Proving bounds on real-valued functions with computations

Guillaume Melquiond

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 provide some efficient yet guaranteed solvers within the Coq proof system.

1 Introduction

In traditional formalisms, proofs are usually composed of deductive steps. Each of these steps is the instantiation of a specific 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, 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 and multiplication of natural numbers and prove that they satisfy Peano's axioms. Deductive steps only occur in these proofs of axiom satisfaction. No deductive steps are needed for proving equality of closed terms, e.g. $3 \times 3 = 9$, as evaluating 3×3 to 9 and replacing its occurrence in the equality is now sufficient. This new approach scales to complicated expressions, as the number of deductive steps no longer depends on the size of the natural numbers nor on the number of arithmetic operators.

In the Coq proof assistant[1], this 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 that the proposition A holds true, then p is also a proof that any B convertible to A also holds true. In particular, as 3×3 evaluates to 9, the proof of $9 = 9$ (reflexivity of equality) is also a proof of $3 \times 3 = 9$. As the convertibility rule can be quite efficient in Coq, programs can be designed for automatically proving propositions on expressions of 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:

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

In order to prove this logical proposition with an existing formal method, one can first transform it into an equivalent system of several polynomial inequalities. Then a resolution procedure, e.g. based on cylindrical algebraic decomposition [1] or on the *Nullstellensatz* theorem [2], will help a proof checker to conclude automatically. This paper presents a different approach, based on numerical computations, which can be extended to propositions containing non-algebraic expressions. Section 2 describes the few concepts needed for making numerical computations and approximations into a formal proof 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 a user-friendly proof tool which relies on logic programing only.

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.

2.1 Extended real numbers

The standard theory of Coq describes real numbers as a complete Archimedean field. As functions have to be total, this formalism may make it a bit troublesome to deal with partial functions such as $\frac{1}{x}$ and $\sqrt{x}$. In order to handle all

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

the arithmetic operators in an uniform way, an element $\perp$ is added 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 [3]. In particular, value $\perp$ is an absorbing element for any arithmetic operators on the extended set $\overline{\mathbb{R}} = \mathbb{R} \cup \{\perp\}$.

In order to benefit from the common theorems of real analysis, the functions defined on $\overline{\mathbb{R}}$ have to be brought back in $\mathbb{R} \to \mathbb{R}$. This can be done by using a projection operator parametrized by a real number a which will be used whenever the extended function would have returned $\perp$:

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

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 of f for any $a \in \mathbb{R}$ are continuous at the 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 x. A function f' is then considered as a derivative of f if $f'(x) = d_f(x)$ whenever $f'(x) \neq \perp$. Note that a given extended function may have several derivatives, according to this definition. For example, the function $x \in \overline{\mathbb{R}} \mapsto \perp$ is an acceptable derivative for any function of $\overline{\mathbb{R}} \to \overline{\mathbb{R}}$, though this function is quite useless.

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 $\frac{f' \times g - g' \times f}{g^2}$ is a derivative of $\frac{f}{g}$. Note that this theorem does not need to assume that g does not evaluate to zero. Indeed, if $g(x) = 0$, then the derivative evaluates to $\perp$, which is fine. This simplicity of the rules will make it easier to manipulate and compute derivatives later on.

2.2 Interval arithmetic

Rather than computing directly with these numbers, closed connected sets of real numbers are considered. They are represented by the set $\mathbb{I} = \overline{\mathbb{R}} \times \overline{\mathbb{R}}$ of all the pairs of extended numbers:

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

The last set is the empty set $\emptyset$ when the upper bound u is less than the lower bound l. Based on these interpretations, a "contains" $\ni$ relational operator is defined between intervals and reals. As for the real numbers, the set $\mathbb{I}$ is extended to a set $\overline{\mathbb{I}} = \mathbb{I} \cup \{\perp_I\}$ by adding an element $\perp_I$, which propagates along all the computations. The usual acronym for this element is *NaI* (Not-an-Interval). Interval $\perp_I$ contains any extended real number. In particular, it contains $\perp$, which is not contained in any interval of $\mathbb{I}$. Note that $\bot_I$ is not the pair $(\bot, \bot)$, which represents $\mathbb{R}$.

The set $\mathbb{I}$ is a lattice for the order $\preceq$ defined as: $(l_1, u_1) \preceq (l_2, u_2)$ if and only if $l_2 \leq l_1$ and $u_1 \leq u_2$. In the previous inequalities, a lower bound is replaced by $-\infty$ if it is $\bot$, and an upper bound by $+\infty$. This partial order $\preceq$ maps to the set inclusion partial order $\subseteq$ when considering non-empty closed connected sets of $\mathbb{R}$. The lattice operators "meet" $\wedge$ and "join" $\vee$ are respectively mapped to the set intersection and the convex hull of the set union. As usual, these operators are extended to $\overline{\mathbb{I}}$ by having them propagate $\bot_I$. For order $\preceq$, $\bot_I$ is defined as being strictly bigger than any interval of $\mathbb{I}$.

Interval extensions and operators A function $F \in \mathbb{I} \to \mathbb{I}$ 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. Some trivial interval extensions are $X \in \mathbb{I} \mapsto \bot_I$ and $X \in \mathbb{I} \mapsto (\bot, \bot)$. The first one extends any function of $\overline{\mathbb{R}} \to \overline{\mathbb{R}}$. The second one extends any total function of $\mathbb{R} \to \mathbb{R}$.

Note that the result of $F(X)$ has to be $\bot_I$ if there exists some $x \in X$ such that $f(x) = \bot$. As a corollary, if $F(X)$ is a pair of numbers, then f has to be the extension of a real function that is defined at each point of X. Another property is that interval extension is compatible with composition: If F and G are extension of f and g respectively, then $G \circ F$ is an extension of $f \circ g$.

Now arithmetic operators can be defined on intervals, such that they extend arithmetic operators on $\overline{\mathbb{R}}$. For example, addition and subtraction are defined as propagating $\bot_I$ and verifying the following rules:

$$\begin{aligned}(l_1, u_1) + (l_2, u_2) &= (l_1 + l_2, u_1 + u_2)\\ (l_1, u_1) - (l_2, u_2) &= (l_1 - u_2, u_1 - l_2)\end{aligned}$$

Except for the particular case of $\bot$ meaning an infinite bound, this is traditional interval arithmetic [4,5], and defining other arithmetic operators does not cause much difficulty. For example, the division is performed by looking at the sign of the bounds: If l_1 is negative and if both u_1 and l_2 are positive, then

$$(l_1, u_1)/(l_2, u_2) = (l_1/l_2, u_1/l_2).$$

Note that performing an interval division this way requires the ability to decide the signs of the bounds. More generally, the ability to compare bounds is needed. This is solved in Section 3.1 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 an interval $X \neq \bot_I$. Assuming we have an interval extension F of f, then the interval $F(X)$ is an acceptable answer.

Usually, if X appears several times in the unfolded expression of $F(X)$, the wider the interval X the poorer the bounds obtained from $F(X)$. Ultimately, $F(X)$ may well become $(\bot, \bot)$ (or worse: $\bot_I$), which does not provide any useful data on the extremal values of function f. A simple example of this loss of correlation is shown by the function $f = x \in \overline{\mathbb{R}} \mapsto x - x$. An immediate interval extension is $F = X \in \mathbb{I} \mapsto X - X$. Considering $X = (-10, 10)$, the expression $F(X)$ evaluates to $(-20, 20)$, while a sharp enclosure of f on X would be $(0, 0)$.

Refining intervals 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 $\bot_I$, 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 $F((l,l)) \vee F((u,u))$. As the interval (l,l) contains one single value when $l \neq \bot$, the interval $F((l,l))$ should not be much bigger than the set $\{f(l)\}$ for any F that is a reasonable interval extension of f. As a consequence, the junction $F((l,l)) \vee F((u,u))$ should be close to a sharp enclosure of f on X. The result is identical if $F'(X)$ does not contain any positive values.

When $F'(X)$ contains both positive and negative values, it is still possible to find a better enclosure of f, while it may not be sharp any longer. As long as $F'(X)$ has finite bounds, 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 proposition states:

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

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: the multiplication by $X - (a,a)$, which is an interval containing only “small” values around zero, will mitigate the loss of correlation.

We now have two ways of computing an enclosure of f on X. As a consequence, $F(x) \wedge (F((a,a)) + (X - (a,a)) \cdot F'(X))$ is also an enclosure of f on X. For an implementation, the value a can be chosen as the midpoint of X.

3 Computational datatypes

If a function F is an interval extension of f, then any function G is also an interval extension of f as long as $\forall X \in \mathbb{I}, F(x) \preceq G(X)$. As a consequence, instead of choosing the “best” bounds for an interval extension, one can decide to use the most practical ones instead. For example, both intervals $(\sqrt{5}, \sqrt{5})$ and

$(2,3)$ contain $\sqrt{5}$, but the second is a lot easier to manipulate if one only needs to prove that $\sqrt{5}$ is bigger than 1.

Only a subset $\mathbb{F}_\beta$ of $\overline{\mathbb{R}}$ will be considered during computations on interval bounds. Radix β is an integer bigger than one. In addition to $\perp$, $\mathbb{F}_\beta$ will contain all the rational numbers of the form $m \cdot \beta^e$ with m and e relative integers. While $\mathbb{F}_\beta \setminus \{\perp\}$ is a ring for $(+, \times)$, this property will hardly matter, as interval computations do not need sharp bounds but only outer enclosures.

3.1 Floating-point arithmetic

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 quotient is meant to be the lower bound of an interval, e.g. because we are performing an interval division, we can chose any floating-point number $m \cdot \beta^e$ less than the ideal quotient. Among these numbers, we can restrict ourselves to numbers with a mantissa m represented with less than p digits in radix β (that is, $|m| < \beta^p$). This is an infinite yet discrete set, so it has a maximum w representable with a floating-point number. 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 $\left(\frac{5}{7}\right)^{2^n}$ done by n successive 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 it does not matter when performing interval arithmetic.

There have been two prior formalizations of floating-point arithmetic in Coq, at least. The first one [6,7] defines rounded results with relations, so the value w would be expressed as satisfying the proposition:

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

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 [8] has introduced effective floating-point operators, but only for addition and multiplication. For floating-point division, an external oracle provides the result and it just has to be checked for validity, which can be achieved with multiplications only.

Implementation So this work relies on a new formalization of floating-point arithmetic, which provides floating-point operators for all the basic operations. This formalization is also further away from IEEE-754 than the two prior ones, as it gets rid of the underflow mechanism. Indeed, while this restriction on the range of exponents is useful when designing processors, it is a hindrance when performing numerical computations. So the exponent range is simply unbounded and there is no loss of precision around zero with this new implementation.

The prototype of the arithmetic operators is

$$\textit{precision} \to \textit{mode} \to \mathbb{F}_\beta \to \cdots \to \mathbb{F}_\beta.$$

So, in addition to the floating-point operands, they also take as arguments a positive integer (bigger than one to avoid degenerate cases) for precision and a rounding mode chosen among the four modes mandated by IEEE-754: to nearest (tie break to even mantissas), toward zero, toward $-\infty$, and toward $+\infty$.

For the purpose of interval arithmetic, only the last two modes are used in this work: Computations are rounded toward $-\infty$ for lower bounds, and toward $+\infty$ for upper bounds. Precision is a global parameter of the algorithms and it is passed down to all the floating-point computations. Setting the precision is a trade-off: A high value can help in proving some propositions, but it also slows down numerical computations.

3.2 Interface and specializations

Operators provided by this formalization are somewhat slow. As β is a generic integer, the operators are missing some optimizations. Indeed, an efficient floating-point implementation needs fast shifts in radix β: left shifts for aligning mantissas, right shifts for rounding numbers. In the generic operators, these shifts are emulated by multiplication and division by powers of β. But when mantissas are represented in the same radix than β, a left shift can be performed by simply adding zeros at the least significant side of the representation, while a right shift can be performed by deleting the least significant digits. Floating-point arithmetic also needs fast integer logarithm in radix β.

These implementation details do not matter to interval algorithms. In order to abstract them, floating-point arithmetic has been described as an interface (a "Module Type" in Coq syntax) which provides all the operators and their correctness theorems. The interval algorithms are then implemented as module functors that can be parametrized by any module that provides floating-point arithmetic.

There are three such implementations of floating-point arithmetic. The first one simply encapsulates the generic floating-point operators. The two other ones provides fast specializations for $\beta = 2$. One is based on the standard integers provided by Coq as little-endian lists of bits. These lists make shifts and logarithms trivial to efficiently implement. The other one is based on integers represented as binary trees with leafs being digits for radix 2^{31} [9]. While shifts are slower than for the previous implementation, arithmetic operations are faster, as Coq can delegate arithmetic on leafs to the computer processor [10].

Interval arithmetic is also described as an interface that is used to parametrize all the higher level algorithms. There is one single implementation of interval arithmetic and it uses floating-point bounds. But one could imagine providing implementation with other kinds of bounds (e.g. rational numbers) or other interval representations (e.g. midpoint - radius).

3.3 Straight-line programs

For now, we can only perform 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 programs 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, as it is worth avoiding to recompute each occurrences of a common sub-expression, the expression is stored as a straight-line program. This is a directed acyclic graph with an explicit topological ordering on the nodes which contain arithmetic operators.

The following straight-line program describes the expression $\sqrt{x} - y \cdot \sqrt{x}$ (as long as v_{-1} is x and v_{-2} is y).

$$\begin{aligned} v_0 &\leftarrow (\mathrm{sqrt}, v_{-1}) \\ v_1 &\leftarrow (\mathrm{mul}, v_{-2}, v_0) \\ v_2 &\leftarrow (\mathrm{sub}, v_0, v_1) \end{aligned}$$

A statement is allowed to access previous results only. So rather than storing the absolute indexes of the previous results for each operations, their relative positions are stored. In other words, during an evaluation, each statement pushes on the top of a stack its result, and the next statements will look into the stack to find their operands.

The evaluation function is generic. It takes a list of statements, the type A of the inputs and outputs (e.g. $\overline{\mathbb{R}}$ or $\mathbb{I}$), 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 returns the stack containing the results of all the statements. Whenever a statement tries to access past the bottom of the evaluation stack, a default value of type A is used, e.g. 0 or $\bot$ or $\bot_I$.

By passing various sets A and operators on A, the evaluation function will produce 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.

4 Automatic proofs

The convertibility rule has two main uses. First it helps transforming logical propositions into data structures on which programs can actually compute. Second it gives a meaning to the subsequent numerical computations. The details will be hidden behind “tactics”, which are tools available to the user of the Coq proof system.

4.1 Converting terms

Convertibility is first used to check that the transformation of propositions into data structures are sound. This process is called reflexion [11]. While checking is done by Coq, the transformation itself cannot be performed with Coq's term language, as the syntax of terms is not available at this level. Three solutions can be considered. First, one could ask the user to perform the transformation itself. This may be fine for small terms, but it quickly becomes cumbersome. Second, one could integrating 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 inside Coq. Third, one could use the tactic language embedded in Coq [12], so that the transformation runs on an unmodified Coq interpreter. This third way is the one implemented.

Let us assume that the proposition to prove is $\sqrt{x}-y\cdot\sqrt{x} \le 9$. In order to use programs, the tactics have to transform the proposition into a data structure. The left hand side will be transformed into a straight-line program, while the right hand side will be transformed into an interval bound which is a floating-point number (with $\beta = 2$). Let us consider this last transformation only, as the first one is conceptually similar but with a more complicated parser. So the goal is to find a positive integer that is the mantissa of this number (which we choose to have exponent 0, so the mantissa will actually be the integer 9).

The real number 9 is actually `9%R` with Coq notations, that is the string `9` parsed with the grammar rules for real numbers `R`. This is simply a notation which Coq's parser expands to `(1 + (1 + 1) * ((1 + 1) * (1 + 1)))%R`, as 0 and 1 are the only two integer constants known by Coq. Conversely, Coq's pretty-printer displays this tree of additions, multiplications, and ones, as the compact notation `9%R` for the sake of readability.

So the objective is to parse this expanded tree and to find the related integer represented by a list of bits. In the Coq formalization of positive integers, the start of this list is the most-significant bit `xH` (necessarily 1) while the other items are `xO` (for 0) and `xI` (for 1), going down to the least-significant bit. Since the binary decomposition of 9 is $1 + 0 \cdot 2 + 0 \cdot 4 + 8$, it is represented by the structure `xI (xO (xO xH))`. This structure is generated by the following tactic:

```
Ltac get_mantissa t :=
  let rec aux t :=
    match t with
    | 1%R => xH
    | 2%R => constr:(xO xH)
    | 3%R => constr:(xI xH)
    | (2 * ?v)%R =>
      let w := aux v in constr:(xO w)
    | (1 + 2 * ?v)%R =>
      let w := aux v in constr:(xI w)
    end in
  aux t.
```

This tactic defines a `rec`-ursive function `aux` which is called on a term `t`. This term is `match`-ed against several syntactic patterns, potentially containing a hole `v` matching any term. When a pattern matches, a recursive call is performed on the sub-term `v` if any, and then an integer is `constr`-ucted and returned by the function. The returned structure is the expected one, which can also be written as the string `9%positive`.

As a tactic is not proved, there is no guarantee it did not return a random integer, e.g. `42%positive`. This is where convertibility strikes in. The main tactic takes the original proposition and replaces it by `(sqrt x - y * sqrt x <= P2R (9%positive))%R`. It then tells Coq that proving this new proposition is sufficient. There is no reason for Coq to trust the tactic, so the assistant checks that both propositions are indeed convertible. Here it amounts to checking that the term `P2R 9%positive` evaluates to the exact same term than `9%R`. The function `P2R`[2] is written in the term language of Coq, and hence usable inside proofs:

```
Fixpoint P2R (p : positive) :=
  match p with
  | xH => 1%R
  | xO xH => 2%R
  | xO t => (2 * P2R t)%R
  | xI xH => 3%R
  | xI t => (1 + 2 * P2R t)%R
  end.
```

In the end, the various tactics have converted the proposition $\sqrt{x} - y \cdot \sqrt{x} \leq 9$ into the following convertible proposition. For the sake of readability, some arguments are left out of it:

$$\texttt{contains }(\bot,\ +9\cdot\beta^{0})\ \texttt{(eval }\overline{\mathbb{R}}\ \texttt{[Sub 1 0, Mul...] [x, y])}$$

Note that the tactics perform some extra work so that the generated straight-line program contains only one occurrence of $\sqrt{x}$, instead of the two occurrences of the original proposition.

4.2 Proving propositions

The new proposition can then be proved by applying this lemma: $v \in A \Rightarrow A \preceq B \Rightarrow v \in B$. It splits the proposition into two parts, with `X` and `Y` respectively $(+3 \cdot \beta^{-1}, +1 \cdot \beta^{1})$ and $(+1 \cdot \beta^{0}, +33 \cdot \beta^{-5})$:

$$\begin{array}{l}\texttt{contains (eval }\overline{\mathbb{I}}\ \texttt{[Sub 1 0, Mul...] [X, Y])}\\ \qquad\qquad\texttt{(eval }\overline{\mathbb{R}}\ \texttt{[Sub 1 0, Mul...] [x, y])}\end{array}$$

$$\texttt{(eval }\overline{\mathbb{I}}\ \texttt{[Sub 1 0, Mul...] [X, Y])} \preceq (\bot,\ +9\cdot\beta^{0})$$

The first part is a consequence of the interval arithmetic operators being interval extensions of the arithmetic operators on $\overline{\mathbb{R}}$, and of the property of

[2] The standard theory of Coq already provides a function named `IZR` for converting integers to real numbers. But this function is useless for our purpose, as it evaluates the integer 9 to the real `(1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1)%R`, which is provably equal yet not convertible to `9%R`.

interval extension being compatible with function composition. No computations are needed for proving this part.

By defining a suitable `subset` function, a proof of the second part can be replaced by a proof of

$$\texttt{subset (eval } \bar{\mathbb{I}} \texttt{ [Sub 1 0, Mul...] [X, Y]) (}\bot\texttt{, }+9\cdot\beta^0\texttt{)} = \mathit{true}$$

As comparisons between interval bounds are decidable, the `subset` function is actually a program, and so is `eval` on floating-point intervals. As a consequence, Coq can convert the left hand side of the equality to either *true* or *false* by evaluating these programs. If the result is *true*, then the reflexivity of equality applies and the proof is concluded: The original proposition holds true.

To summarize, this proof relies almost entirely on convertibility, except for the four deductive steps, which are instantiations of the following theorems:

1. If the result of a formula on extended reals is contained in an interval non-$\bot_I$, then the same formula on real numbers is well-formed and produces the same result.
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 $B = \mathit{true}$, then any value contained in A is also contained in B.
4. Boolean equality is reflexive.

Bisection and refined evaluation Actually, because of a loss of correlation, the left hand side evaluates to *false* on the example of the introduction. This is expected [13], and two methods experimented with the PVS proof assistant[3] can be reused here. 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 [4].

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 (hence the method is named `bisect` in next section) recursively defined as:

```
bisect $n$ $F$ $(l,u)$ $target$ =
  if $n = 0$ then $false$
  else if subset $F((l,u))$ $target = true$ then $true$
  else let $m$ be the midpoint of $(l,u)$ in
    (bisect $(n-1)$ $F$ $(l,m)$ $target$)&&(bisect $(n-1)$ $F$ $(m,u)$ $target$)
```

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

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

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 true for any $x \in (l, u)$. Once again, the complete proof contains only four deductive steps. Everything else is obtained through convertibility.

As long as n is big enough and the floating-point computations are precise enough (the precision is user-settable), this method can solve most of the provable propositions of the form $\forall x \in (l, u),\ f(x) \in Y$ with f a straight-line program. The method is guaranteed[4] to succeed when l and u are finite and the sharpest enclosure of f on (l, u) 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.

While this method often works, it is practical only when the amount of analyzed sub-intervals is small. To prevent its growth, a second method (namely `bisect_diff`) sharpens the interval evaluation by using the derivatives, as described in Section 2.3. However, while it reduces the number of sub-cases, it increases the memory footprint and the amount of floating-point operations per interval evaluation.

4.3 Performances

The following Coq script contains the formal proof of the toy proposition mentioned in the introduction of this paper.

```
Require Import Reals.
Require Import tactics.

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

The first `Require` imports the standard theory of real numbers, so that the proposition can be expressed. The second one imports the tactics presented in this paper. The `Goal` statement contains the proposition. The position of the `forall` quantifiers does not matter, as the `intros` tactic loads all the hypotheses in the proof context. The `interval` tactic then applies an interval-based method to solve the goal. Its `(i_bisect x)` parameter indicates that `x` is the primary variable on which to perform a bisection. This variable will be abstracted in the real-valued functions the programs handle. Finally, Coq checks a second time that the proof is sound and complete, when it encounters the `Qed` statement.

[4] If there is $x \in (l, u)$ such that $f(x)$ evaluates to $\bot$, Y has to be $\bot_I$. Otherwise, f is continuous on the compact set (l, u), hence it is also uniform continuous. This uniformity ensures that some suitable precision and n exist.

Table 1 shows the time needed for Coq to check the previous proof script, depending on the interval-based method. The first column contains timings when the tactics use the floating-point arithmetic based on the default integer representation of Coq. The second column contains timings when the tree-based representation of integers in radix 2^{31} is used. When a numeric precision is indicated, it means the user manually selected the lowest (hence fastest) precision so that the computations succeed. Otherwise, the default 30-bit precision was used. The timings are relative to the best method, which is the bisection method with derivatives and optimized integers, as expected.

	`Z`	`BigZ`
`bisect`	45.4	2.22
`bisect` $prec = 19$	23.0	2.10
`bisect_diff`	2.66	1.01
`bisect_diff` $prec = 18$	1.56	1.00
CAD	—	∞
Gappa	17.3	—

Table 1. Relative timings for proving fact about $\sqrt{1 + \frac{x}{\sqrt{x+y}}}$.

The CAD algorithm with fast integers has also been executed on this example, but no results were obtained after a decent amount of time. After replacing y by 1 (hence using a uni-variate degree-5 problem), the algorithm was able to finish, but it was still a lot slower than the bisection algorithms.

The last row of the table corresponds to the 19216-line long proof script generated by the Gappa tool [8]. As Gappa is primarily aimed at proving propositions on numerical programs, its methods do not take into account the derivatives of the expression. Indeed, due to rounding, programs do not even represent a continuous expression, so the derivative is useless. Anyway, the proposition is no different from a degenerate program (infinitely-precise computations), so Gappa is able to prove it. Moreover, as it acts as an external oracle, checking the proof involves additions and multiplications only, no division nor square root; and the precision of each single numerical operation is the lowest possible. With respect to the amount of interval computations, Gappa approach is similar to the `bisect` method with `Z` integers. It performs a lot more deductive steps though, which explains why it does not fare much better although it performs simpler interval computations. Note that the relative timing of Gappa was halved to account that, in practice, the other timings are doubled due to the second proof check performed by Coq at `Qed` time.

4.4 Another example

The previous proposition is just a toy example and its occurrence in a real-life proof might be doubtful. The following one is a more realistic example. Taylor models have been experimented in Coq [14] in order to formally prove some inequalities of Hales' proof[5] 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 $(0.5, 2)$.

```
Goal
  forall x, (1/2 <= x <= 2)%R ->
  (Rabs ((((((122 / 7397 * x + (-1733) / 13547) * x
            + 529 / 1274) * x + (-767) / 999) * x
            + 407 / 334) * x + 227 / 925)
         - sqrt x)
      <= 5/65536)%R.
```

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 Coq to automatically prove it with the `bisect_diff` tactic with `BigZ` integers on a desktop computer (or half a minute with `Z` integers). In comparison, the CAD algorithm needs more than ten minutes in Coq. For Hales' proof, one also needs the arctan function, which is in the scope of this tactic. So interval-based methods open the way to using low-degree approximations of elementary functions in Taylor models.

5 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 computationally-intensive nature, they have been seldom used within formal proofs. With the advent of fast program evaluation in proof checkers, the situation is starting to change [8,14].

While all the expressions currently handled are written with basic arithmetic operators, this is not an intrinsic limitation of the interval methods. Interval extensions for usual elementary functions (log, sin, arctan, cosh, and so on) are being added, so that propositions containing these functions can be handled automatically too. Moreover, interval-based methods can also deal with functions defined implicitly, e.g. a zero of an expression or a solution of a differential equation. This is another extension under consideration.

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

Another interesting property of interval-based methods is their scalability: Checking the numerical computations can trivially be split amongst several processing units. Indeed, domains of logical propositions can be split into several parts, as checking one part is a process independent from checking all the other parts. For now, this still requires some user interaction for manually splitting a proposition amongst several proof files and checking these files concurrently [13].

The Coq development presented in this paper is available at

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

References

1. Mahboubi, A.: Implementing the cylindrical algebraic decomposition within the Coq system. Mathematical Structure in Computer Sciences **17**(1) (2007)
2. 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
3. Stevenson, D., et al.: An American national standard: IEEE standard for binary floating point arithmetic. ACM SIGPLAN Notices **22**(2) (1987) 9–25
4. Moore, R.E.: Methods and Applications of Interval Analysis. SIAM (1979)
5. 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)
6. 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
7. Boldo, S.: Preuves formelles en arithmétiques à virgule flottante. PhD thesis, École Normale Supérieure de Lyon (2004)
8. Melquiond, G.: De l'arithmétique d'intervalles à la certification de programmes. PhD thesis, École Normale Supérieure de Lyon, Lyon, France (2006)
9. 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
10. Spiwack, A.: Ajouter des entiers machine à Coq. Technical report (2006)
11. Boutin, S.: Using reflection to build efficient and certified decision procedures. In: Theoretical Aspects of Computer Software. (1997) 515–529
12. 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
13. 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
14. 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