TECHNICAL REPORT

IRIT/RR–2014-09–FR

Proving Tight Bounds on Univariate Expressions in Coq

Érik Martin-Dorel[1] and Guillaume Melquiond[2]

[1]*Université Paul Sabatier, IRIT, équipe ACADIE.*
[2]*Inria Saclay–Île-de-France, LRI, UMR 8623 CNRS,*
PCRI, bât 650, Université Paris-Sud, 91405 Orsay Cedex, France.

November 24, 2014

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 traditional tools. In fact, the inherent difficulty of computing such bounds often mandate 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 kernel 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.

Contents

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 it is critical that libraries document the accuracy of the computed values. In fact, some library authors go as far as publishing mathematical proofs showing that the documented accuracy is correct.

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 $|x| \le 355 \cdot 2^{-22}$, we have the following bound:

$$\left| \frac{x + 0.5 \cdot x^2 + c_3 x^3 + c_4 x^4 - \exp x + 1}{\exp x - 1} \right| \le 2^{-62} \qquad (1)$$

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 certainly 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 most tools fail to plot the relative error should not come as a surprise. Indeed, their plotting procedures perform binary64 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 requires at least 80 bits for its plot to look sensible.

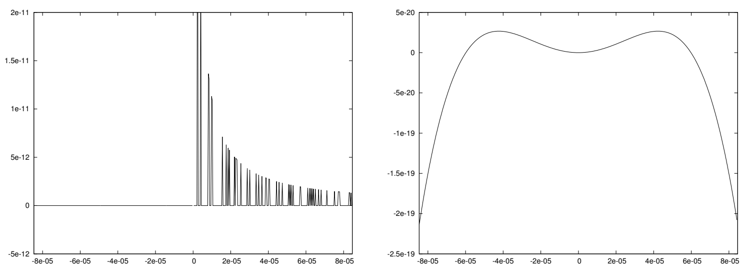

Figure 1: On the left, the function from Equation (1), as plotted by tools relying on binary64 floating-point numbers. On the right, its actual graph, as plotted by Sollya.[3]

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. 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/`
[3]`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 [17] and a decision procedure based on interval arithmetic was built on top of it [18]. 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 [19]. Independently developed, the CoqApprox library was built on top of the interval kernel of CoqInterval. It was designed to compute guaranteed polynomial approximations of univariate functions which can then be fed to programs solving the Table-Maker's Dilemma [4, 16].

So, at that point in time, we had, on one side, a decision procedure for verifying bounds on univariate functions and, on the other side, a library able to generate accurate polynomial approximations. 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 and thus available at

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

Before going any further, let us specify what univariate means in the context of this paper. The first interpretation is: only one variable can occur in an expression; all the other arity-0 symbols are numeric constants. This is a bit too restrictive. Indeed, the approach we present is based on interval arithmetic, so the amount of variables does not really matter. A better interpretation would therefore be that, for tight bounds to be computed, among all the variables of the expression, only one occurs several times. This requirement can 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 is quite restrictive, hence the usage of "univariate" rather than "multivariate".

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, they are never incorrect, by design. It does not generate any proof.

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 [12]. 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,[4] a new HOL Light procedure `verify_ineq` was designed in order to support multivariate expressions involving some elementary functions [23]. An external tool first precomputes a suitable subdivision of the input domain. Then HOL Light computes on each subdomain an order-1 Taylor–Lagrange polynomial approximation (that is, with a quadratic remainder) 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 [21]. 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 [22].

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.[5]

1.3 Content

Interval arithmetic is a well-known tool to compute bounds on real-valued expressions. In most settings, an interval is a closed subset of the real numbers which is represented by a pair of bounds $[a,b]$. 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 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 shows how to formalize an interval arithmetic and how to compute with it inside 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, they might be so grossly overestimated that it is impossible to deduce anything interesting from them. This issue appears as soon as a variable occurs several times in an expression evaluated by interval computations. Indeed, interval arithmetic does not track the dependencies between all these occurrences. Section 3 presents three approaches that we have formalized in Coq to alleviate this issue: bisection, automatic differentiation, and Taylor models.

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

[5]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.

With both an effective interval arithmetic and some ways to reduce the dependency effect, it is possible to implement some tools to automatically and formally prove tight bounds on real-valued univariate expressions. They are packaged in the `interval` tactic for Coq. It converts a goal to be proved into a problem that is hopefully solvable using interval computations. Section 4 presents this *reflection*-based approach. We also compare the performances 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 Floating-point and Interval Arithmetic

2.1 Preliminaries About Interval Arithmetic

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

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

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

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

Interval operators usually satisfy some other properties, such as *isotonicity*, but they are somewhat useless when proving the correctness of an algorithm.[6]

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.* $\boldsymbol{f} \circ \boldsymbol{g}$ is an interval extension of $f \circ g$ [20]. Thus one only needs to build interval extensions of basic operators in order to perform interval computations.

An interval $\boldsymbol{x}$ is represented by a pair $[\underline{x}, \overline{x}]$ of bounds. 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 possible to easily represent. For instance, in order to prove that $\forall x,\ x \le 0 \Rightarrow f(x) \ge 0$, one can just compute the following interval inclusion: $\boldsymbol{f}([-\infty, 0]) \subseteq [0, +\infty]$.

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:

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

Notice that an interval subtraction can be performed at the cost of just two bound subtractions, which makes it an efficient operation. As a rule of thumb, 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:

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

[6]Isotonicity means that, the tighter $\boldsymbol{x}$ is, the tighter $\boldsymbol{f}(\boldsymbol{x})$ is. This property does not occur in any of the proofs, but it helps with performances.

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 the performances to be dreadful.

This issue is not new and people have dealt with it since day one. 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 $\vartriangle$ that, given a real number, return a bound that is smaller, respectively larger. The interval extension of exp can then be defined as

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

Note that these two functions $\triangledown$ and $\vartriangle$ 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. 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.

Section 2.2 gives an overview of the floating-point kernel we are using. Section 2.3 presents the interval type and how to define interval operators for addition, multiplication, division, and so on, once the corresponding floating-point operators have been formalized. Section 2.4 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.

2.2 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. Their set is

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

While most algorithms can cope with an odd radix, they are often more efficient when the radix is even. In fact, the parser of the `interval` tactic assumes $\beta = 2$. So let us suppose β to be set once and for all; we will simply write $\mathbb{F}$ instead of $\mathbb{F}_\beta$.

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 not an issue, not even a source of inefficiency, 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 $\triangledown$ and $\vartriangle$. 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 library increases the precision of intermediate computations on-the-fly, if the current precision would lead to grossly inaccurate results; this will be detailed in Section 2.4.

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

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

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

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

By definition, we have $\nabla(x) \le x \le \triangle(x)$ for any real x. So these operators are sufficient to ensure the containment property. The definitions of $\mathbb{F}$, of $\mathbb{G}_p$, 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[7] or MPFR.[8] 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 kernel of CoqInterval is that the exponent range of the numbers in $\mathbb{G}_p$ is unbounded, so underflow and overflow never occur.

The definitions of ∇ and $\triangle$ in Equation (5) are suitable for proofs, but they do not offer a way to actually compute the results of the operators. This makes them useless for the purpose of automated proof. So the library also provides some functions that, given a value of $\mathbb{F}$ compute a value of $\mathbb{G}_p$. For instance, `Fround_at_prec`(β,*mode*,p,$m \cdot \beta^e$) returns the element of $\mathbb{G}_p$ the closest to $m \cdot \beta^e$, with the meaning of closest being controlled by *mode* (∇ or $\triangle$). 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 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 this 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 are not that efficient. In particular, they do not take advantage of the value of β or of the regularity of $\mathbb{G}_p$ (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 $\beta = 2$, 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 bits that are equal to zero.

In the end, our 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, so the user can swap one for the other. In

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

particular, the interval operators do not care about the actual implementation of the floating-point kernel.

This kernel just has to provide the computable versions 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.

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 behave as if they were part of the option monad. Note that, in general, having such an absorbing element is not that useful for formalizing floating-point arithmetic. In fact, Flocq does not even support it. Its point will be clearer when it comes to interval arithmetic.

2.3 Interval Operators

As explained before, in order to support half-bounded intervals, one wants to be able 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 snippets 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 subtraction from a kernel of floating-point arithmetic, and `rnd_DN` and `rnd_UP` being the directions $\triangledown$ and $\vartriangle$.

```
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 needs to be 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

$$\mathbf{mul}(\boldsymbol{u},\boldsymbol{v}) = [\triangledown(\min(\underline{u}\underline{v},\overline{u}\underline{v},\underline{u}\overline{v},\overline{u}\overline{v})), \vartriangle(\max(\underline{u}\underline{v},\overline{u}\underline{v},\underline{u}\overline{v},\overline{u}\overline{v}))].$$

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 more complicated algorithm that returns 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 singleton 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 $\boldsymbol{x}^2$ as $\mathbf{mul}(\boldsymbol{x},\boldsymbol{x})$ would give grossly overestimated results if $\boldsymbol{x}$ straddles zero. More generally, computing $\boldsymbol{x}^{m+n}$ as $\mathbf{mul}(\boldsymbol{x}^m, \boldsymbol{x}^n)$ (as would a fast exponentiation algorithm) is a poor choice in interval arithmetic. 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 $\boldsymbol{x}$, 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 $k \in [-1, 2]$. For the other powers, the bounds of the result might be off by a few units in the last place (while still satisfying the containment property), for the sake of speed.

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 will show up when implementing the tactic, as it allows 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, we no longer use a bold face for the basic interval operators, unless there is an ambiguity. In fact, we will simply use the infix operators, as if the inputs were real numbers rather than intervals. The bold face will still be used for composite interval functions, as the dependency effect causes them to be a large overestimation of set-extension functions, hence the need to set them apart.

2.4 Floating-point and Interval 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 correctly-rounded approximations of elementary functions. 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 $y = \triangledown(\exp(x))$. Indeed, while there are simple algorithms for computing $\triangledown(\exp(x))$, proving their termination is next to impossible in Coq.[9] 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.

Since the precision parameter is unbounded, methods based on fixed polynomial evaluation cannot be used. So we evaluate truncated power series instead. There are two difficulties: knowing at which point to truncate the series and knowing how much the partial sum should be overestimated to account for the part that was truncated away. There are techniques to solve these two issues, as can be seen in MPFR, but they are still too complicated for our purpose. 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 with a Taylor series $f(x) = \sum(-1)^k a_k x^k$. Let us assume

[9]The idea of the proof is as follows. If such an algorithm were to not terminate, it would mean that $\exp(x)$ is a rational number, which is impossible when $x \neq 0$.

that the sequence $k \mapsto a_k x^k$ is positive decreasing. Then we have the following inequalities:

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

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 singleton $[x, x]$, so the overestimation stays in check. But for interval extensions, the results would be so wide that they would be meaningless.

We now have a way to compute an elementary function when x is close to 0, but we are not yet out of the woods though. We still have to invert the argument reduction process. Unfortunately, for a function like cos, the 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 performances. 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 [19].

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 $[-2\pi, 2\pi]$.

3 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 $\boldsymbol{x} - \boldsymbol{x}$. Let us perform this interval computation and see what we can deduce about $x - x$. For $\boldsymbol{x} = [0, 1]$, we have

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

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 $|f(x) - \tilde{f}(x)| \le \varepsilon$ with

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

This section presents the various methods we have formalized to reduce the dependency effect, from the simplest one to the most complicated. The most naive one is the bisection (Section 3.1), then comes an approach based on automatic differentiation (Section 3.2), and finally the most powerful approach using Taylor models (Section 3.3).

3.1 Bisection

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

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

This approach only works if the right-hand side is tighter than $\boldsymbol{f}(\boldsymbol{x})$ (which it fortunately is, by isotonicity). If we consider the $x - x$ example again, we can see that, by using $\boldsymbol{x}_1 = [0, 1/2]$ and $\boldsymbol{x}_2 = [1/2, 1]$, 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 good rule of thumb when it comes to the dependency effect.

This approach is far from a panacea though. As can be seen on the $x - x$ example, 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 $|x - x| \le 10^{-12}$, 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, it has a property that makes it interesting to have. It can be combined easily with the other approaches to reduce the dependency effect. Assume that the other approaches fail because of some singularity in the input interval $\boldsymbol{x}$. Then the bisection will be able to isolate the singularity in a tiny interval $\boldsymbol{x}_2$ and let the other approaches deal with $\boldsymbol{x} \setminus \boldsymbol{x}_2 = \boldsymbol{x}_1 \cup \boldsymbol{x}_3$.

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

```
bisect($n$,$\boldsymbol{x}$) =
  if $n = 0$ then $false$
  else if $chk(\boldsymbol{x})$ then $true$
  else let $m$ be the midpoint of $\boldsymbol{x}$ in
    bisect($n-1$,$[\underline{x}, m]$) && bisect($n-1$,$[m, \overline{x}]$)
```

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 $\boldsymbol{x}$. At each step, either $chk(\boldsymbol{x})$ returns *true*, or $\boldsymbol{x}$ is split into two sub-intervals and the `bisect` function is recursively called on each sub-intervals. If `bisect`$(n,\boldsymbol{x})$ evaluates to *true*, then for any predicate P over real numbers, we have

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

So, when trying to prove some property $\forall x \in \boldsymbol{x},\ P(x)$, 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 $|f(x) - \tilde{f}(x)| \le \varepsilon$. We define $chk(\boldsymbol{u})$ as $|\boldsymbol{f}(\boldsymbol{u}) - \tilde{\boldsymbol{f}}(\boldsymbol{u}))| \subseteq [-\infty, \varepsilon]$. The left-hand side of Equation (6) thus becomes a straightforward consequence of the containment property.

3.2 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 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 $\boldsymbol{f'}(x)$ 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 remind the Taylor–Lagrange formula at order 0 (also known as the mean-value theorem). Assuming that f is differentiable over $\boldsymbol{x}$ and that $x_0 \in \boldsymbol{x}$, we have

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

Weakening it using the containment property, it becomes

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

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

To automate this approach, we need some way to compute $\boldsymbol{f'}(\boldsymbol{x})$. 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 arithmetic operations:

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

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 $(\boldsymbol{f}(\boldsymbol{x}), \boldsymbol{f'}(\boldsymbol{x}))$ at the end of the computation. Once we have $\boldsymbol{f'}(\boldsymbol{x})$, we compute $\boldsymbol{f}([x_0,x_0])$ 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 $(\boldsymbol{u},\boldsymbol{u'})$ to indicate that the sub-expression is not differentiable at some points of $\boldsymbol{x}$ (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.

Automatic differentiation works rather fine, but there is still some dependency effect when computing $(\boldsymbol{x} - [x_0,x_0]) \times \boldsymbol{f'}(\boldsymbol{x})$. We can do slightly better in some cases. Indeed, if the interval $\boldsymbol{f'}(\boldsymbol{x})$ happens to be of constant sign, then we can prove that f is monotonic on $\boldsymbol{x}$. As a consequence, we can just compute $\boldsymbol{f}([\underline{x},\underline{x}])$ and $\boldsymbol{f}([\overline{x},\overline{x}])$, take their convex hull, and thus obtain a superset of $f(\boldsymbol{x})$. Since the input intervals are singleton, 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 $\boldsymbol{f'}$ 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.

3.3 Taylor Models

3.3.1 Preliminaries about Taylor models

Borrowing the term chosen in [13], we call *rigorous polynomial approximation* (RPA) a pair $(P, \mathbf{\Delta})$ where P is a univariate polynomial in a given polynomial basis, and $\mathbf{\Delta}$ is an interval error. Over a given interval $\boldsymbol{I}$, an RPA $(P, \mathbf{\Delta})$ represents a whole set of functions, namely the functions $f : \mathbb{R} \to \mathbb{R}$ such that

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

or more concisely $f \in [\![(P, \mathbf{\Delta})]\!]_{\boldsymbol{I}}$. Typical instances of univariate RPAs include *Taylor models* ([15, 14] and [13, Chap. 2]), where the polynomial P is represented in the Taylor polynomial basis,[10] and *Chebyshev models* [13, Chap. 4].

We will work with Taylor models $(\boldsymbol{P}, \mathbf{\Delta})$ where the polynomial $\boldsymbol{P}$ has small interval coefficients. This choice simplifies the algorithms within an approximate arithmetic, as the rounding errors are directly handled by the interval computations. This specific kind of Taylor models will be used in all the intermediate steps of the algorithms. If need be, we could turn an interval Taylor model $(\boldsymbol{P}, \mathbf{\Delta})$ into a standard Taylor model $(P, \mathbf{\Delta}')$ satisfying Equation (7), by picking the P_i inside the $\boldsymbol{P}_i$ and accumulating the errors into $\mathbf{\Delta}'$.

3.3.2 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, 16]. The main data type involved in the formalization is a record `rpa` (for rigorous polynomial approximation) that combines a polynomial `approx` and an interval `error`; it is parameterized by a type for representing polynomials and another type for intervals. We say that an `rpa` $(\boldsymbol{P}, \mathbf{\Delta})$ is a Taylor model for a given function $f : \mathbb{R} \to \mathbb{R}$ if it satisfies the following predicate (cf. [16, Definition 1]):

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

The library first provides some Coq functions for computing Taylor models of *basic functions* such as square root, exponential, or sine. For instance, `TM_exp` $(p, \boldsymbol{x_0}, \boldsymbol{I}, n)$ returns an `rpa` centered at $\boldsymbol{x_0}$ that approximates exp over the interval $\boldsymbol{I}$. Here, p is the working precision of the floating-point computations and n is the order of the Taylor model. The library then makes it possible to combine these Taylor models according to some *composite functions*. Each operation $+$, $-$, $\times$, $\div$, and $\circ$ (composition), corresponds to a dedicated algorithm [13]. Note that for division, we currently handle a function $\frac{f}{g}$ as $f \times (\mathrm{inv} \circ g)$ (as in [13]) 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:

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

[10] In the univariate case, $P(x) = \sum_{i=0}^{n} P_i \cdot (x - x_0)^i$ for a given expansion point $x_0 \in \boldsymbol{I}$.

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

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

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.

The formalization of these algorithms was carried out with a special focus on modularity. Relying on the module system of Coq, we reused CoqInterval's interface for intervals, and defined a dedicated interface for polynomials. The Taylor model algorithms are then implemented and proved in a parameterized module (*i.e.* a functor) that depends on these interfaces. This approach makes it easier for the user to swap a given implementation of the basic data structures for another one, possibly more efficient (*e.g.* floating-point bounds based on the `BigZ` type of integers in place of the `Z` type of integers).

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 $(\boldsymbol{P}, \boldsymbol{\Delta})$ for some function f, we only have to implement two algorithms: an interval extension $\boldsymbol{f}$ of f (see Section 2.4) 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 $c_k = F(c_{k-1}, k)$ with $c_k = f^{(k)}(x)/k!$.

The Taylor model is then computed as follows. For $\boldsymbol{P}$, the recurrence is "unrolled" by a higher-order function, *e.g.*, $\texttt{trec1}(F, \boldsymbol{f}(\boldsymbol{x_0}), n)$ for an order-1 recurrence F as above. For $\boldsymbol{\Delta}$, we mainly use Zumkeller's technique [16, Algorithm 1]. It gives sharp bounds when it detects that the function $R_n(f, x_0)(x) := f(x) - \sum_{k=0}^{n} f^{(k)}(x_0)/k! \cdot (x - x_0)^k$ is monotonic on each side of x_0, which is usually the case for the basic functions we are interested in. The formal verification of this technique benefited from the pen-and-paper proof of Proposition 2.2.1 in Mioara Joldeş' thesis [13], which relies on Lemma 5.12 in Roland Zumkeller's thesis [25].

3.3.3 Integration of CoqApprox into CoqInterval

The integration of CoqApprox into CoqInterval is performed through a generic interface `UnivariateApprox` for approximating univariate expressions. This extra layer will make it possible to easily swap our Taylor model implementation with another implementation of polynomial approximations, such as Chebyshev models.

This interface is a parameterized signature that depends on an implementation of intervals. It declares an abstract data type `T` and a ternary relation $\mathcal{A}(\boldsymbol{I}, t, f)$ which holds if t of type `T` is an approximation of function f over the interval $\boldsymbol{I}$. It also declares abstract functions to build approximations, starting from identity or constants, 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

$$\texttt{exp\_correct} : \forall u, \boldsymbol{I}, t, f, \quad \mathcal{A}(\boldsymbol{I}, t, f) \implies \mathcal{A}(\boldsymbol{I},\ \texttt{exp}(u, \boldsymbol{I}, t),\ \exp \circ f). \tag{9}$$

The interface also declares a function `eval` for computing the range of an expression from its approximation with the following correctness claim:

$$\texttt{eval\_correct} : \forall u, \boldsymbol{I}, t, f, \quad \mathcal{A}(\boldsymbol{I}, t, f) \implies \forall \boldsymbol{J},\ \forall x \in \boldsymbol{J},\ f(x) \in \texttt{eval}(u, t, \boldsymbol{I}, \boldsymbol{J}).$$

The `eval` function takes two interval arguments $\boldsymbol{I}$ and $\boldsymbol{J}$ so that one can evaluate an approximation that is valid on $\boldsymbol{I}$ over multiple subsets $\boldsymbol{J} \subsetneq \boldsymbol{I}$. This additional flexibility is not currently used by the tactic presented in the upcoming Section 4; we will just call the function with $\boldsymbol{J} := \boldsymbol{I}$.

Finally, an implementation of this interface `UnivariateApprox` is provided in the form of a functor, which depends on an implementation of intervals, and which uses the CoqApprox algorithms. Regarding the implementation of the abstract data type `T`, we do not directly use the type `rpa` for efficiency reasons. Instead, we add three extra constructors to handle some specific families of functions:

```
Inductive T := Dummy | Const of 𝕀 | Var | Tm of rpa.
```

`Dummy` approximates any function. `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 $\boldsymbol{x_0}$ 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 singleton interval built upon the midpoint of the interval $\boldsymbol{I}$.

4 The `interval` Tactic

The kernel of interval arithmetic and the three approaches for reducing the dependency effect have been packaged into two tactics: `interval_intro` and `interval`. When the goal is an inequality, calling `interval` tries to formally prove it. 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*,
- "`i_bisect_diff` *x*" asks for a bisection and an automatic differentiation of expressions with respect to variable *x*,
- "`i_bisect_taylor` *x d*" asks for a bisection and the computation of degree-*d* Taylor models with respect to variable *x*.

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 performances, 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.

4.1 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, 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 for doing 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.

4.2 Power and Shortcomings

For a univariate expression on a bounded input interval, as long as the bounds the user wants to prove are not attained for some value in the input interval, 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 interval is a subset of its domain. As a consequence, if the proved bounds 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.

That is it for the theoretical semi-decidability of our approach; let us see some of its shortcomings. One of them comes from the bisection process. Since it 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.

There are also some issues with the approach based on automatic differentiation, since 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

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

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 inflate 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.

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 $(\mathbf{1} \times \boldsymbol{x} + \mathbf{0}) \times \boldsymbol{x} + \mathbf{0}$, which amounts to computing $\boldsymbol{x} \times \boldsymbol{x}$. Let us see what happens for the input interval $\boldsymbol{x} = [-1, 1]$. (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:

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

If we assume that the higher-degree part of polynomials induces only negligible correlations, this rewriting gives an optimal enclosure without being much slower than Horner's rule. Because of the bisection process, this assumption is ultimately valid, though it might have a large overhead.

4.3 Performances

We have compared our tactic to the tools described in Section 1.2. To do so, we have selected several problems and tested whether they successfully verified them, and if so, how long it took to prove them.

4.3.1 Selection of some reference problems

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

- CRlibm exponential: $|(x + 0.5 \cdot x^2 + 6004799504235417 \cdot 2^{-55} \cdot x^3 + 1501199876148417 \cdot 2^{-55} \cdot x^4 - \exp x + 1)/(\exp x - 1)| \le 2^{-62}$ when $2^{-20} \le |x| \le 355 \cdot 2^{-22}$;

- square root [18]: $|\sqrt{x} - (((((122/7397 \cdot x - 1733/13547) \cdot x + 529/1274) \cdot x - 767/999) \cdot x + 407/334) \cdot x + 227/925)| \le 5 \cdot 2^{-16}$ when $x \in [0.5, 2]$;

- arctangent, with a tighter bound w.r.t. [8, p. 235]: $|\arctan x - (x - 11184811/33554432 \cdot x^3 - 13421773/67108864 \cdot x^5))| \le 5 \cdot 2^{-28}$ when $|x| \le 1/30$;

- Earth's radius of curvature [9, 18]: $|(r(\phi) - p((715/512)^2 - \phi^2))/r(\phi)| \le 23 \cdot 2^{-24}$ 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 [11, 24]: $|(\exp x - 1) - (x + 8388676 \cdot 2^{-24} \cdot x^2 + 11184876 \cdot 2^{-26} \cdot x^3)| \le (23/27) \cdot 2^{-33}$ when $|x| \le 10831 \cdot 10^{-6}$.

We have also crafted some increasingly-difficult approximation problems using the minimax relative polynomial approximations of $f(x) = \cos(1.5 \cdot \cos x)$ for $x \in [-1, 1/2]$. The goal is to prove $|(p_n(x) - f(x))/f(x)| \le C_n$ 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 [21, 23]. While multivariate (up to 7 variables), they fit into our definition of univariate expressions and thus were a sensible choice:

- 3-variable reaction diffusion: $-36.7126907 \le -x_1 + 2 \cdot x_2 - x_3 - 0.835634534 \cdot x_2 \cdot (1 + x_2)$ when $x_1, x_2, x_3 \in [-5, 5]$;
- adaptive Lotka-Volterra system: $-20.801 \le x_1 \cdot x_2^2 + x_1 \cdot x_3^2 + x_1 \cdot x_4^2 - 1.1 \cdot x_1 + 1$ when $x_1, x_2, x_3, x_4 \in [-2, 2]$;
- Butcher's problem: $-1.44 \le x_6 \cdot x_2^2 + x_5 \cdot x_3^2 - x_1 \cdot x_4^2 + x_4^3 + x_4^2 - x_1/3 + 4/3 \cdot x_4$ 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: $-0.25001 \le x_1^2 + 2 \cdot x_2^2 + 2 \cdot x_3^2 + 2 \cdot x_4^2 + 2 \cdot x_5^2 + 2 \cdot x_6^2 + 2 \cdot x_7^2 - x_1$ 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 $2|x|/(2+x) \le |\ln(1+x)|$ 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: $2|x|/(2+x) \le |\ln(1+x)| + \varepsilon$ when $x \in [-1+\varepsilon, 10]$;
- MT2: $|\ln(1+x)| \le -\ln(1-|x|) + \varepsilon$ when $x \in [-1+\varepsilon, 1-\varepsilon]$;
- MT3: $|x|/(1+|x|) \le |\ln(1+x)| + \varepsilon$ when $x \in [-1+\varepsilon, 1]$;
- MT4: $|\ln(1+x)| \le |x|(1+|x|)/|1+x| + \varepsilon$ 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: $|\exp x - 1| \le \exp|x| - 1$ when $x \in [-10, -\varepsilon]$;
- MT8: $|\exp x - (1+x)| \le |\exp|x| - (1+|x|)|$ when $x \in [-10, -\varepsilon]$;
- MT9: $|\exp x - (1+x/2)^2| \le |\exp|x| - (1+|x|/2)^2|$ when $x \in [-10, -\varepsilon]$;
- MT10: $2x/(2+x) \le \ln(1+x) + \varepsilon$ when $x \in [0, 10]$;
- MT11: $x/\sqrt{1+x} \le \ln(1+x) + \varepsilon$ when $x \in [-1/3, 0]$;

- MT12: $\ln((1+x)/x) \le (12x^2+12x+1)/(12x^3+18x^2+6x)$ when $x \in [1/3, 10]$;
- MT13: $\ln((1+x)/x) \le 1/\sqrt{x^2+x}$ when $x \in [1/3, 10]$;
- MT14: $\exp(x - x^2) \le 1 + x + \varepsilon$ when $x \in [0, 1]$;
- MT15: $\exp(-x/(1-x)) \le 1 - x + \varepsilon$ when $x \in [-10, 1/2]$;
- MT16: $|\sin x| \le 6/5 \cdot |x| + \varepsilon$ when $x \in [-1, 1]$;
- MT17: $1 - 2x < \cos(\pi \cdot x)$ when $x \in [\varepsilon, 100/201]$;
- MT18: $0 \le \cos x - 1 + x^2/2 + \varepsilon$ when $x \in [-10, 10]$;
- MT19: $8\sqrt{3} \cdot x/(3\sqrt{3} + \sqrt{75+80x^2}) \le \arctan x + \varepsilon$ when $x \in [0, 10]$;
- MT20: $1 < (x + 1/x) \cdot \arctan x$ when $x \in [\varepsilon, 10]$;
- MT21: $3x/(1+2\sqrt{1+x^2}) \le \arctan x + \varepsilon$ when $x \in [0, 10]$;
- MT22: $\cos x \le \sin x/x$ 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 $x \in [0, 2]$.

4.3.2 Experimental results

For the experiments conducted within the PVS proof assistant, we have been using the `ProofLite` package by César Muñoz to install the following scripts in batch mode:

- Using the `metit` strategy of PVS 6.0 (based on MetiTarski 2.2):

```
%|- * : PROOF
%|- (metit :timeout 180)
%|- QED
```

- Using the `interval` strategy of PVS 6.0:

```
%|- * : PROOF
%|- (then (skosimp*)(split)(skosimp*)
%|-   (apply (interval) :time? t :timeout 180))
%|- QED
```

- Using the Bernstein strategy of PVS 5.0:

```
%|- * : PROOF
%|- (then (skosimp*)
%|-   (apply (bernstein) :time? t :timeout 180))
%|- QED
```

As Alexey Solovyev's `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. For all the goals that have been discharged with this HOL Light decision procedure, the base of arithmetic was 200 and precision was 5 (except for inequality MT23, which was proved with precision 6).

Table 1 shows the CPU time for proving our selection of problems. For these benchmarks, a laptop with an Intel Core i5-4200M CPU clocked at 2.50 GHz has been used, along with Sollya 4.1, OCaml 4.01.0, and Coq 8.4pl5. The version 6.0.8 of the NASA PVS libraries has been used with PVS 6.0 (except for the Bernstein strategy, which is only available for PVS 5.0 at the time we are writing this paper). Finally, we have been using development versions of HOL Light[11] (rev. 196) as well as of `flyspeck/formal_ineqs`[12] (rev. 3660). 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,[13] 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 4.2, 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 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 performances, CoqInterval is much faster than `verify_ineq` on the most complicated examples, since it is not restricted to degree-2 polynomial 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`, which is rather slow.

On the tests extracted from MetiTarski, CoqInterval does not perform as fast as MetiTarski. The comparison is not that unfavorable though, when one keeps in mind that all the results are

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

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

[13]`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	1.04⋆	0.01	Failed	-	-	Failed	-	-	-
remez_sqrt	0.57	0.02	0.07	16.82⋆	Timeout	Failed	4.02⋆	-	-
abs_err_atan	0.54	0.01	0.09	Failed	Failed	Timeout	2.59⋆	-	-
rel_err_geodesic	3.51	2.43	Timeout	Timeout	Timeout	Failed	252.54⋆	-	-
harrison97	0.50	0.01	0.14	-	-	Failed	-	-	-
cos_cos_d2	0.86	0.06	Timeout	Timeout	Timeout	22.50	6.39⋆	-	-
cos_cos_d3	0.93	0.06	Timeout	Timeout	Timeout	53.30	6.97⋆	-	-
cos_cos_d4	1.04	0.07	Timeout	Timeout	Timeout	Timeout	9.86⋆	-	-
cos_cos_d5	1.64	0.07	Timeout	Timeout	Timeout	Timeout	17.41⋆	-	-
cos_cos_d6	1.78	0.07	Timeout	Timeout	Timeout	Timeout	23.19⋆	-	-
cos_cos_d7	2.53	0.08	Timeout	Timeout	Timeout	Timeout	46.57⋆	-	-
cos_cos_d8	3.23	0.09	Timeout	Timeout	Timeout	Timeout	97.09⋆	-	-
MT1	0.60	-	0.12	-	-	Failed	-	-	-
MT2	1.75	-	0.04	11.30⋆	Timeout	Failed	-	-	-
MT3	0.20	-	0.17	-	-	1.23	-	-	-
MT4	0.26	-	0.20	1.39⋆	21.70⋆	1.33	-	-	-
MT5	0.13⋆	-	0.06	-	-	1.37	-	-	-
MT6	0.18⋆	-	0.08	-	-	1.40	-	-	-
MT7	0.05	-	0.03	-	-	2.25	-	-	-
MT8	0.38	-	0.18	-	-	Timeout	-	-	-
MT9	0.58	-	0.48	-	-	Timeout	-	-	-
MT10	0.21	-	0.05	1.03	16.70	Failed	-	-	-
MT11	0.11	-	0.22	0.42	7.70	1.84	-	-	-
MT12	3.21	-	0.07	Timeout	Timeout	Timeout	-	-	-
MT13	1.10	-	0.09	12.83	153.71	Failed	-	-	-
MT14	0.08	-	0.06	-	-	0.93	-	-	-
MT15	0.17	-	0.08	-	-	1.08	-	-	-
MT16	0.14	-	0.04	0.66⋆	9.42⋆	3.49	0.63⋆	-	-
MT17	0.12	-	0.03	0.22	4.55	1.37	0.25	-	-
MT18	0.19	-	0.01	0.22	2.78	0.73	0.84	-	-
MT19	0.58	-	Failed	6.03	84.13	Failed	2.11	-	-
MT20	3.48	-	0.04	2.83	50.41	Timeout	17.19	-	-
MT21	0.36	-	0.43	4.12	58.55	Failed	1.53	-	-
MT22	0.78	-	0.08	Timeout	Timeout	Failed	126.19	-	-
MT23	1.31	-	0.11	Failed	Failed	Failed	Failed	-	-
MT24	0.12	-	0.40	0.19	2.75	Failed	0.27	-	-
MT25	0.33	-	0.14	-	-	1.95	-	-	-
RD	0.29	-	0.02	2.12	75.01	1.87	0.54	3.70	Timeout
adaptiveLV	0.19	-	0.04	0.26	3.63	1.09	1.41	4.44	4.22
butcher	0.48	-	0.05	0.82	12.41	21.87	2.45	20.10	Timeout
magnetism	0.21	-	0.04	1.63	23.28	Timeout	347.25	Timeout	0.25

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.

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 multivariate problems show that CoqInterval can handle them quite easily, as long as only one variable really matters. It also shows that some multivariate solvers would benefit from heuristics for reducing multivariate problems to univariate ones. In particular, magnetism is actually a straightforward problem, once one has noticed that only x_1 matters.

5 Conclusion

5.1 Summary

This paper has presented a tactic for the Coq proof assistant that is designed to automatically and formally verify numerical bounds on univariate 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 univariate 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. (In a sense, that is already what the `native_compute` tactic performs, except that the resulting native code is then transparently invoked by Coq.)

Let us summarize how our approach compares to the other tools. First, it only handles mostly univariate expressions, as does Sollya [7], while all the other tools are meant to support truly multivariate expressions. It supports some elementary functions, contrarily to `REAL_SOS` for HOL Light [12] and `bernstein` for PVS [21]. 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 [23], while all the other tools use rational arithmetic.

Finally, if we exclude the tools that only supports polynomial systems, the main difference between the remaining tools is whether they focus on univariate or multivariate expressions. Indeed, in the multivariate case, Taylor-like approximations grow up exponentially, while they grow only linearly in the univariate case. This partly explains why `verify_ineq` (HOL Light) and NLCertify handle only degree-2 polynomials as approximations. Unfortunately, this low degree is not sufficient to tackle the kind of proofs required when verifying mathematical libraries. Tools dedicated to univariate expressions do not suffer from this limitation.

5.2 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 kernel 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 $\triangledown(u+v) \le u+v \le \triangle(u+v)$. 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 performances 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.

References

[1] Behzad Akbarpour and Lawrence C. Paulson. 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] Xavier Allamigeon, Stéphane Gaubert, Victor Magron, and Benjamin Werner. Certification of bounds of non-linear functions: The templates method. In Jacques Carette, David Aspinall, Christoph Lange, Petr Sojka, and Wolfgang Windsteiger, editors, *Intelligent Computer Mathematics - MKM, Calculemus, DML, and Systems and Projects*, volume 7961 of *Lecture Notes in Computer Science*, pages 51–65, 2013. doi:10.1007/978-3-642-39320-4_4.

[3] Sylvie Boldo and Guillaume Melquiond. Flocq: A unified library for proving floating-point algorithms in Coq. In Elisardo Antelo, David Hough, and Paolo Ienne, editors, *Proceedings of the 20th IEEE Symposium on Computer Arithmetic*, pages 243–252, Tübingen, Germany, 2011. doi:10.1109/ARITH.2011.40.

[4] Nicolas Brisebarre, Mioara Joldeş, Érik Martin-Dorel, Micaela Mayero, Jean-Michel Muller, Ioana Paşca, Laurence Rideau, and Laurent Théry. Rigorous polynomial approximation using Taylor models in Coq. In Alwyn Goodloe and Suzette Person, editors, *Proceedings of 4th International Symposium on NASA Formal Methods*, volume 7226 of *Lecture Notes in Computer Science*, pages 85–99, Norfolk, Virginia, 2012. Springer. doi:10.1007/978-3-642-28891-3_9.

[5] Martine Ceberio and Laurent Granvilliers. Horner's rule for interval evaluation revisited. *Computing*, 69(1):51–81, 2002. doi:10.1007/s00607-002-1448-y.

[6] Sylvain Chevillard, John Harrison, Mioara Joldeş, and Christoph Lauter. 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] Sylvain Chevillard, Mioara Joldeş, and Christoph Lauter. Sollya: An environment for the development of numerical codes. In Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama, editors, *Proceedings of the 3rd International Congress on Mathematical Software*, volume 6327 of *Lecture Notes in Computer Science*, pages 28–31, Heidelberg, Germany, 2010.

[8] Marc Daumas, David Lester, and César Muñoz. Verified real number calculations: A library for interval arithmetic. *IEEE Transactions on Computers*, 58(2):226–237, 2009.

[9] Marc Daumas, Guillaume Melquiond, and César Muñoz. Guaranteed proofs using interval arithmetic. In Paolo Montuschi and Eric Schwarz, editors, *Proceedings of the 17th IEEE Symposium on Computer Arithmetic*, pages 188–195, Cape Cod, MA, USA, 2005. doi:10.1109/ARITH.2005.25.

[10] William Denman and César Muñoz. Automated real proving in PVS via MetiTarski. In Cliff B. Jones, Pekka Pihlajasaari, and Jun Sun, editors, *FM*, volume 8442 of *Lecture Notes in Computer Science*, pages 194–199. Springer, 2014. doi:10.1007/978-3-319-06410-9_14.

[11] John Harrison. Verifying the accuracy of polynomial approximations in HOL. In Elsa L. Gunter and Amy P. Felty, editors, *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics*, volume 1275 of *Lecture Notes in Computer Science*, pages 137–152, Murray Hill, NJ, USA, 1997. doi:10.1007/BFb0028391.

[12] John Harrison. Verifying nonlinear real formulas via sums of squares. In Klaus Schneider and Jens Brandt, editors, *Proceedings of the 20th International Conference on Theorem Proving in Higher Order Logics*, volume 4732 of *Lecture Notes in Computer Science*, pages 102–118, Kaiserslautern, Germany, 2007.

[13] Mioara Joldeş. *Rigorous Polynomial Approximations and Applications*. PhD thesis, ENS de Lyon, France, 2011. URL: http://tel.archives-ouvertes.fr/tel-00657843/en/.

[14] K. Makino. *Rigorous Analysis of Nonlinear Motion in Particle Accelerators*. PhD thesis, Michigan State University, East Lansing, Michigan, USA, 1998.

[15] K. Makino and M. Berz. Taylor models and other validated functional inclusion methods. *International Journal of Pure and Applied Mathematics*, 4(4):379–456, 2003. URL: http://bt.pa.msu.edu/pub/papers/TMIJPAM03/TMIJPAM03.pdf.

[16] Érik Martin-Dorel, Micaela Mayero, Ioana Paşca, Laurence Rideau, and Laurent Théry. Certified, efficient and sharp univariate Taylor models in Coq. In *SYNASC 2013*, pages 193–200, Timişoara, Romania, 2013. IEEE. URL: http://hal.inria.fr/hal-00845791v2/en/, doi:10.1109/SYNASC.2013.33.

[17] Guillaume Melquiond. Floating-point arithmetic in the Coq system. In *Proceedings of the 8th Conference on Real Numbers and Computers*, pages 93–102, Santiago de Compostela, Spain, 2008.

[18] Guillaume Melquiond. Proving bounds on real-valued functions with computations. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Proceedings of the 4th International Joint Conference on Automated Reasoning*, volume 5195 of *Lecture Notes in Artificial Intelligence*, pages 2–17, Sydney, Australia, 2008. doi:10.1007/978-3-540-71070-7_2.

[19] Guillaume Melquiond. Floating-point arithmetic in the Coq system. *Information and Computation*, 216:14–23, 2012. doi:10.1016/j.ic.2011.09.005.

[20] R. E. Moore. *Interval Analysis*. Prentice-Hall, 1966.

[21] César Muñoz and Anthony Narkawicz. 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.

[22] Anthony Narkawicz and César Muñoz. A formally verified generic branching algorithm for global optimization. In Ernie Cohen and Andrey Rybalchenko, editors, *Proceedings of the 5th International Conference on Verified Software: Theories, Tools, Experiments*, volume 8164 of *Lecture Notes in Computer Science*, pages 326–343, Menlo Park, CA, USA, 2013. doi:10.1007/978-3-642-54108-7_17.

[23] Alexey Solovyev and Thomas C. Hales. Formal verification of nonlinear inequalities with Taylor interval approximations. In Guillaume Brat, Neha Rungta, and Arnaud Venet, editors, *Proceedings of the 5th International Symposium on NASA Formal Methods*, volume 7871 of *Lecture Notes in Computer Science*, pages 383–397, Moffett Field, CA, USA, 2013. doi:10.1007/978-3-642-38088-4_26.

[24] Ping Tak Peter Tang. 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.

[25] Roland Zumkeller. *Global Optimization in Type Theory*. PhD thesis, École polytechnique, France, 2008. URL: http://alacave.net/~roland/FormalGlobalOpt.pdf.

Acknowledgements

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.

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

CNRS
INPT
UPS
UT1

Institut de Recherche en Informatique de Toulouse

ACADIE

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 traditional tools. In fact, the inherent difficulty of computing such bounds often mandate 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 kernel 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.

UPS-IRIT, 118 route de Narbonne, 31062 Toulouse CEDEX 9

+33 (0) 561 55 67 65 info@irit.fr www.irit.fr