Proving Bounds on Real-Valued Functions with Computations

Guillaume Melquiond

INRIA–Microsoft Research joint center,
Parc Orsay Université, F-91893 Orsay Cedex, France,
guillaume.melquiond@inria.fr

Abstract. Interval-based methods are commonly used for computing numerical bounds on expressions and proving inequalities on real numbers. Yet they are hardly used in proof assistants, as the large amount of numerical computations they require keeps them out of reach from deductive proof processes. However, evaluating programs inside proofs is an efficient way for reducing the size of proof terms while performing numerous computations. This work shows how programs combining automatic differentiation with floating-point and interval arithmetic can be used as efficient yet certified solvers. They have been implemented in a library for the Coq proof system. This library provides tactics for proving inequalities on real-valued expressions.

1 Introduction

In traditional formalisms, proofs are usually composed of deductive steps. Each of these steps is the instantiation of a logical rule or a theorem. While this may be well-adapted for manipulating logic expressions, it can quickly lead to inefficiencies when explicit computations are needed. Let us consider the example of natural numbers constructed from 0 and a successor function S. For example, the number 3 is represented by $S(S(S(0)))$. If one needs to prove that 3×3 is equal to 9, one can apply Peano's axioms, e.g. $a \times S(b) = a \times b + a$ and $a + S(b) = S(a + b)$, until 3×3 has been completely transformed into 9. The first steps of the proof are: $3 \times 3 = 3 \times 2 + 3 = (3 \times 1 + 3) + 3 = \ldots$ This proof contains about 15 instantiations of various Peano's axioms. Due to the high number of deductive steps, this approach hardly scales to more complicated expressions, even if more efficient representations of integers were to be used, e.g. radix-2 numbers.

While numerical computations are made cumbersome by a deductive approach, they can nonetheless be used in formal proofs. Indeed, type-theoretic checkers usually come with a concept of programs which can be expressed in the same language than the proof terms. Moreover, the formalism of these checkers assumes that replacing an expression $f(x)$ of a functional application by the result of the corresponding evaluation does not modify the truth of a statement. As a consequence, one can write computable recursive functions for addition `add`

and multiplication `mul` of natural numbers. For instance, in a ML-like language, `mul` can be defined as:

```
let rec mul x = function
  | 0 -> 0
  | S y -> add (mul x y) x
```

This function is extensionally equal to Peano's multiplication. More precisely, the following theorem can be proved by recursive reasoning on y:

$$\texttt{mul\_spec} : \forall x\ \forall y \quad x \times y = \texttt{mul}\ x\ y.$$

Therefore, in order to prove the statement $3 \times 3 = 9$, the first step of the proof is an application of `mul_spec` in order to rewrite 3×3 as `mul` 3 3. So one has now to prove that `mul` 3 3 is equal to 9. This is achieved by simply evaluating the function. So the proof contains only one deductive step: the use of `mul_spec`. With this computational approach, the number of deductive steps depends on the number of arithmetic operators only; It does not depend on the size of the integers. As a matter of fact, one can go even further so that the number of deductive steps is constant, irrespectively of the complexity of the expressions. This is the approach presented in this article.

In the Coq proof assistant[1], the ability to use programs inside proofs is provided by the convertibility rule: Two convertible well-formed types have the same inhabitants. In other words, if p is a proof of a proposition A, then p is also a proof of any proposition B such that the type B is convertible to the type A. Terms – types are terms – are convertible if they have the same normal form with respect to β-reduction (and a few other Coq-related reductions). In particular, since `mul` 3 3 evaluates to 9 by β-reduction, the types/propositions `mul` $3\ 3 = 9$ and $9 = 9$ are convertible, so they have exactly the same inhabitants/proofs.

More generally, an unproven proposition $P(x, y, \ldots)$ is transformed into a sufficient proposition $f_P(x, y, \ldots) = \mathit{true}$ — the difficulty is in designing an efficient f_P function and proving it evaluates to *true* only when P holds. The proof system then checks if this boolean equality is convertible to $\mathit{true} = \mathit{true}$. If it is, then it has a trivial proof from which a proof of P can be deduced. Going from P to f_P is a single deductive step, so most of the verification time will be spent in evaluating f_P. Fortunately, the convertibility rule happens to be implemented quite efficiently in Coq [1,2], so it becomes possible to automatically prove some propositions on real numbers by simply evaluating programs. An example of such a proposition is the following one, where x and y are universally-quantified real numbers:

$$\frac{3}{2} \leq x \leq 2 \Rightarrow 1 \leq y \leq \frac{33}{32} \Rightarrow \left| \sqrt{1 + \frac{x}{\sqrt{x+y}}} - \frac{144}{1000} \times x - \frac{118}{100} \right| \leq \frac{71}{32768}$$

In order to prove this logical proposition with existing formal methods, one can first turn it into an equivalent system of several polynomial inequalities. Then a resolution procedure, e.g. based on cylindrical algebraic decomposition [3] or

[1] `http://coq.inria.fr/`

on the *Nullstellensatz* theorem [4], will help a proof checker to conclude automatically. When the proposition involves elementary functions (e.g. cos, arctan, exp) in addition to algebraic expressions, the problem becomes undecidable. Some inequalities can still be proved by first replacing elementary functions by polynomial approximations [5,6]. A resolution procedure for polynomial systems can then complete the formal proof.

A different approach is based on interval arithmetic and numerical computations. The process inductively encloses sub-expressions with numbers and propagates these bounds until the range of all the expressions is known [7]. Naive interval arithmetic, however, suffer from a loss of correlation between the multiple occurrences of a variable. In order to avoid this issue, the problems can be split into several smaller problems or higher-order enclosures can be used instead [8,9].

This paper presents an implementation of this approach for Coq. It will focus on the aspects of automatic proof and efficiency. Section 2 describes the few concepts needed for turning numerical computations and approximations into a formal tool. Section 3 describes the particularities of the datatypes and programs used for these computations. Section 4 finally brings those components together in order to provide fully-automatized "tactics" (the tools available to the user of Coq).

2 Mathematical foundations

While both terms "computations" and "real numbers" have been used in the introduction, this work does not involve computational real numbers, e.g. sequences of converging rational numbers. As a matter of fact, it is based on the standard Coq theory of real numbers, which is a pure axiomatization with no computational content.

Section 2.1 presents the extension of real numbers defined for this work. This extension is needed in order to get a simpler definition of differentiation. Section 2.2 describes the interval arithmetic formalized in order to bound expressions. Section 2.3 then shows how differentiation and intervals are combined in order to tighten the computed bounds.

2.1 Extended real numbers

The standard theory of Coq describes real numbers as a complete Archimedean field. Since functions of type $\mathbb{R} \to \mathbb{R}$ are total, this formalism makes it a bit troublesome to deal with mathematical partial functions. For instance, $x \mapsto \frac{x}{x}$ is 1 for any real $x \neq 0$. For $x = 0$, the function is defined and its value is 0 in Coq. Indeed, $0 \cdot 0^{-1} = 0$, since 0^{-1} is the result of a total function, hence real. Similarly, one can prove, that the derivative of this function is always zero. So we have a function that seems both constant and discontinuous at $x = 0$.

Fortunately, this does not induce a contradiction, since it is impossible to prove that the derivative (the variation limit) of $x \mapsto \frac{x}{x}$ was really 0 at point 0.

The downside is that, every time one wants to use the value of the derivative function, one has to prove that the original function is actually derivable. This requires to carry lots of proof terms around, which will prevent a purely computational approach.

So an element $\perp$ is added to the formalism in order to represent the "result" of a function outside its definition domain. In the Coq development, this additional element is called *NaN* (Not-a-Number) as it shares the same properties as the *NaN* value from the IEEE-754 standard on floating-point arithmetic [10]. In particular, *NaN* is an absorbing element for all the arithmetic operators on the extended set $\overline{\mathbb{R}} = \mathbb{R} \cup \{\perp\}$. That way, whenever an intermediate result is undefined, a $\perp$ value is propagated till the final result of the computation.

Functions of $\overline{\mathbb{R}} \to \overline{\mathbb{R}}$ can then be created by composing these new arithmetic operators. In order to benefit from the common theorems of real analysis, the functions are brought back into $\mathbb{R} \to \mathbb{R}$ by applying a projection operator:

$$\mathit{proj}_a : f \in (\overline{\mathbb{R}} \to \overline{\mathbb{R}}) \mapsto \left(x \in \mathbb{R} \mapsto \begin{cases} a & \text{if } f(x) = \perp \\ f(x) & \text{otherwise} \end{cases} \right) \in (\mathbb{R} \to \mathbb{R})$$

Then, an extended function f is defined as continuous at a point $x \neq \perp$ if $f(x) \neq \perp$ and if all the projections $\mathit{proj}_a(f)$ are continuous at point x. Similarly, f is defined as derivable at $x \neq \perp$ if $f(x) \neq \perp$ and if all the projections of f have the same derivative $d_f(x)$ at point x. A function f' is then a derivative of f if $f'(x) = d_f(x)$ whenever $f'(x) \neq \perp$.

From these definitions and standard analysis, it is easy to formally prove some rules for building derivatives. For example, if f' and g' are some derivatives of f and g, then the extended function $(f' \times g - g' \times f)/g^2$ is a derivative of f/g. This is true even if g evaluates to 0 at a point x, since the derivative would then evaluate to $\perp$ at this point.

As a consequence, if the derivative f' of an expression f does not evaluate to $\perp$ on the points of a connected set of $\mathbb{R}$, then f is defined, continuous, and derivable on all the points of this set, when evaluated on real numbers. The important point is that the extended derivative f' can be automatically built from an induction on the structure of the function f, without having to prove that f is derivable on the whole domain.

2.2 Interval arithmetic

All the *intervals* (closed connected subsets) of $\mathbb{R}$ can be represented as pairs of extended numbers[2]:

$$\begin{array}{l} \langle \perp , \perp \rangle \mapsto \mathbb{R} \\ \langle \perp , u \rangle \mapsto \{x \in \mathbb{R} \mid x \leq u\} \\ \langle l , \perp \rangle \mapsto \{x \in \mathbb{R} \mid l \leq x\} \\ \langle l , u \rangle \mapsto \{x \in \mathbb{R} \mid l \leq x \leq u\} \end{array}$$

[2] The pair $\langle \perp, \perp \rangle$ traditionally represents the empty set. This is the opposite here, as $\langle \perp, \perp \rangle$ means $\mathbb{R}$. This is a consequence of having no infinities in the formalization: $\perp$ on the left means $-\infty$ while $\perp$ on the right means $+\infty$. The empty set is represented by any pair $\langle l, u \rangle$ with $l > u$.

This set $\mathbb{I}$ of intervals is extended with a *NaI* (Not-an-Interval): $\bar{\mathbb{I}} = \mathbb{I} \cup \{\perp_I\}$. This new interval $\perp_I$ represents the set $\overline{\mathbb{R}}$. In particular, this is the only interval that contains $\perp$. As a consequence, if the value of an expression on $\overline{\mathbb{R}}$ is contained in an interval $\langle l, u\rangle$, then this value is actually a real number. This implies that the expression is well-formed on the real numbers and that it is bounded by l and u.

Interval extensions and operators A function $F \in \bar{\mathbb{I}} \to \bar{\mathbb{I}}$ is defined as an interval extension of a function $f \in \overline{\mathbb{R}} \to \overline{\mathbb{R}}$, if

$$\forall X \in \mathbb{I}, \forall x \in \overline{\mathbb{R}}, \quad x \in X \Rightarrow f(x) \in F(X).$$

This definition can be adapted to non-unary functions too. An immediate property is: The result of $F(X)$ is $\perp_I$ if there exists some $x \in X$ such that $f(x) = \perp$. Another property is the compatibility of interval extension with function composition: If F and G are extensions of f and g respectively, then $F \circ G$ is an extension of $f \circ g$.

Again, an interval extension will be computed by an induction on the structure of an expression. This requires interval extensions of the arithmetic operators and elementary functions. For instance, addition and subtraction are defined as propagating $\perp_I$ and verifying the following rules:

$$\langle l_1, u_1\rangle + \langle l_2, u_2\rangle = \langle l_1 + l_2, u_1 + u_2\rangle$$
$$\langle l_1, u_1\rangle - \langle l_2, u_2\rangle = \langle l_1 - u_2, u_1 - l_2\rangle$$

Except for the particular case of $\perp$ meaning an infinite bound, this is traditional interval arithmetic [11,12]. So extending the other arithmetic operators does not cause much difficulty. For instance, if l_1 is negative and if both u_1 and l_2 are positive, the result of the division $\langle l_1, u_1\rangle / \langle l_2, u_2\rangle$ is $\langle l_1/l_2, u_1/l_2\rangle$.

Notice that this division algorithm depends on the ability to decide the signs of the bounds. More generally, one has to compare bounds when doing interval arithmetic. Section 3.1 solves it by restricting the bounds to a subset of $\overline{\mathbb{R}}$.

2.3 Bounds on real-valued functions

Properties described in previous sections can now be mixed together for the purpose of bounding real-valued functions. Let us consider a function f of $\overline{\mathbb{R}} \to \overline{\mathbb{R}}$, for which we want to compute an interval enclosure Y such that $f(x) \in Y$ for any x in the interval $X \neq \perp_I$. Assuming we have an interval extension F of f, then the interval $F(X)$ is an acceptable answer.

Unfortunately, if X appears several times in the unfolded expression of $F(X)$, loss of correlation occurs: The wider the interval X is, the poorer the bounds obtained from $F(X)$ are. The usual example is $f(x) = x - x$. It always evaluates to 0, but its trivial extension $F(X) = X - X$ evaluates to $\langle 0, 0\rangle$ only when X is a singleton.

Monotone functions Let us suppose now that we also have an interval extension F' of a derivative f' of f. By definitions of interval extension and derivability, if $F'(X)$ is not $\perp_I$, then f is continuous and derivable at each point of X.

Moreover, if $F'(X)$ does not contain any negative value, then f is an increasing function on X. If X has real bounds l and u, then an enclosure of f on X is the convex hull $F(\langle l, l\rangle) \vee F(\langle u, u\rangle)$. As the interval $\langle l, l\rangle$ contains one single value when $l \neq \perp$, the interval $F(\langle l, l\rangle)$ should not be much bigger than the set $\{f(l)\}$ for any F that is a reasonable interval extension of f. As a consequence, $F(\langle l, l\rangle) \vee F(\langle u, u\rangle)$ should be a tight enclosure of f on X. The result is identical if $F'(X)$ does not contain any positive values.

First-order interval evaluation When $F'(X)$ contains both positive and negative values, there are no methods giving sharp enclosure of f. Yet $F'(X)$ can still be used in order to find a better enclosure than $F(x)$. Indeed, variations of f on X are bounded:

$$\forall a, b \in X \ \exists c \in X \quad f(b) = f(a) + (b - a) \cdot f'(c).$$

Once translated to intervals, this gives the following theorem:

$$\forall a, b \in X \quad f(b) \in F(\langle a, a\rangle) + (X - \langle a, a\rangle) \cdot F'(X).$$

As $F'(X)$ may contain even more occurrences of X than $F(X)$ did, the loss of correlation may be worse when computing an enclosure of f' than an enclosure of f. From a numerical point of view, however, we have more leeway. Indeed, the multiplication by $X - \langle a, a\rangle$, which is an interval containing only "small" values around zero, will mitigate the loss of correlation. This approach to proving bounds on expressions can be generalized to higher-order derivatives by using Taylor models [9].

3 Computational datatypes

Proving propositions by computations requires adapted data types. This work relies on floating-point arithmetic (Section 3.1) for numerical computations and on straight-line programs (Section 3.2) for representing and evaluating expressions.

3.1 Floating-point arithmetic

Since interval arithmetic suffers from loss of correlation, the bounds are usually not sharp, so they do not need to be represented with exact real numbers. As a consequence, an interval extension does not need to return the "best" bounds. Simpler ones can be used instead. An interesting subset of $\overline{\mathbb{R}}$ is the set $\overline{\mathbb{F}} = \mathbb{F} \cup \{\perp\}$ of radix-2 floating-point numbers. Such a number is a rational number that can be written as $m \cdot 2^e$ with m and e integers.

Rounding Let us consider the non-$\perp$ quotient $\frac{u}{v}$ of two floating-point numbers. This quotient is often impossible to represent as a floating-point number. If this value is meant to be the lower bound of an interval quotient, we can chose any floating-point number $m \cdot 2^e$ less than the ideal quotient. Among these numbers, we can restrict ourselves to numbers with a mantissa m represented with less than p bits (in other words, $|m| < 2^p$). There is an infinity of such numbers. But one of them is bigger than all the others. This is what the IEEE-754 standard calls the result of u/v rounded toward $-\infty$ at precision p.

Computing at fixed precision ensures that the computing time is linear in the number of arithmetic operations. Let us consider the computation of $\left(\frac{5}{7}\right)^{2^n}$ with n consecutive squaring. With rational arithmetic, the time complexity is then $O(n^3)$, as the size of the numbers double at each step. With floating-point arithmetic at fixed precision, the time complexity is just $O(n)$. The result is no longer exact, but interval arithmetic still works properly.

There have been at least two prior formalizations of floating-point arithmetic in Coq. The first one [13,14] defines rounded results with relations, so the value w would be expressed as satisfying the proposition:

$$w \leq \frac{u}{v} \quad \wedge \quad \forall m, e \in \mathbb{Z}, \quad |m| < \beta^p \Rightarrow m \cdot \beta^e \leq \frac{u}{v} \Rightarrow m \cdot \beta^e \leq w$$

While useful and sufficient for proving theorems on floating-point algorithms, such a relation does not provide any computational content, so it cannot be used for performing numerical computations. The second formalization [15] has introduced effective floating-point operators, but only for addition and multiplication. The other basic operators are evaluated by an external oracle. The results can then be checked by the system with multiplications only. Elementary functions, however, cannot be reached with such an approach.

Implementation In order to have effective floating-point operators that can be evaluated entirely inside Coq, this work needed a new formalization of floating-point arithmetic. The resulting library implements multi-radix[3] multi-precision operators for the four IEEE-754 rounding directions[4].

This library supports the basic arithmetic operators ($+, -, \times, \div, \sqrt{\cdot}$) and some elementary functions (arctan, cos, sin, tan, for now). Floating-point precision is a user-settable parameter of the automatic tactics. Its setting is a trade-off: A high precision can help in proving some propositions, but it also slows down computations.

In order to speed up floating-point computations by a $\times 10$ factor[5], the tactics do not use the standard integers of Coq, which are represented by lists of bits. They specialize the floating-point library so that it uses integers represented as

[3] Numbers are $m \cdot \beta^e$ for any integer $\beta \geq 2$. For interval arithmetic, the radix hardly matters. So the automatic tactics chose an efficient radix: $\beta = 2$.

[4] Only rounding toward $-\infty$ and $+\infty$ are needed when performing interval arithmetic.

[5] This speed-up is lower than one could expect. This is explained by the proofs not needing high-precision computations, so the mantissa integers are relatively small.

binary trees with leaves being radix-2^{31} digits [1]. The arithmetic on these leaves is then delegated by Coq to the computer processor [16].

3.2 Straight-line programs

Until now, we have only performed interval computations. We have yet to prove properties on expressions. A prerequisite is the ability to actually represent these expressions. Indeed, as we want Coq functions to be able to evaluate expressions in various ways, e.g. for bounds or for derivatives, they need a data structure containing an abstract syntax tree of the expressions. More precisely, an expression will be represented as a straight-line program. This is a directed acyclic graph with an explicit topological ordering on the nodes which contain arithmetic operators.

For example, the expression $\sqrt{x} - y \cdot \sqrt{x}$ is encoded as a sequence of three tuples representing the following straight-line program. Each tuple represents an arithmetic operation whose operands are the results of some of previous operations represented by a relative index – index 0 was computed at the previous step, index 1 two steps ago, and so on. The input values x and y are assumed to have already been computed by pseudo-operations with results in v_1 and v_0.

$$\begin{array}{ll} v_2 : (\mathrm{sqrt}, 0) & \text{so } v_2 = \sqrt{v_{1-0}} = \sqrt{x} \\ v_3 : (\mathrm{mul}, 2, 0) & \text{so } v_3 = v_{2-2} \cdot v_{2-0} = y \cdot \sqrt{x} \\ v_4 : (\mathrm{sub}, 1, 0) & \text{so } v_4 = v_{3-1} - v_{3-0} = \sqrt{x} - y \cdot \sqrt{x} \end{array}$$

Notice that the square root occurs only once in the program. Representing expressions as straight-line programs makes it possible to factor common subexpressions. In particular, the computation of a given value (e.g. $\sqrt{x}$ here) will not be repeated several times during an evaluation.

The evaluation function is generic. It takes a list encoding the straight-line program, the type A of the inputs and outputs (e.g. $\overline{\mathbb{R}}$ or $\overline{\mathbb{I}}$), a record of functions implementing the operators (functions of type $A \to A$ and $A \to A \to A$), and a stack of inputs of type A. It then pushes on this stack the result of evaluating each operation stored in the list. It finally returns the stack containing the results of all the statements. Whenever an operation tries to access past the bottom of the evaluation stack, a default value of type A is used, e.g. 0 or $\bot$ or $\bot_I$.

When given various sets A and operators on A, the Coq function `eval` will produce, either an expression on real numbers corresponding to the straight-line program, or an interval enclosing the values of the expression, or an expression of the derivative of the expression, or bounds on this derivatives, etc. For instance, the derivative is bounded by evaluating the straight-line program on the set A of interval pairs – the first interval encloses the value of an expression, while the second one encloses the value of its derivative. The operators on A create these intervals with formulas related to automatic differentiation:

$$\begin{array}{lcl} \mathrm{plus} & \equiv & (X, X') \in A \mapsto (Y, Y') \in A \mapsto (X + Y, X' + Y') \\ \mathrm{mul} & \equiv & (X, X') \in A \mapsto (Y, Y') \in A \mapsto (X \times Y, X' \times Y + X \times Y') \end{array}$$

$$\begin{aligned} \tan \quad &\equiv \quad (X, X') \in A \mapsto (\tan X, X' \times (1 + \tan^2 X)) \\ &\ldots \end{aligned}$$

4 Automatic proofs

Now that we have the basic blocks, we can use the convertibility rule to build automatic tactics. First, convertibility helps transforming logical propositions into data structures on which programs can actually compute. Second, it gives a meaning to the subsequent numerical computations. The user does not have to worry about it though, since the tactics will take care of all the details.

4.1 Converting terms

Let us assume that the user wants to prove $\sqrt{x} - y \cdot \sqrt{x} \leq 9$ knowing some bounds on x and y. As explained in Section 2.2, interval arithmetic will be used to bound the expression $\sqrt{x} - y \cdot \sqrt{x}$. The upper bound can then be compared to 9 in order to check that the expression was indeed smaller than this value. In order to perform this evaluation, the functions need the straight-line program representing the expression.

Unfortunately, this straight-line program cannot be built within Coq's term language, as the syntax of the expressions on real numbers is not available at this level. So the list representing the program has to be provided by an oracle.

Three approaches are available. First, the user could perform the transformation by hand. This may be fine for small terms, but it quickly becomes cumbersome. Second, one could implement the transformation directly into the Ocaml code of Coq, hence creating a new version of the proof assistant. Several existing reflexive tactics actually depend on Ocaml helpers embedded inside Coq, so this is not an unusual approach. Third, one could use the tactic language embedded in Coq [17], so that the transformation runs on an unmodified Coq interpreter. This third way is the one chosen for this work.

A Coq tactic will therefore parse the expression and create the program described in Section 3.2. While the expression has type $\mathbb{R}$, this program is a list. But when evaluated with the axiomatized operations on real numbers, the result should be the original expression, if the tactic did not make any mistake. The tactic also transforms the real number 9 into the floating-point number $+9 \cdot 2^0$. So the tactic tells Coq that proving the following equality is equivalent to proving the original inequality.

$$(\texttt{eval}\ \mathbb{R}\ \texttt{[Sqrt 0, Mul 2 0, Sub 1 0]}\ \texttt{[y, x]}) \leq +9 \cdot 2^0$$

Coq does not trust the tactic, so it will check that this transformation is valid. Here, both inequalities are convertible, so they have the same proofs. Therefore, the proof system just has to evaluate the members of the new inequality, in order to verify that the transformation is valid. This transformation process is called reflexion [18]: An oracle produces a higher-level representation of the user

proposition, and the proof system has only to check that the evaluation of this better representation is convertible to the old one. This transformation does not involve any deductive steps; There is no rewriting steps with this approach, contrarily to the 3×3 example of the introduction.

4.2 Proving propositions

At this point, the proposition still speaks about real numbers, since convertibility cannot modify what the proposition is about. So we need the following theorem in order to get a proposition about extended real numbers, hence suitable for interval arithmetic with automatic differentiation.

$$\begin{array}{l} \forall prog\ \forall inputs\ \forall l, u \\ (\texttt{eval}\ \overline{\mathbb{R}}\ \ prog\ \ inputs) \in \langle l, u \rangle \Rightarrow \\ (\texttt{eval}\ \mathbb{R}\ \ prog\ \ inputs) \in \langle l, u \rangle \end{array}$$

Since the interval operators are interval extensions of the arithmetic operators on $\overline{\mathbb{R}}$, and since interval extension is compatible with function composition, we also have the following theorem.

$$\begin{array}{l} \forall prog\ \forall inputs\ \forall ranges \\ (\forall j \quad inputs_j \in ranges_j) \Rightarrow \\ (\texttt{eval}\ \overline{\mathbb{R}}\ \ prog\ \ inputs) \in (\texttt{eval}\ \overline{\mathbb{I}}\ \ prog\ \ ranges) \end{array}$$

Let us assume there are hypotheses in the context that state `x` $\in$ `X` and `y` $\in$ `Y`. By applying the two theorems above and the transitivity of interval inclusion, the tactic is left with the following proposition to prove:

$$(\texttt{eval}\ \overline{\mathbb{I}}\ \texttt{[Sqrt 0, Mul...] [Y, X]}) \subseteq \langle \bot, +9 \cdot 2^0 \rangle$$

While the interval evaluation could be performed in this proposition, the interval inclusion cannot be verified automatically yet. In order to force the proof system to compare the bounds, a last theorem is applied, so that the inclusion is transformed into a boolean equality:

$$\texttt{subset}\ (\texttt{eval}\ \overline{\mathbb{I}}\ \texttt{[Sqrt 0, Mul...] [Y, X]})\ \langle \bot, +9 \cdot 2^0 \rangle = true$$

The tactic then tells to Coq that this proposition is convertible to $true = true$. As comparisons between interval bounds are decidable, the `subset` function performs an effective computation, and so does `eval` on floating-point intervals. As a consequence, Coq is able to check the convertibility of these propositions by evaluating the left hand side of the equality. If the result of this evaluation is *true*, then the propositions have the same proofs. This conversion is numerically intensive and can take a long time, since it performs all the interval and floating-point computations. At this point, the tactic just has to remind Coq that equality is reflexive, so $true = true$ holds.

To summarize, this proof relies almost entirely on convertibility, except for a few deductive steps[6], which are instantiations of the following theorems:

[6] In addition, some optional steps may be performed to simplify the problem. For instance, $|expr| \leq u$ is first transformed into $-u \leq expr \leq u$.

1. If the result of a formula on extended reals is contained in a non-$\perp_I$ interval, then the formula on real numbers is well-formed and has the same bounds.
2. The interval evaluation of a given straight-line program is an interval extension of the evaluation of the same program on extended reals.
3. If `subset` A $B = true$, then any value contained in A is also contained in B.
4. Boolean equality is reflexive.

4.3 Bisection and refined evaluation

Actually, because of a loss of correlation, the left hand side evaluates to *false* on the example given in the introduction, so Coq complains that the proposition are not convertible. This is expected [8], and two methods experimented with the PVS proof assistant[7] can be reused here. The first method is the bisection: If the interval evaluation fails to return an interval small enough, split the input interval in two parts and perform the interval evaluation again on each of these parts. As the parts are smaller than the whole interval, the loss of correlation should be reduced, so the interval evaluation produces a sharper result [11].

In the PVS work, interval splitting is performed by applying a theorem for each sub-interval. Here we keep relying on programs in order to benefit from convertibility and reduce the number of deductive steps. The method relies on a `bisect` function recursively defined as:

$$
\begin{array}{l}
\texttt{bisect}\ n\ F\ \langle l, u\rangle\ \mathit{target} = \\
\quad \text{if } n = 0 \text{ then } \mathit{false} \\
\quad \text{else if } (\texttt{subset}\ F(\langle l, u\rangle)\ \mathit{target}) \text{ then } \mathit{true} \\
\quad \text{else let } m \text{ be the midpoint of } \langle l, u\rangle \text{ in} \\
\qquad (\texttt{bisect}\ (n-1)\ F\ \langle l, m\rangle\ \mathit{target}) \quad \&\& \\
\qquad (\texttt{bisect}\ (n-1)\ F\ \langle m, u\rangle\ \mathit{target})
\end{array}
$$

This function is meant to replace the `subset` call in the previous proof. Its associated theorem is a bit more complicated though, but its hypotheses are as easy to satisfy: If F is an interval extension of a function f and if `bisect` evaluates to *true*, then $f(x) \in \mathit{target}$ holds for any $x \in \langle l, u\rangle$. Once again, the complete proof contains only four deductive steps. Everything else is obtained through convertibility, with the evaluation of `bisect` being the numerically-intensive part of the proof.

As long as n is big enough and the floating-point computations are accurate enough, this method can solve most of the provable propositions of the form $\forall x \in \langle l, u\rangle,\ f(x) \in Y$ with f a straight-line program. The method is guaranteed[8] to succeed when l and u are finite and the sharpest enclosure of f on $\langle l, u\rangle$ is a subset of the interior of Y. The method can also be extended to multi-variate problems, by splitting interval boxes along each dimension iteratively.

[7] `http://pvs.csl.sri.com/`

[8] If there is $x \in \langle l, u\rangle$ such that $f(x)$ evaluates to $\perp$, Y has to be $\perp_I$. Otherwise, f is continuous on the compact set $\langle l, u\rangle$, hence it is also uniform continuous. This uniformity ensures that some suitable precision and some bisection depth exist.

The bigger the number of sub-intervals, the longer the proof will take. In order to keep this number small, the tactic calls `bisect` with an interval function F that performs first-order derivation, as described in Section 2.3.

5 Examples

The user-visible tactic is called `interval`. It can be used for proving inequalities involving real numbers. For instance, the following script is the Coq version of a PVS example proved by interval arithmetic [7]:

```
Goal
  let v := 250 * (514 / 1000) in
  3 * pi / 180 <= g / v * tan (35 * pi / 180).
Proof.
  apply Rminus_le. (* transform into a - b <= 0      *)
  interval.        (* prove by interval computations *)
Qed.
```

The strength of this work lies, however, in its ability to prove theorems on function approximations. Most mathematical functions are not available to developers, so they are usually replaced with approximations, e.g. truncated series. In order to certify that the programs are still valid after these transformations, one has to give and prove a bound on the error between the actual implementation and the ideal function. The absolute error is the difference between two close values (if the implementation is any good), which makes it hard to prove a tight bound on it — this is the $X - X$ issue of Section 2.3. The two examples below exercise the tactic on such ill-conditioned problems.

5.1 Remez' polynomial of the square root

Taylor models have been experimented in Coq [9] in order to formally prove some inequalities of Hales' proof[9] of Kepler's conjecture. Part of the difficulty with Taylor models lies in handling elementary functions. Indeed, one has to use polynomial approximations for this purpose. Usually, as the name implies, these polynomials are Taylor expansions, since their expansion remainder can be bounded by symbolic methods. Yet Taylor expansions are poor approximations, so high-degree polynomials are needed, which needlessly slow down the proof.

There are much better polynomial approximations, e.g. the ones obtained from Remez' algorithm. Unfortunately, the approximation error is no longer available to symbolic methods. One has to bound it numerically. The following proposition states the error bound between the square root function and its Remez approximation of degree 5 with rational coefficients of width $20+20$ bits, on the interval $0.5 \leq x \leq 2$:

$$\left| \left(\left(\frac{122}{7397} \times x - \frac{1733}{13547} \right) \times x + \cdots + \frac{227}{925} \right) - \sqrt{x} \right| \leq \frac{5}{65536}$$

[9] `http://code.google.com/p/flyspeck/`

Since Remez' algorithm returns the best polynomial approximation with real coefficients, checking the error bound is a numerically difficult problem. Yet it only takes a few seconds for the `interval` tactic to automatically prove it in Coq on a desktop computer. In comparison, the CAD algorithm (with fast integers too) needs more than ten minutes in Coq. For Hales' proof, one also needs the arctan function, which is in the scope of this tactic.

5.2 Relative error for an elementary function

The following example is taken from another PVS proof [8]. In order to certify a numerical code, the objective was to prove a bound on the relative error between the following function r_p and the degree-10 polynomial $\hat{r}_p$ that approximated it.

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

The relative error is defined as the quotient $\epsilon(\phi) = (r_p(\phi) - \hat{r}_p(\phi))/r_p(\phi)$ on the interval $0 \leq \phi \leq \frac{715}{512}$. Due to the loss of correlation, PVS interval strategies are unable to automatically prove the bound $23 \cdot 2^{-24}$ on this error. The problem could be split into smaller intervals, so that interval computations are able to prove the bound on each sub-interval. But the loss of correlation is extreme, so more than 10^6 sub-intervals are needed, which make it impossible to complete the whole proof in a reasonable amount of time. So first-order interval evaluation was performed (see Section 2.3) in order to bring the number of sub-intervals down to 10^5.

Unfortunately, several user actions are required with the PVS approach. First, the user has to prove beforehand that the relative error is well-formed (no division by zero, no square root of negative number, and so on). Second, the user has to prove the formulas involving the derivative. Third, an external oracle analyzes the proposition in order to choose good sub-intervals. It also searches the order at which the power series of tan have to be evaluated in order to provide results that are accurate enough. With these data, the oracle then generates 10^5 PVS scripts corresponding to all the sub-intervals, and one master script that states the theorem on the error bound. Fourth, the user dispatches all the generated scripts on the 48 cores of a parallel computer. A few hours later, proof verification is complete. Finally, PVS checks the master script: The bound $|\epsilon(\phi)| \leq 24 \cdot 2^{-23}$ is formally proved.

Thanks to the work described in this article, the situation is now a lot more satisfying in Coq. The following script proves the same theorem (`arp` is the approximation polynomial) in a few minutes on the single core of a desktop computer.

```
Goal
  forall phi, 0 <= phi <= max ->
  Rabs ((rp phi - arp phi) / rp phi) <= 23/16777216.
Proof.
  unfold rp, arp, umf2, a, f, max. intros.
  interval with (i_bisect_diff phi, i_nocheck).  (* Time:  4s *)
Qed.                                             (* Time: 96s *)
```

The user has to tell the tactic on which variable to perform a bisection followed by a first-order evaluation: `i_bisect_diff phi`. The user could also tell at which precision the floating-point computations are performed and how deep the bisection should explore. But the default parameters, `i_prec 30` and `i_depth 15`, are sufficient for this proof.

All the details of the proof are then handled by the tactic. It first parses the proposition and creates the corresponding straight-line program. It then performs the four deductive steps and Coq is left with a boolean equality. The evaluation of this Coq term by the system will first cause an interval function that encloses the derivative of the straight-line program to be built. It will then launch the execution of an interval bisection until the expression is bounded on all the sub-intervals.

When the proof is achieved (at `Qed` time), Coq checks that the lambda-term corresponding to the whole proof is correctly typed. In particular, it means that the numerically-intensive convertibility is checked a second time. The `i_nocheck` parameter avoids this redundant computation: The convertibility check is no longer done at `interval` time, but only at `Qed` time. So the tactic needs 4 seconds for parsing the expressions and preparing the computations, and then `Qed` needs 96 seconds to actually perform them.

Because there are no oracles, the Coq proof performs at least twice as many numerical computations[10] and with a higher precision than needed. Yet, the proof verification is tremendously faster in Coq than in PVS, although the approach is similar. This improvement is explained by the underlying arithmetic. In PVS, the interval bounds are rational numbers, so the intermediate computations quickly involve integers with thousands of digits. In Coq, thanks to the floating-point rounded arithmetic, the integers are at most 62-bit long. So the computation time does not explode with the size of the expressions.

6 Conclusion

Interval-based methods have been used for the last thirty years whenever a numerical problem (bounding an expression, finding all the zeros of a function, solving a system of differential equations, and so on) needed to be solved in an efficient and reliable way. But due to their numerically-intensive nature, they have been seldom used within formal proofs. With the advent of fast program evaluation in proof checkers, the situation is evolving [15,9].

This work improves on the existing formal approaches by relying on an efficient underlying arithmetic. Indeed, the computations are performed thanks to an effective formalization of floating-point arithmetic, instead of relying on an arithmetic on rational numbers. This brings the formal proofs closer to the non-formal approaches that are based on numerical computations with floating-point

[10] The bisection process can be seen as a binary tree. An inner node is an evaluation failure, which leads to an evaluation on two sub-intervals, its children. Leaves are successful interval evaluations, and they are the only nodes needed for the proof.

numbers, e.g. Hales' original proof. Another improvement lies in a careful extension of the real analysis: All the internal notions are designed so that theorems can be entirely handled with computations. The user does not have to deal with fastidious details anymore, e.g. proofs of derivability.

As this work deals with automatic proofs of numeric bounds on real-valued expressions suffering from correlations, it seems closely related to the Gappa system [15]. There are two important differences though. First, Gappa is an external oracle that produces a deductive proof that has been optimized, while this work is embedded and produces a straightforward and computational proof. Second, Gappa is specially designed for non-continuous expressions with basic arithmetic operators and a strong underlying structure, while this work focuses on (infinitely-) derivable expressions with elementary functions.

While designed for different kinds of expressions, these two approaches are complementary when performing formal certification of numerical applications. For instance, Section 5.2 was replacing an elementary function r_p with a polynomial $\hat{r}_p$. This polynomial is meant to be evaluated by a processor, but this evaluation $\tilde{r}_p$ will suffer from rounding errors. The implementation will be useful, only if the computed value $\tilde{r}_p(\phi)$ is close enough to the mathematical value $r_p(\phi)$. This certification is usually performed by separately bounding the distances between $\tilde{r}_p$ and $\hat{r}_p$, and between $\hat{r}_p$ and r_p. The first bound can be proved by Gappa but not by this work, since the expression is non-continuous. The second bound, however, can be proved by this work but not by Gappa, since the expression contains trigonometric terms and it has to be automatically differentiated for efficiency.

This work computes first-order derivatives of the expressions. This is usually sufficient for handling the correlations that appear when certifying numerical applications in most embedded systems, e.g. with a relative error of magnitude 2^{-25}. It will, however, fail to prove longer approximations which have a much higher accuracy. Therefore, tools that access higher-order derivatives through Taylor series [6,9] should perform much better in these latter cases. In case of high-dimension input domains, the approach presented in this article will also perform worse than multi-variate Bernstein polynomials [9].

Therefore, this work should not be seen as a panacea for proving inequalities on real-valued expressions without any user interaction. It is more of a proof-of-concept that shows how some numerical methods (floating-point arithmetic and interval arithmetic) can be combined with the convertibility rule in order to formally prove theorems. The generated proof terms contain almost no deductive steps and the tactic is nothing more than a parser, yet this approach is able to automatically prove arbitrarily-complicated inequalities.

The Coq development presented in this paper is available at
`http://www.msr-inria.inria.fr/~gmelquio/soft/coq-interval/`

References

1. Grégoire, B., Théry, L.: A purely functional library for modular arithmetic and its application to certifying large prime numbers. In Furbach, U., Shankar, N., eds.:

Proceedings of the 3rd International Joint Conference on Automated Reasoning. Volume 4130 of Lectures Notes in Artificial Intelligence., Seattle, WA, USA (2006) 423–437
2. Grégoire, B., Théry, L., Werner, B.: A computational approach to Pocklington certificates in type theory. In Hagiya, M., Wadler, P., eds.: Proceedings of the 8th International Symposium on Functional and Logic Programming. Volume 3945 of Lecture Notes in Computer Science., Fuji-Susono, Japan, Springer (2006) 97–113
3. Mahboubi, A.: Implementing the cylindrical algebraic decomposition within the Coq system. Mathematical Structure in Computer Sciences **17**(1) (2007)
4. Harrison, J.: Verifying nonlinear real formulas via sums of squares. In Schneider, K., Brandt, J., eds.: Proceedings of the 20th International Conference on Theorem Proving in Higher Order Logics. Volume 4732 of Lectures Notes in Computer Science., Kaiserslautern, Germany (2007) 102–118
5. Harrison, J.: Floating point verification in HOL light: The exponential function. In: Algebraic Methodology and Software Technology. (1997) 246–260
6. Akbarpour, B., Paulson, L.C.: Towards automatic proofs of inequalities involving elementary functions. In Cook, B., Sebastiani, R., eds.: PDPAR: Pragmatics of Decision Procedures in Automated Reasoning. (2006) 27–37
7. Muñoz, C., Lester, D.: Real number calculations and theorem proving. In Hurd, J., Melham, T., eds.: Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics. Volume 3603 of Lecture Notes in Computer Science., Oxford, UK (2005) 195–210
8. Daumas, M., Melquiond, G., Muñoz, C.: Guaranteed proofs using interval arithmetic. In Montuschi, P., Schwarz, E., eds.: Proceedings of the 17th IEEE Symposium on Computer Arithmetic, Cape Cod, MA, USA (2005) 188–195
9. Zumkeller, R.: Formal global optimisation with Taylor models. In Furbach, U., Shankar, N., eds.: Proceedings of the 3rd International Joint Conference on Automated Reasoning. Volume 4130 of Lectures Notes in Artificial Intelligence., Seattle, WA, USA (2006) 408–422
10. Stevenson, D., et al.: An American national standard: IEEE standard for binary floating point arithmetic. ACM SIGPLAN Notices **22**(2) (1987) 9–25
11. Moore, R.E.: Methods and Applications of Interval Analysis. SIAM (1979)
12. Jaulin, L., Kieffer, M., Didrit, O., Walter, E.: Applied Interval Analysis, with Examples in Parameter and State Estimation, Robust Control and Robotics. Springer-Verlag (2001)
13. Daumas, M., Rideau, L., Théry, L.: A generic library of floating-point numbers and its application to exact computing. In: Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics, Edinburgh, Scotland (2001) 169–184
14. Boldo, S.: Preuves formelles en arithmétiques à virgule flottante. PhD thesis, École Normale Supérieure de Lyon (2004)
15. Melquiond, G.: De l'arithmétique d'intervalles à la certification de programmes. PhD thesis, École Normale Supérieure de Lyon, Lyon, France (2006)
16. Spiwack, A.: Ajouter des entiers machine à Coq. Technical report (2006)
17. Delahaye, D.: A tactic language for the system Coq. In: Proceedings of the 7th International Conference on Logic for Programming and Automated Reasoning. Volume 1955 of Lecture Notes in Computer Science., Springer-Verlag (2000) 85–95
18. Boutin, S.: Using reflection to build efficient and certified decision procedures. In: Theoretical Aspects of Computer Software. (1997) 515–529