (eds.) FM, *Lecture Notes in Computer Science*, vol. 8442, pp. 194–199. Springer (2014). DOI 10.1007/978-3-319-06410-9_14
11. Hansen, E., Walster, G.: Global Optimization Using Interval Analysis: Revised And Expanded. Monographs and textbooks in pure and applied mathematics. CRC Press (2003)
12. Harrison, J.: Verifying the accuracy of polynomial approximations in HOL. In: E.L. Gunter, A.P. Felty (eds.) Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics, *Lecture Notes in Computer Science*, vol. 1275, pp. 137–152. Murray Hill, NJ, USA (1997). DOI 10.1007/BFb0028391
13. Harrison, J.: Verifying nonlinear real formulas via sums of squares. In: K. Schneider, J. Brandt (eds.) Proceedings of the 20th International Conference on Theorem Proving in Higher Order Logics, *Lecture Notes in Computer Science*, vol. 4732, pp. 102–118. Kaiserslautern, Germany (2007)
14. Joldeş, M.: Rigorous Polynomial Approximations and applications. Ph.D. thesis, ENS de Lyon, France (2011). URL `http://tel.archives-ouvertes.fr/tel-00657843/en/`
15. Makino, K.: Rigorous analysis of nonlinear motion in particle accelerators. Ph.D. thesis, Michigan State University, East Lansing, Michigan, USA (1998)
16. Makino, K., Berz, M.: Taylor models and other validated functional inclusion methods. International Journal of Pure and Applied Mathematics **4**(4), 379–456 (2003)
17. Martin-Dorel, É., Mayero, M., Paşca, I., Rideau, L., Théry, L.: Certified, efficient and sharp univariate Taylor models in Coq. In: SYNASC 2013, pp. 193–200. IEEE, Timişoara, Romania (2013). DOI 10.1109/SYNASC.2013.33
18. Melquiond, G.: Floating-point arithmetic in the Coq system. In: Proceedings of the 8th Conference on Real Numbers and Computers, pp. 93–102. Santiago de Compostela, Spain (2008)
19. Melquiond, G.: Proving bounds on real-valued functions with computations. In: A. Armando, P. Baumgartner, G. Dowek (eds.) Proceedings of the 4th International Joint Conference on Automated Reasoning, *Lecture Notes in Artificial Intelligence*, vol. 5195, pp. 2–17. Sydney, Australia (2008). DOI 10.1007/978-3-540-71070-7_2
20. Melquiond, G.: Floating-point arithmetic in the Coq system. Information and Computation **216**, 14–23 (2012). DOI 10.1016/j.ic.2011.09.005
21. Moore, R.E.: Interval Analysis. Prentice-Hall (1966)
22. Muller, J.M., Brisebarre, N., de Dinechin, F., Jeannerod, C.P., Lefèvre, V., Melquiond, G., Revol, N., Stehlé, D., Torres, S.: Handbook of Floating-Point Arithmetic. Birkhäuser Boston (2010). DOI 10.1007/978-0-8176-4705-6
23. Muñoz, C., Narkawicz, A.: Formalization of a representation of Bernstein polynomials and applications to global optimization. Journal of Automated Reasoning **51**(2), 151–196 (2013). DOI 10.1007/s10817-012-9256-3
24. Narkawicz, A., Muñoz, C.: A formally verified generic branching algorithm for global optimization. In: E. Cohen, A. Rybalchenko (eds.) Proceedings of the 5th International Conference on Verified Software: Theories, Tools, Experiments, *Lecture Notes in Computer Science*, vol. 8164, pp. 326–343. Menlo Park, CA, USA (2013). DOI 10.1007/978-3-642-54108-7_17

25. Solovyev, A., Hales, T.C.: Formal verification of nonlinear inequalities with Taylor interval approximations. In: G. Brat, N. Rungta, A. Venet (eds.) Proceedings of the 5th International Symposium on NASA Formal Methods, *Lecture Notes in Computer Science*, vol. 7871, pp. 383–397. Moffett Field, CA, USA (2013). DOI 10.1007/978-3-642-38088-4_26
26. Tang, P.T.P.: Table-driven implementation of the exponential function in IEEE floating-point arithmetic. ACM Transactions on Mathematical Software **15**(2), 144–157 (1989). DOI 10.1145/63522.214389
27. Ziv, A.: Fast evaluation of elementary mathematical functions with correctly rounded last bit. ACM Transactions on Mathematical Software **17**(3), 410–423 (1991). DOI 10.1145/114697.116813
28. Zumkeller, R.: Global Optimization in Type Theory. Ph.D. thesis, École polytechnique, France (2008). URL `http://alacave.net/~roland/FormalGlobalOpt.pdf`