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

Polynomial Function Intervals for Floating-Point Software Verification

Jan Duracz · Michal Konečný

Received: date / Accepted: date

Abstract The focus of our work is the verification of tight functional properties of numerical programs, such as showing that a floating-point implementation of Riemann integration computes a close approximation of the exact integral. Programmers and engineers writing such programs will benefit from verification tools that support an expressive specification language and that are highly automated. Our work provides a new method for verification of numerical software, supporting a substantially more expressive language for specifications than other publicly available automated tools.

The additional expressivity in the specification language is provided by two constructs. First, the specification can feature *inclusions* between *interval arithmetic expressions*. Second, the *integral operator* from classical analysis can be used in the specifications, where the integration bounds can be arbitrary expressions over real variables. To support our claim of expressivity, we outline the verification of four example programs, including the integration example mentioned earlier.

A key component of our method is an algorithm for proving numerical theorems. This algorithm is based on automatic polynomial approximation of non-linear real and real-interval functions defined by expressions. The PolyPaver tool is our implementation of the algorithm and its source code is publicly available. In this paper we report on experiments using PolyPaver that indicate that the additional expressivity does not come at a performance cost when comparing with other publicly available state-of-the-art provers. We also include a scalability study that explores the limits of PolyPaver in proving tight functional specifications of progressively larger randomly generated programs.

This research has been sponsored by Altran Praxis Ltd and EPSRC grant EP/C01037X/1.

Jan Duracz
School of Information Science, Computer and Electrical Engineering,
Halmstad University, Sweden
E-mail: Jan.Duracz@hh.se

Michal Konečný
School of Engineering and Applied Science, Aston University
Aston Triangle, B4 7ET, Birmingham, UK
E-mail: m.konecny@aston.ac.uk

```
-- This function computes an upper Riemann sum of the error function
-- scaled by sqrt(pi)/2, using a uniform partition of size n.
function erfRiemann(x : Float; n : Integer) return Float is
 segmentSize : Float; -- size of one segment in the partition
 result : Float; -- accumulator of the Riemann sum
 segment : Integer; -- number of the current segment, starting from 0
 segmentStart : Float; -- the starting point of the current segment
 valueStart : Float; -- the value of the integrand at segmentStart
begin
 segmentSize := x/Float(n); -- the integration range is 0..x
 result := 0.0; -- initialise the accumulator
 segment := 0; -- start with the first segment
 while segment < n loop -- iterate over segments
   segmentStart := Float(segment) * segmentSize;
   valueStart := exp(-(segmentStart * segmentStart)); -- max at the start
   result := result + segmentSize * valueStart;        --  as e^{-(x*x)} decreasing
   segment := segment + 1;
 end loop;
 return result;
end erfRiemann;
```

$$\left(\texttt{erfRiemann}(x,n) - \int_0^x e^{-t^2}dt\right) \in \left[-(n+1)\cdot\delta\,,\ \frac{x}{n}(1-e^{-x^2}) + (n+1)\cdot\delta\right]$$

Fig. 1: A floating-point program and its functional specification parametrised by $\delta > 0$.

Keywords non-linear numerical constraint solving · theorem proving · floating-point software verification · polynomial intervals · validated computation · interval arithmetic

1 Introduction

The capability to efficiently specify and write useful programs and automatically prove them correct is one of the holy grails of applied computer science. Tools for formal methods, such as SPARK Ada [6], go some way to achieving this goal. There is some evidence [4] that developers of safety-critical systems benefit from employing formal methods when obtaining the required high-level certifications for their software. These certifications could be obtained without formal methods, instead based on extensive and rigorous testing, but in some cases such testing costs substantially more than the application of formal methods [4]. The Tokeneer experiment conducted by the NSA [7] gives strong evidence that formal methods can bring substantial savings also to the development process due to its emphasis on thorough specification and early detection of defects. This work focuses on methods and tools for formal verification of floating-point (FP) software, an area where there is currently demand for improvement.

As an example, consider the Ada function `erfRiemann` and its specification shown in Figure 1. The Ada code and the specification are both hand-written. The function is intended to compute an approximation of the integral that appears in the Gaussian error function (erf), using an upper Riemann sum over a uniform partition of size n. The specification ties the FP result of the function with the intended exact real meaning by bounding the absolute error within an interval that depends on the values of x and n. Note that the bound can be made arbitrarily tight by increasing the parameter n while decreasing the positive constant δ closer to zero at a high enough rate. The minimum δ for which the contract holds depends on the overall magnitude of the compound rounding error incurred during one execution of the loop. The specification reflects the fact that, for very large n, the accumulated rounding error outweighs the improvement in precision gained by refining the partition.

Most of the paper is devoted to our method of automatically proving that programs such as `erfRiemann` meet tight functional specifications. Subsection 5.3 explores in more detail how we have verified `erfRiemann` against the specification in Figure 1. The remainder of this section clarifies how our approach advances the state-of-the-art and outlines its technical content.

1.1 Related work

Published approaches to FP program verification can be classified by the level of specification they aim to verify. Let us consider the following levels of specification for FP programs:

– *Exception freedom*: No overflow, division by zero, illegal parameter to square root, etc.
– *Rough functional specification*: Bounding the computed values with expressions, but not attempting to describe the values exactly. The gap between the bounds is typically so large that one cannot determine more than a few (if any) significant digits of the computed FP values. This usually happens when using constant or linear bounds for a nonlinear computation.
– *Tight functional specification*: As above, but the gap between the bounds is typically small enough to ensure accuracy of some of the significant digits of the computed FP numbers or to closely express the model error in the computation. The expressions often relate the computation to its intuitive meaning, which tends to be a non-linear expression featuring operators such as root, exponential, logarithm, sine, cosine, arctan, absolute value, or integral.
– *Very tight functional specification*: The computed FP values are expressed exactly as expressions in terms of the input FP values. The specifications need to take precise account of rounding and typically depend on specific details of the FP arithmetic such as rounding mode.

The above levels are progressively harder to specify and verify, besides that establishing exception freedom is usually equivalent to establishing rough functional specification. Note that in the process of verifying a tight functional specification, one usually verifies exception freedom too.

How much can verification of various levels of specification be automated with current technology? Exception freedom and rough functional specification can be automatically derived and verified, even for industrial-scale programs, using tools such as Astreé [12]. Tight and very tight properties of human-authored programs typically cannot be automatically derived. Tight properties can be automatically verified only for relatively short and simple programs, and it is often necessary to manually add loop invariants. Very tight specification can be automatically verified only for the simplest of programs. Verifying a non-trivial program to this level usually requires significant human expert effort and represents a significant achievement [8,9].

To our best knowledge, no other published tools can automatically verify `erfRiemann` with its tight specification in Figure 1, even after adding a suitable loop invariant. We will now explain why this is the case, focusing on some of the better known tools.

Abstract interpretation tools such as Astreé [12] or Fluctuat [15,22,32] do not address tight functional specification.

Verification of a functional specification such as ours is typically performed in two steps. First, one derives verification conditions (VCs), which are mathematical theorems that imply the correctness of the programs.[1] Second, one proves the VCs using a theorem prover. This approach is common to SPARK Ada [5], Frama-C [13], and Why [20], and we adopt it too. We therefore need to ask whether some of the available theorem provers can automatically prove the VCs obtained for our example verification problem. These VCs will contain variations of the formula in Figure 1, including the integral operator and interval inclusion. For example, one of the VCs produced for the program in Figure 1 by SPARK is of the form

$$
\begin{aligned}
& r - \int_0^{s \circledast (x \oslash n)} e^{-t^2}\,\mathrm{d}t \;\subseteq\; \left[-(s+1)\cdot\delta,\ \tfrac{x}{n}\left(1 - e^{-(xs/n)^2}\right) + (s+1)\cdot\delta\right] \\
& \wedge \qquad s = n - 1 \\
& \wedge \qquad (\ldots\ \mathit{omitted}) \\
& \Longrightarrow \\
& r \oplus \Big((x \oslash n) \circledast \mathtt{exp}\big(-((x \oslash n) \circledast s) \circledast ((x \oslash n) \circledast s)\big)\Big) - \int_0^{x} e^{-t^2}\,\mathrm{d}t \\
& \qquad\qquad\qquad\qquad \subseteq \left[-(n+1)\cdot\delta,\ \tfrac{x}{n}\left(1 - e^{-x^2}\right) + (n+1)\cdot\delta\right]
\end{aligned}
$$

where $\oplus$, $\circledast$, $\oslash$, and `exp` are rounded (FP) addition, multiplication, division, and exponentiation, respectively.

Automated numerical provers such as RealPaver [23], Gappa [14] and MetiTarski [3] cannot at present deal with the integral operator. Also, while some of these provers support interval expressions, none are general enough to express the interval in our specification.

The VCs can be expressed in full-featured interactive theorem provers such as Coq [10,11] or PVS [30]. With suitable tactics one could perhaps automate their proof. Nevertheless, we have not found a tactic that could automatically prove theorems that feature an integral similar to the one in our example.

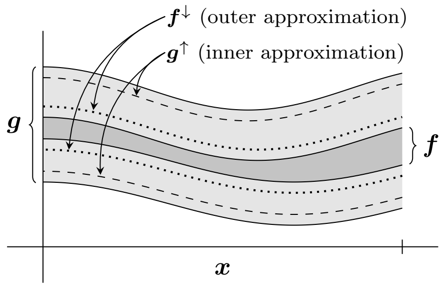
Inspired by our verification example, we identify the following two challenges:

Challenge 1 (Automation). Find a way to prove tight functional specifications of FP programs automatically and efficiently, i. e. in a push-button manner accessible to programmers and engineers.

Challenge 2 (Expressivity). Automatically verify specifications involving parametrised interval inclusions and the integral operator. This should happen with a comparable efficiency as when verifying specifications not involving these constructs.

We consider RealPaver, MetiTarski, and Gappa to be the state of art in addressing Challenge 1 and we are not aware of any tool, beside ours, that addresses Challenge 2.

[1] The derivation of VCs is automatic except for the derivation of loop invariants.



$\boldsymbol{f}^{\downarrow}$ (outer approximation)
$\boldsymbol{g}^{\uparrow}$ (inner approximation)
$\boldsymbol{g}$
$\boldsymbol{f}$
$\boldsymbol{x}$

Fig. 2: Illustration of a numerical proof of an inclusion of unary interval functions $\boldsymbol{f}$ and $\boldsymbol{g}$.

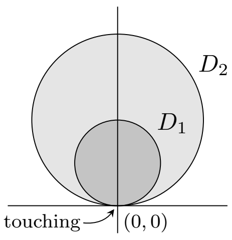


Fig. 3: Touching in the inclusion $D_1 \subseteq D_2$

1.2 Contributions

- We present the essence of the algorithm in PolyPaver, our constraint-based numerical theorem prover. This algorithm overcomes Challenge 2 and thus significantly extends the state-of-the-art of automatic verification of FP programs. Our benchmarks in Section 6 indicate that, despite the boost in expressivity, the algorithm performs comparably to publicly available state-of-the-art provers in terms of its ability to prove numerical theorems automatically and efficiently (Challenge 1). The following points outline how the algorithm overcomes Challenge 2:
  - Inequalities are decided using automatically computed polynomial approximations of the functions given by the left- and right-hand-side expressions.
  - Polynomial approximations are also computed for expressions with the integral operator. PolyPaver can thus decide some tight properties featuring integrals.
  - Our polynomial approximation approach naturally extends to approximating interval expressions and thus deciding interval inclusions.
- We provide four FP programs with tight specifications, whose verification is challenging relative to the state-of-the-art as of April, 2013 (Section 5). These are suitable for inclusion in FP program verification and numerical theorem proving benchmarks.

1.3 Overview of our proof method

In essence, a *polynomial interval* (PI) is a pair of polynomials $[p_L, p_R]$ in variables $x_1, \ldots, x_m$ ranging over some intervals $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m$, respectively. A PI usually represents some information about a specific function f, namely that the graph of f lies between the graphs of p_L and p_R on the whole domain $\boldsymbol{x}_1 \times \boldsymbol{x}_2 \times \cdots \times \boldsymbol{x}_m$.

Figure 2 illustrates how an interval inclusion statement $\forall x \in \boldsymbol{x} : \boldsymbol{f}(x) \subseteq \boldsymbol{g}(x)$ can be proved using PIs. The left-hand-side real or interval expression $\boldsymbol{f}$ is "outwards" approximated by a PI $\boldsymbol{f}^{\downarrow}$, indicated by dotted lines, and the right-hand-side interval expression $\boldsymbol{g}$ is "inwards" approximated by a PI $\boldsymbol{g}^{\uparrow}$, indicated by dashed lines. The inclusion is derived from $\boldsymbol{f}(x) \subseteq \boldsymbol{f}^{\downarrow}(x) \subseteq \boldsymbol{g}^{\uparrow}(x) \subseteq \boldsymbol{g}(x)$. An inequality is proved similarly but using outer approximations for both sides of the inequality.

This approach relies on a separation between the boundaries of the regions defined by $\boldsymbol{f}$ and $\boldsymbol{g}$. As long as the boundaries are separated over a compact rectangular domain $\boldsymbol{x}$, and the expressions $\boldsymbol{f}$ and $\boldsymbol{g}$ contain only operators supported by our PI arithmetic, the arithmetic can be used to automatically produce PI approximations $\boldsymbol{f}^{\downarrow}$ and $\boldsymbol{g}^{\uparrow}$ tight enough to separate the boundaries.

The following example gives an inclusion where such separation is not possible due to touching of the boundaries of the regions.

Example 1 (Touching) Consider the discs $D_k = \{(x, y) \in \mathbb{R}^2 \mid \sqrt{x^2 + (y-k)^2} \leq k\}$ in the real plane $\mathbb{R}^2$ (see Fig. 3). We have a theorem $\varphi \equiv D_1 \subseteq D_2$. There is touching in φ because the boundaries of the discs D_1 and D_2 share the point $(0, 0)$. $\square$

Our method does not prove inclusions or inequalities with touching except in rare cases such as when the touching boundaries are polynomials of the same kind as used in the PIs. Our method does not attempt to detect touching. If touching occurs only on a small part of the domain, such as a single point, our method proves the theorem is true on all parts of the domain except in an arbitrarily small neighbourhood of the area where touching occurs.

In numerical theorems without touching, we sometimes discuss an informal degree of *tightness* which indicates the minimum distance between the supports of two constituent predicates. The tighter a theorem is, the harder it is to find the separating approximations $\boldsymbol{f}^{\downarrow}$, $\boldsymbol{g}^{\uparrow}$.

As explored in [18], a non-trivial complication arises when approximating an interval expression such as $\sqrt{x}\cdot[-1, 1]$ inwards. One first computes inner PI approximations of both $\sqrt{x}$ and $[-1, 1]$ and then applies an inner-rounded PI product. The inner approximation of $\sqrt{x}$ is an interval, say $[p_{\mathrm{L}}, p_{\mathrm{R}}]$, which is not an interval in the usual sense, because $p_{\mathrm{L}} \leq p_{\mathrm{R}}$ does not hold. Instead, $p_{\mathrm{L}} \geq p_{\mathrm{R}}$ holds. We call such intervals *inverted*. Although the inner approximation of $\sqrt{x}$ is inverted, the inner approximation of the product $\sqrt{x} \cdot [-1, 1] = [-\sqrt{x}, \sqrt{x}]$ is typically not inverted.

Thus, we have to account for inverted intervals, and define an arithmetic of PIs that works for a mixture of normal and inverted intervals. Moreover, there can be PIs that are normal on a part of the domain and inverted on another part of their domain. For example, an inner PI approximation of the function interval $[0, \sin(x)]$ over $x \in [0, 1]$ is typically inverted at $x = 0$ but normal at $x = 1$. The approximation is likely to be inverted at $x = 0$ because the exact interval $[0, \sin(x)]$ is a singleton at $x = 0$. We adopt the name *generalised intervals* for pairs $[\alpha,\beta]$ without any restriction on the relative order of the endpoints α,β.

When defining operations for generalised function intervals, one of the hardest cases is multiplication. Kaucher [24] defined a multiplication of generalised real intervals that is suitable for our purposes. Nevertheless, it is defined in 16 distinct cases when the operands are different combinations of normal, reversed, covering zero or entirely positive or negative intervals. This definition cannot be directly applied to function intervals, because, at different parts of the function domain, the operand values can be at a different one of the 16 cases, possibly covering all 16 cases in one pair of interval operands. The details of our solution to this problem, as well as other issues related to the definition of PI arithmetic, are in Section 3.

The recurring issue when defining basic PI operations is the need for *rounding consistently over the whole domain* of the PIs. Another perpetual concern is the growth of size of the polynomials. Such growth is controlled by an operator for polynomial size reduction. These issues makes the design and implementation of PI arithmetic fairly challenging.

An important feature of our PI arithmetic is that it is well suited for proving theorems that involve indefinite integrals, thanks to the simplicity of integrating polynomials.

Polynomials have been used previously to approximate sets of functions but they all differ substantially from our PIs. Taylor models (TMs) [25, 27] are essentially function intervals of constant width, centred around one polynomial. The TM approach has been proposed as a general solution to the precision loss problem, but the merit of this approach has been disputed in [28]. We believe that our PI approach is more promising than TMs for reducing the dependency problem in constraint solving. PI multiplication is more accurate than TM multiplication, thanks to using two different polynomials as bounds, as discussed in Subsection 3.3. Another improvement of our PIs over TMs is in the use of the Chebyshev basis, which leads to a much improved rounding, as discussed in Subsection 3.1.

1.4 Outline of sections

Sections 2–4 give a formal account of the most important concepts of our numerical theorem proving algorithm, while Sections 5–7 focus on practical applications of our method.

Section 2 starts the formal account by defining a general approach to proving inequalities and interval inclusions by validated numerical approximation. Subsection 2.1 introduces generalised intervals over partially ordered sets and the approximation order on these intervals. Subsection 2.2 extends the concept of generalised interval to the concept of generalised interval function and some operations and constructions involving interval functions. Subsection 2.3 formalises an extension of common order and inclusion predicates for intervals and interval functions. Subsection 2.4 discusses how interval functions are used to approximate the values of numerical expressions and predicates. Subsection 2.5 introduces domain subdivision as a technique to improve approximations of numerical expressions in the context of deciding numerical predicates.

Section 3 provides some detail of our novel polynomial interval arithmetic. Subsection 3.1 specifies how multivariate polynomials are represented and defines a safe “rounding” of polynomials to polynomials that can be represented in a bounded space. The remaining subsections are dedicated to various approximate operations over polynomial intervals, covering addition, multiplication, pointwise maximum and minimum, division, exponentiation and square root, and integration, respectively.

Section 4 describes PolyPaver’s algorithm for deciding formulas using domain subdivision, the language of formulas supported by PolyPaver (Subsection 4.1), how these formulas are approximately evaluated using polynomial intervals (Subsection 4.2), giving specific attention to the evaluation of integrals (Subsection 4.3).

A formal treatment of the derivation of a correctness theorem (i. e. verification conditions) from a program-specification pair is outside the scope of this paper. Such treatment can be found in [16].

The use of PolyPaver in program verification is illustrated in Section 5, where we discuss how several programs are proved correct with respect to their formal functional specification. The examples cover the computation of the error function (erf) using a rational approximation (Subsection 5.1), computation of the square root using Newton’s iterative method (Subsection 5.2), computation of the error function using a Riemann sum as introduced in this section (Subsection 5.3), a selection of verified programs from the ProVal project (Subsection 5.4), and a program for computing the maximum of a quadratic interpolation of three points, comprising several cooperating procedures (Subsection 5.5).

Section 6 considers PolyPaver as a general numerical prover, and compares it with other such provers using several benchmarks. The first benchmark is based on a verification condition from the square root verification example (Subsection 6.1). Subsection 6.2 compares the provers based on several hand-crafted sequences of progressively more complex formulas.

Section 7 concludes the experimental part of the paper by reporting on a scalability study, based on randomly generated formulas, studying the effect of arity, expression depth and expression sizes on PolyPaver's proof success rate and proving duration.

Section 8 concludes the paper and discusses possible avenues of further work.

2 Background

We take the same high-level approach to automated numerical theorem proving as RealPaver [23]. This approach is based on *safe numerical approximation* and *domain subdivision*. The main point of departure from RealPaver and other similar solvers is that we employ a more precise interval arithmetic, namely the PI arithmetic introduced in Subsection 1.3 and outlined in Section 3. To prepare the ground for a presentation of our arithmetic, the remainder of this section formalises the general approach to safe approximation of numerical theorems, as well as of the expressions, relations, and predicates from which the theorems are built up.

2.1 Approximation in posets

Let $(R, \leq_R)$ be a *partially ordered set* (*poset*) [29]. The *poset of generalised intervals* over R, denoted $(\mathbb{J}(R), \sqsubseteq_R)$, is the set $\mathbb{J}(R) = \{[a, b] \mid a, b \in R\}$ of pairs of elements of R, equipped with the *refinement relation* $\sqsubseteq_R$, defined by:

$$[a, b] \sqsubseteq_R [c, d] \quad\Longleftrightarrow\quad a \leq_R c \text{ and } d \leq_R b.$$

The *conjugate* of $[a, b] \in \mathbb{J}(R)$ is given by $[b, a]$. The sub-poset $\mathbb{I}(R) = \{[a, b] \mid a \leq_R b\}$ of $\mathbb{J}(R)$ is usually called the poset of *order intervals* over R. We identify an order interval $[a, b]$ with the set $\{x \in R \mid a \leq_R x \text{ and } x \leq_R b\}$, viewed as a sub-poset of R. The refinement relation on $\mathbb{J}(R)$ coincides with the opposite of interval inclusion whenever $[a, b]$ and $[c, d]$ are order intervals, i. e. $[a, b] \sqsubseteq_R [c, d] \iff [a, b] \supseteq [c, d]$.

We shall often refer to generalised intervals simply as intervals. We often omit the subscript R in $\leq_R$ and $\sqsubseteq_R$.

With $\boldsymbol{a} \sqsubseteq \boldsymbol{b}$, we say that $\boldsymbol{a}$ is an *outer approximation* of $\boldsymbol{b}$, and conversely, $\boldsymbol{b}$ is an *inner approximation* of $\boldsymbol{a}$.

A function $f : R \to R$ is called *isotonic* if $x \leq y \Rightarrow f(x) \leq f(y)$, and *antitonic* if $x \leq y \Rightarrow f(y) \leq f(x)$, for all $x, y \in R$. Functions with multiple arguments are called isotonic and antitonic if they are isotonic and antitonic in each argument, respectively. Functions on intervals that are isotonic in the refinement order are of particular interest. Such functions preserve the outer approximations in the following sense: If $\boldsymbol{x}$ is an outer approximation of $\boldsymbol{y}$ (i. e. $\boldsymbol{x} \sqsubseteq \boldsymbol{y}$) and g is an refinement-isotonic function, that also provides an outer approximation of $f(\boldsymbol{z})$ for every $\boldsymbol{z}$, then $g(\boldsymbol{x})$ is an outer approximation of $f(\boldsymbol{y})$, because $g(\boldsymbol{x}) \sqsubseteq g(\boldsymbol{y}) \sqsubseteq f(\boldsymbol{y})$. Dually, refinement-antitonic functions preserve inner approximations.

Note that in our context inverted intervals occur only when computing inner approximations, which occurs only while proving or disproving an interval inclusion as discussed in Subsection 1.3. For deciding inequalities and other predicates, we compute only outer approximations and thus always obtain order intervals.

Due to reflexivity of partial orderings, $\mathbb{I}(R)$ is only empty if R is the empty set. There is a canonical *singleton map* $R \hookrightarrow \mathbb{I}(R)$, mapping elements $x \in R$ to singleton intervals $[x, x] \in \mathbb{I}(R)$. We shall often identify poset elements with their corresponding singleton intervals, and generally say that R is approximated by $\mathbb{I}(R)$ or $\mathbb{J}(R)$ through this identification.

2.2 Approximation of functions

Poset-valued functions form posets under pointwise ordering. Let X be a set and $f, g : X \to R$, then the *pointwise order* $\leq_{X\to R}$ is defined by:

$$f \leq_{X\to R} g \iff \forall x \in X : f(x) \leq_R g(x).$$

We may thus consider the poset of *function intervals* $\mathbb{J}(X \to R)$ ordered under *pointwise refinement* $\sqsubseteq_{X\to R}$.

The approximation terminology transfers from ordinary intervals to function intervals as follows. Given $\boldsymbol{f}, \boldsymbol{g}, \boldsymbol{h} \in \mathbb{J}(X \to R)$ such that $\boldsymbol{f} \sqsubseteq \boldsymbol{g} \sqsubseteq \boldsymbol{h}$, we say that $\boldsymbol{f}$ is an *outer approximation* of $\boldsymbol{g}$ over X, and that $\boldsymbol{h}$ is an *inner approximation* of $\boldsymbol{g}$ over X.

The map $\mathbb{J}(R) \hookrightarrow \mathbb{J}(X \to R)$ taking an interval $[l, r]$ to the singleton function interval $[\lambda x.l, \lambda x.r]$ is called the *constant map*, and function interval approximations using only constant bound functions are called *constant interval approximations*.

A free *variable* in an n-ary expression can be approximated with no loss of precision using a projection function in the following sense:

Definition 1 (Projection) Let $\boldsymbol{x} = (\boldsymbol{x}_1, \dots, \boldsymbol{x}_n) \in \mathbb{I}(R)^n$ be a tuple of intervals. The function interval $[\lambda x.x_i, \lambda x.x_i] \in \mathbb{I}(\boldsymbol{x} \to R)$, where x_i is the i^{th} component of $x \in \boldsymbol{x}$, is called the i^{th} *projection* over $\boldsymbol{x}$, and denoted by $proj_i(\boldsymbol{x})$.

When one is restricted to constant interval approximations, it is impossible to approximate variables using projections. In such cases, one uses the constant interval approximation with the domain whose value is the domain of the variable.

Refinement-isotonic function interval operations may be used to build up interval approximations of functions given by expressions with operators. The following example specifies two simple function interval operations.

Example 2 (Interval operations) Consider the poset $X \to \mathbb{R}$ of real-valued functions with point-wise ordering and arithmetic operations, and the associated interval poset $\mathbb{J}(X \to \mathbb{R})$. Addition of intervals and scaling of intervals by real numbers are given by:

$$\begin{aligned} [f_1, f_2] + [g_1, g_2] &= [f_1 + g_1, f_2 + g_2]\,, \text{ and} \\ a \cdot [f, g] &= \begin{cases} [af, ag] & \text{if } a \geq 0 \\ [ag, af] & \text{if } a < 0 \end{cases} \end{aligned} \tag{1}$$

for $a \in \mathbb{R}$. $\square$

See Kaucher [24] for a more complete set of refinement-isotonic arithmetic operations on $\mathbb{J}(\mathbb{R})$, which can be point-wise extended to operations on $\mathbb{J}(X \to \mathbb{R})$.

Next, we determine the form of outer interval approximations of elementary FP functions, which we will need when such functions are used in the programs we attempt to verify. The error of a function depends on its implementation. We adopt the method used to specify FP elementary functions in the Ada language standard [1]. There, a FP function $\mathtt{fun}$ is expressed in terms of the exact real function f that $\mathtt{fun}$ is indented to compute and a *maximum relative error* $\varepsilon_{\mathtt{fun}}$ as follows:

$$\mathtt{fun}(x_1, \ldots, x_n) \in (1 + [-\varepsilon_{\mathrm{rel}}, \varepsilon_{\mathrm{rel}}])(1 + [-\varepsilon_{\mathtt{fun}}, \varepsilon_{\mathtt{fun}}]) \cdot f(x_1, \ldots, x_n) + [-\varepsilon_{\mathrm{abs}}, \varepsilon_{\mathrm{abs}}] \tag{2}$$

where $\varepsilon_{\mathrm{rel}} = \beta^{p-1}$ and $\varepsilon_{\mathrm{abs}} = \beta^{e_{\min}}$, with β, p and $e_{\min}$ being the *base*, *precision*, and *minimum exponent* of the FP format, respectively. In the above, the scaling factor $(1 + [-\varepsilon_{\mathtt{fun}}, \varepsilon_{\mathtt{fun}}])$ encloses the truncation error of the implementation and the remaining terms account for one final rounding of the result into the respective FP format.

Thus, (2) tells us that the total approximation error made when computing with a standard-compliant implementation $\mathtt{fun}$ of an elementary function f can be bounded in terms of an interval expression. For each choice of input values $x_1, \ldots, x_n$ to the function the error-bounding expression is given by the right-hand side of (2) which, for each floating point format, varies only in the function f and the real number $\varepsilon_{\mathtt{fun}}$. We can therefore write (2) in a more succinct form by introducing the *generalised outward-rounding operator* $\mathcal{S}_{out}$ below. The operator is an interval function defined by:

$$\mathcal{S}_{out}(e, y) = (1 + [-\varepsilon_{\mathrm{rel}}, \varepsilon_{\mathrm{rel}}])(1 + [-e, e])y + [-\varepsilon_{\mathrm{abs}}, \varepsilon_{\mathrm{abs}}] \tag{3}$$

for real numbers e and y, representing the maximal relative error $\varepsilon_{\mathtt{fun}}$ and function value $f(x_1, \ldots, x_n)$, respectively. We may now write (2) in the more succinct form:

$$\mathtt{fun}(x_1, \ldots, x_n) \in \mathcal{S}_{out}(\varepsilon_{\mathtt{fun}}, f(x_1, \ldots, x_n))$$

or even more succintly by lifting $\mathcal{S}_{out}$ to functions:

$$\mathcal{S}_{out}(\varepsilon_{\mathtt{fun}}, f) \sqsubseteq \mathtt{fun}. \tag{4}$$

2.3 Interval approximation of predicates

Let $\mathbb{B}$ denote the set $\{\mathtt{t}, \mathtt{f}\}$ of Boolean truth values, with $\mathtt{t}$ denoting *true* and $\mathtt{f}$ denoting *false*. Relations $\rho \subseteq X^n$ on some set X may then be viewed as $\mathbb{B}$-valued functions $\rho : X^n \to \mathbb{B}$. We extend $\mathbb{B}$ with the *bottom* value $\perp$, denoting *undecidedness*, write $\mathbb{B}_\perp$ for the set $\{\mathtt{t}, \mathtt{f}, \perp\}$, and define the *flat Boolean* poset $(\mathbb{B}_\perp, \lesssim)$, where the partial order relation $\lesssim$ is determined by $\perp \lesssim \mathtt{t}$ and $\perp \lesssim \mathtt{f}$.

Relations ρ on posets R may be lifted to $\mathbb{B}_\perp$-valued functions $\rho_{\mathbb{I}}$ on the associated interval poset $\mathbb{I}(R)$ in a canonical fashion. We define:

$$\boldsymbol{x} \, \rho_{\mathbb{I}} \, \boldsymbol{y} = \begin{cases} \mathtt{t} & \text{if for all } x \in \boldsymbol{x}, y \in \boldsymbol{y} \text{ it holds } x \rho y = \mathtt{t} \\ \mathtt{f} & \text{if for all } x \in \boldsymbol{x}, y \in \boldsymbol{y} \text{ it holds } x \rho y = \mathtt{f} \\ \perp & \text{otherwise} \end{cases} \tag{5}$$

for $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{I}(R)$. A *solution* of a predicate $p : X \to \mathbb{B}$ is an $x \in X$ such that $p(x) = \mathtt{t}$ and a *counterexample* to p is an $x \in X$ such that $p(x) = \mathtt{f}$. To *prove* p over $A \subseteq X$ is to show that A is contained in the set of solutions of p, i. e. $A \subseteq p^{-1}(\mathtt{t})$, [2] and to *disprove* p over A is to show that A contains a counterexample to p.

From now on we switch from general subsets $A \subseteq X$ to order intervals $A \in \mathbb{I}(R)$, seen as subsets of R. A predicate p is *safely approximated* over $A \in \mathbb{I}(R)$ by a function $\mathcal{A}_p : \mathbb{I}(R) \to \mathbb{B}_\perp$, if $\mathcal{A}_p(A) \lesssim p(x)$ for all $x \in A$. If $\mathcal{A}_p$ is a safe approximation over each $A \in \mathbb{I}(R)$, we call $\mathcal{A}_p$ a *safe interval approximation* of p. In particular, safe interval approximations satisfy the following property: if $\mathcal{A}_p(A) = \mathtt{t}$ then $A \subseteq p^{-1}(\mathtt{t})$ and if $\mathcal{A}_p(A) = \mathtt{f}$ then $A \subseteq p^{-1}(\mathtt{f})$, *i. e.* safe interval approximations allow us to *prove* or *disprove* predicates.

Equation (5) above defines suitable safe interval approximations for most binary relations.

2.4 Safe numerical approximation

Safe interval approximations for atomic formulas such as $f(x) \leq g(x)$ are usually computed using so-called *natural interval approximations*. Natural interval approximations are obtained by interpreting the real number functions, operators, and the relation in the atomic formula with corresponding interval extensions.

The simplest natural interval approximation is the one provided by constant interval extensions. The following example illustrates its use.

Example 3 (Constant interval approximation) Consider the real predicate $p : \mathbb{R} \to \mathbb{B}$ given by $x \mapsto \frac{5x}{8} \leq \frac{3+5x}{8}$. A safe approximation $\mathcal{A}_p$ of p may be computed using constant interval approximations for addition and scaling. We get $\mathcal{A}_p([0,1]) = \mathcal{A}_\leq([0, \frac{5}{8}], [\frac{3}{8}, 1]) = \perp$, where $\mathcal{A}_\leq$ is the safe approximation $\leq_\mathbb{I}$ of $\leq$ given by (5). [3] The result $\perp$ means that the approximation $\mathcal{A}_p$ is too weak to decide p over $[0,1]$. $\square$

The undecided result of Example 3 shows that safe approximations may severely over-approximate, and often fail to decide seemingly trivial predicates. By dividing the decision problem over the initial domain into decision problems over subsets, tighter approximations of numerical expressions are obtained, leading to better approximations of the predicate over the sub-domains. The following section describes this approach.

2.5 Domain subdivision

Domain subdivision is a method for improving an approximation of a predicate by reducing the size of the domain of the predicate. Consider the predicate p from Example 3. We were unable to decide p over $[0,1]$ because of an overlap between the interval approximation $[0, \frac{5}{8}]$ of the left hand side and $[\frac{3}{8}, 1]$ of the right hand side of the inequality. By splitting the domain $[0,1]$ of p into the two halves $[0, \frac{1}{2}]$ and $[\frac{1}{2}, 1]$, and then using $\mathcal{A}_p$ to approximate p over each half, we obtain: $\mathcal{A}_p([0, \frac{1}{2}]) = \mathcal{A}_\leq([0, \frac{5}{16}], [\frac{3}{8}, 1]) = \mathtt{t}$ and $\mathcal{A}_p([\frac{1}{2}, 1]) = \mathcal{A}_\leq([\frac{5}{16}, \frac{5}{8}], [\frac{11}{16}, 1]) = \mathtt{t}$.

[2] We use the notation p^{-1} to denote the inverse image of p, i. e. $p^{-1}(b) = \{x \in X \mid p(x) = b\}$.

[3] $\mathcal{A}_\leq([0, \frac{5}{8}], [\frac{3}{8}, 1]) = \perp$ follows from (5) as neither $x \leq y$ nor $x > y$ holds for all $x \in [0, \frac{5}{8}]$ and $y \in [\frac{3}{8}, 1]$.

We have proved that p is true over sets covering $[0,1]$, and thus true over all of $[0,1]$, using the same approximation $\mathcal{A}_p$ as in Example 3, but because the reduction in domain size led to an improvement in the approximation of p, we managed to decide the predicate, paying the cost of evaluating p twice. The fact that iterated subdivision creates an exponential number of sub-problems means that the improvements in approximation come at an exponential computational cost. It follows that many predicates become practically undecidable with this approach. One such predicate is given in the following example.

Example 4 (Tight inequality) Consider the real predicate $p : \mathbb{R} \to \mathbb{B}$ given by $x \mapsto x \leq x + 2^{-n}$ and its natural *constant* interval approximation $\mathcal{A}_p$. Clearly, p holds over all of $\mathbb{R}$ and therefore over any subset, however evaluating $\mathcal{A}_p$ over an interval $[a,b]$ yields: $\mathcal{A}_p([a,b]) = \mathcal{A}_{\leq}([a,b],[a+2^{-n},b+2^{-n}]) = \big([a,b] \leq [a+2^{-n},b+2^{-n}]\big) = (b \leq a + 2^{-n})$, which holds if and only if the width of the interval $[a,b]$ is 2^{-n} or less. □

The order in which the sub-problems are processed (such as breadth- or depth-first) has an effect on the number of subdivisions that need to be made to decide a predicate using constant interval approximations. Deciding predicate p from Example 4 over the interval $[0,1]$ requires splitting $[0,1]$ into sub-intervals of width $\leq 2^{-n}$. Thus there will be at least 2^n sub-problems, regardless of search strategy. The following example shows that the choice of a search strategy can reduce the complexity class of a decision problem.

Example 5 (Search strategy affects complexity) Consider deciding the real predicate p given by $x \mapsto x > 2^{-n}$ over the interval $[0,1]$. As in Example 4, sub-problems with domain width larger than 2^{-n} yield the undecided value $\bot$. Using breadth-first search generates a best-case exponential number of sub-problems before identifying a counterexample. On the other hand, a left-biased depth-first search descends directly towards 0 and decides p by returning $\mathtt{f}$ for the interval $[0,2^{-n}]$, thus having proved the predicate false by exhibiting a nonempty set of counterexamples, while generating a linear number of sub-problems. □

While the predicates in Examples 3, 4, and 5 can be decided using constant approximations and bisection, those with touching cannot, as is illustrated by the following example.

Example 6 (Undecided predicate) Let the predicate $p : [0,1] \to \mathbb{B}$ be given by $x \mapsto x > 0$ and $\mathcal{A}_p$ be the natural constant interval approximation of p. Although $\mathcal{A}_p(\boldsymbol{x}) = \mathtt{t}$ for each sub-interval $\boldsymbol{x}$ of $[0,1]$ that does not contain 0, $\mathcal{A}_p([0,y])$ is undecided for each $y > 0$. Thus one cannot decide p over $[0,1]$ in a finite time using $\mathcal{A}_p$. □

We have seen that the choice of search strategy can have a dramatic effect on the performance of a decision procedure. Consider a variation on Example 6 where the predicate p is given by $x \mapsto x < 0$. $\mathcal{A}_p(\boldsymbol{x}) = \mathtt{f}$ for each interval $\boldsymbol{x} \in \mathbb{I}([0,1])$ such that $0 <_{\mathbb{I}} \boldsymbol{x}$. If left-biased depth-first strategy is chosen, then the decision problem will not be decided within a finite number of subdivisions. Any other search strategy will decide the predicate as soon as the strategy deviates from left-biased depth-first, thus encountering an interval of counterexamples within a finite number of splits.

For predicates with more than one variable, there is another strategy choice beyond sub-problem processing order. All variable domains may be subdivided at once, but this leads to a rapid increase in the size of the sub-problem set, potentially exhausting the available space resources. Usually, one variable domain is subdivided at a time, which introduces the choice of a *subdivision direction selection* strategy.

A subdivision direction selection strategy determines the *shape* of the domain of sub-problems. As an illustration, consider a decision problem with two variables $x_1, x_2 \in [0, 1]$. The domain of the problem is clearly a unit square, but the domains of its immediate sub-problems are non-square. Clearly, if x_1 is split more often than x_2, then the generated sub-problems will have domains that are thin and long while if x_2 is split more often than x_1, then sub-problems with wide and short domains are generated.

When deciding a formula in which there is a variable that does not influence the validity of the formula, it is a waste of time to split that variable. A poor subdivision direction selection strategy could lead to splitting only this variable, resulting in failing to decide a formula that could possibly be easily decided with another strategy. To avoid this issue, we use a subdivision strategy that revisits each variable within a finite number of iterations, such as the *round-robin* or the *largest-first* splitting strategy. The round-robin strategy maintains the proportions of the initial domain while the largest-first splitting strategy chooses the variable with greatest domain width to split. As a consequence, largest-first splitting leads to sub-problem domains with roughly the same width for each variable.

3 Polynomial Interval Arithmetic

As previously noted, constant interval approximations can be very inefficient. Often the problem is that the chosen approximations are too coarse, meaning they over-approximate ranges for expressions over a domain, forcing an exponentially costly subdivision loop. We propose to alleviate this problem by making the accuracy of the numerical approximation variable. With this variability, failing to decide does not immediately induce subdivision. Instead the accuracy of the approximation is increased and the approximation re-computed, now standing a better chance of deciding a Boolean value for the problem. This approach will be computationally less costly than subdivision, provided that increasing the accuracy has sub-exponential cost. The experiments reported in Subsection 5.2 support this thesis.

We increase the accuracy of approximations by using *polynomial intervals* (PIs), as briefly introduced in Subsection 1.3. PIs are intervals of the form $[p_L, p_R]$ where p_L and p_R are polynomials of a certain degree over a certain domain $I \in \mathbb{I}(\mathbb{R})^n$. Constant intervals considered so far are a special case of PIs, namely those of degree zero. To construct safe polynomial bounds for expressions, we use our polynomial arithmetic to compute upper and lower approximations of the results of pointwise application of common arithmetic operators to polynomials. In some sense, the polynomial arithmetic approximates pointwise functional arithmetic in a way analogous to how FP arithmetic with directed rounding approximates real number arithmetic. A bound on the degree and term size of all polynomials involved in a computation is analogous to the FP precision used in an FP computation in the way it implies a certain level of rounding.

The polynomial coefficients are taken from $\mathbb{F}_g$, which is the set of all binary FP numbers with g-bits long mantissas and g-bits long exponents. We call the number g the *granularity*[4] of the FP number. The granularity of the polynomial coefficients is determined from the

[4] We use the unusual term granularity here because we reserve the term precision for the size of intervals and polynomial enclosures.

granularity used to express the endpoints of its domain I. Thus, when the domain becomes very small, the polynomials gain more flexibility thanks to higher coefficient granularity and the rounding errors for the associated polynomial arithmetic become smaller. In the experiments reported in this paper, one does not need to exceed the granularity and exponent range of the standard machine double FP format and therefore it is used in order to improve speed. Nevertheless, our implementation of polynomial interval arithmetic supports arbitrary granularity, should it be required.

To keep the polynomials within the specified limits, one often needs to reduce the degree and/or term count of a polynomial while rounding upwards or downwards consistently over the whole domain. We describe this operation first.

3.1 Degree and term-size reduction

We represent the polynomials by coefficients in the basis of the Chebyshev polynomials of the first kind, denoted $T_i(x)$ [26], instead of the more common power basis x^i, in order to facilitate size reduction that is both relatively efficient and relatively accurate.

Most of the useful properties of Chebyshev polynomials apply only when restricted to the domain $[-1, 1]$. Therefore, from now on, instead of $I \in \mathbb{I}(\mathbb{R})^n$, the domain of p_{L} and p_{R} will be the unit cube $[-1, 1]^n$. We use the canonical affine transformation to translate between I and the unit domain. For operations that are not point-wise, e.g. when defining a projection $x \mapsto x_i$ or when integrating, we take care to take the domain translation into account. For example, variables in the original expression are translated to certain affine polynomials over the unit domain and not to projections in the sense of Definition 1.

The general format used for a polynomial of degree (at most) m is:

$$p(x) = \sum_{0 \le j_1 + \cdots + j_n \le m} a_j \cdot \prod_{1 \le i \le n} T_{j_i}(x_i) \text{ where } j = (j_1, \ldots, j_n), a_j \in \mathbb{F}_g.$$

Sometimes we refer to such a polynomial using the short-cut notation $P_m\{a_j\}$.

A *reduction* of the above polynomial *to degree $m' < m$ rounding upwards* is performed as follows:

$$\deg^{\mathrm{R}}_{m'}(P_m\{a_j\})(x) = \sum_{0 \le |j| \le m'} \left(a_j \cdot \prod_{1 \le i \le n} T_{j_i}(x_i) \right) \oplus_{\mathrm{R}} \sum\nolimits^{\mathrm{R}}_{m' < |j| \le m} |a_j| \tag{6}$$

where $\oplus_{\mathrm{R}}$ and $\sum_{\mathrm{R}}$ refer to an upwards rounded addition and $|j| = j_1 + \ldots + j_n$. The above additions contribute to the coefficient of the constant term since $T_0(x) = 1$.

When rounding downwards, the last sum in the formula will be subtracted rounding downwards (denoted $\ominus_{\mathrm{L}}$) instead of added rounding upwards (i. e. $\oplus_{\mathrm{R}}$).

The reduction (6) is majoring p thanks to the fact that $T_i(x)$ maps the unit domain to $[-1, 1]$. If reducing by one degree only, this reduction is the nearest-best in the minimax norm ($\mathcal{L}_\infty$) according to [26, Corollary 3.4A, page 49]. More precisely, this corollary states that without the sum term on the right, formula (6) gives the best reduction possible without the requirement of being upwards rounded. It follows that if the sum on the right and the addition operator are exact, we get the best possible upwards-rounded reduction. Rounding spoils this property, so we have in fact only a near-best reduction. When reducing by more than one degree, a more substantial departure from the best upwards-rounded approximation is

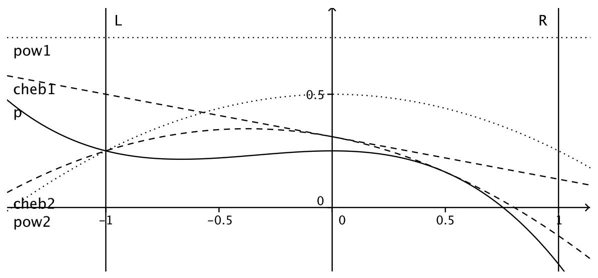


Fig. 4: Reducing polynomial degree by removing terms is more accurate in the Chebyshev basis than in the power basis. Upwards-rounded degree reduction of $p = {}^{1}\!/\!{}_{4}(1-x^2-x^3)$ to degrees 2 and 1. Chebyshev reductions are showed using dashed lines while power reductions are showed using dotted lines.

made but still substantially better than when doing the same in the power basis. As we do not currently have a compatible version of PI arithmetic in the power basis, we cannot conduct a robust test to measure the difference between the bases. In Fig. 4 we illustrate the difference between reductions in Taylor and power bases on an example polynomial.

Reduction in number of terms is performed analogously to degree reduction, except the choice of terms to eliminate is guided by the absolute value of the coefficients — the smallest ones are eliminated and their coefficients added to (or subtracted from) the constant term.

3.2 Addition

Adding two PIs with outwards rounding (denoted $\oplus^{\updownarrow}$) is fairly simple—it amounts to adding the upper bound polynomials rounding upwards and adding the lower bound polynomials rounding downwards:

$$[p_{\mathrm{L}}, p_{\mathrm{R}}] \oplus^{\updownarrow} [q_{\mathrm{L}}, q_{\mathrm{R}}] = [p_{\mathrm{L}} \oplus_{\mathrm{L}} q_{\mathrm{L}}, p_{\mathrm{R}} \oplus_{\mathrm{R}} q_{\mathrm{R}}].$$

An addition of polynomials that is correctly rounding upwards across the whole domain is performed as follows:

$$P_m\{a_j\} \oplus_{\mathrm{R}} P_m\{b_j\} = P_m\{a_j \oplus_{\mathrm{L}} b_j\} \oplus_{\mathrm{R}} \sum\nolimits_{\mathrm{R}} \left(a_j \oplus_{\mathrm{E}} b_j\right)$$

where $\oplus_{\mathrm{L}}$ refers to a downwards rounded addition and $\oplus_{\mathrm{E}}$ is the following upper bound on the rounding error when performing addition:

$$a \oplus_{\mathrm{E}} b = (a \oplus_{\mathrm{R}} b) \ominus_{\mathrm{R}} (a \oplus_{\mathrm{L}} b).$$

Later, we will need similar error estimates for multiplication and sum, denoted $\circledast_{\mathrm{E}}$ and $\sum_{\mathrm{E}}$, respectively. These are defined analogously.

This polynomial addition is correctly rounded upwards because the error in each non-constant term is compensated for by a small increase in the constant term. The difference of upwards and downwards rounded sum is an upper bound on the absolute value of the error in

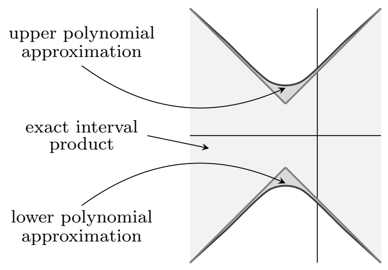

upper polynomial approximation
exact interval product
lower polynomial approximation

Fig. 5: The product of polynomial intervals is not a polynomial interval. An outer approximation of the interval product $[x, x+1] \cdot [-1, 1]$ for x ranging over the unit domain $[-1, 1]$. The exact product is enclosed by all four slanted straight lines, with piece-wise linear concave boundaries.

the coefficient. This error estimate should be multiplied by the size of the basic Chebyshev polynomials in the term but due to the restriction to the unit domain, this value is always within $[-1, 1]$.

To round downwards, all we need to change is the addition symbol preceding the summation operator. Instead of an upwards rounded plus, it should be a downwards rounded minus. Everything else can stay the same, except that a better bound is usually obtained when the coefficients are summed rounding *upwards* instead of downwards.

3.3 Multiplication

PI multiplication is much more complicated than addition even before any coefficient rounding errors are considered. This is caused by the fact that, unlike with addition, we need to consider *both* upper and lower bounds of both operands to determine the upper bound of the product. This is illustrated using a simple example in Figure 5. Notice that with multiplication it is important that we use independent polynomials for lower and upper bounds. If we were to use a single polynomial with a radius, we would need to use a much worse function interval in Figure 5, most likely the constant interval $[-2, 2]$. In this example the difference in the two bounds is magnified due to the fact that both operand enclosures cross zero. Even when they do not cross zero, there is an advantage to separated bounds. For example, multiplying $[x, x+1] \cdot [1, 2]$ with $x \in [1, 2]$ gives us a trapezoid enclosure $[x, 2x+2]$, which is much better than a parallelogram enclosure such as $1.5x + [-1, 3]$.

Formally, we define outwards-rounded generalised interval multiplication (denoted $\circledast^{\downarrow}$) as follows:

$$\begin{aligned}
&[p_{\mathrm{L}}, p_{\mathrm{R}}] \circledast^{\downarrow} [q_{\mathrm{L}}, q_{\mathrm{R}}] = \\
&\qquad [\min\nolimits_{\mathrm{L}} (p_{\mathrm{L}} \circledast_{\mathrm{L}} q_{\mathrm{L}}, p_{\mathrm{L}} \circledast_{\mathrm{L}} q_{\mathrm{R}}, p_{\mathrm{R}} \circledast_{\mathrm{L}} q_{\mathrm{L}}, p_{\mathrm{R}} \circledast_{\mathrm{L}} q_{\mathrm{R}}), \\
&\qquad\ \max\nolimits_{\mathrm{R}} (p_{\mathrm{L}} \circledast_{\mathrm{R}} q_{\mathrm{L}}, p_{\mathrm{L}} \circledast_{\mathrm{R}} q_{\mathrm{R}}, p_{\mathrm{R}} \circledast_{\mathrm{R}} q_{\mathrm{L}}, p_{\mathrm{R}} \circledast_{\mathrm{R}} q_{\mathrm{R}})]
\end{aligned}$$

and note that as multiplication increases the degree of the polynomials, we usually apply appropriate degree reduction for both bounds of the above interval. To complete the definition, we need to say how individual polynomials are multiplied, rounding point-wise upwards or downwards and, more crucially, how we estimate the minimum and maximum of multiplied polynomials as a polynomial.

Let us start with the easier part, i. e. correctly rounded multiplication of polynomials. Polynomial multiplication is worked out in a similar way as addition of polynomials, except that

$$
\begin{aligned}
&\sum_{0\le|j|\le m}\left(a_j\cdot\prod_{1\le i\le n}T_{j_i}(x_i)\right)\cdot\sum_{0\le|j|\le m}\left(b_j\cdot\prod_{1\le i\le n}T_{j_i}(x_i)\right)\\
&=\sum_{\substack{0\le|j'|\le m\\0\le|j''|\le m}}\left(a_{j'}\cdot b_{j''}\cdot\prod_{1\le i\le n}\left(T_{j'_i}(x_i)\cdot T_{j''_i}(x_i)\right)\right) &&\text{(distributive law)}\\
&=\sum_{\substack{0\le|j'|\le m\\0\le|j''|\le m}}\left(a_{j'}\cdot b_{j''}\cdot\prod_{1\le i\le n}\left(\frac{T_{j'_i+j''_i}(x_i)+T_{|j'_i-j''_i|}(x_i)}{2}\right)\right) &&\text{(by equation (7))}\\
&=\sum_{0\le|j|\le 2m}\left(\left(\sum_{\substack{0\le|j'|\le m\\0\le|j''|\le m\\j'\pm j''\ni j}}\left(a_{j'}\cdot b_{j''}\right)\right)\cdot\prod_{1\le i\le n}T_{j_i}(x_i)\right) &&\begin{array}{l}\text{(distributive law,}\\\text{grouping pairs of indices that yield}\\\text{the same index vector } j \text{ by some}\\\text{combination of addition or difference)}\end{array}
\end{aligned}
$$

where $j'\pm j''=\left\{j\,\middle|\,(\forall i)(j_i=j'_i+j''_i \text{ or } j_i=|j'_i-j''_i|)\right\}$

Fig. 6: Multiplication of multivariate polynomials in Chebyshev basis.

it turns out to be a bit more complex as one has to pairwise multiply all terms using the rule:

$$T_i(x)\cdot T_j(x)=T_{i+j}(x)/2+T_{|i-j|}(x)/2. \tag{7}$$

An upwards rounded multiplication of polynomials is derived from the formula for polynomial multiplication in exact arithmetic, shown in Figure 6. We use a rounded version of this formula to work out the coefficients of the rounded polynomial product $P_m(a_{j'})\circledast_{\mathrm{R}}P_m(b_{j''})$. Analogously to Figure 6, the coefficients c_j of this product are computed as:

$$c_j=\sum\nolimits_{\mathrm{L}}\limits_{\substack{0\le|j'|\le m\\0\le|j''|\le m\\j'\pm j''\ni j}}\left(a_{j'}\circledast_{\mathrm{L}}b_{j''}\right)\qquad\text{for } j\ne(0,\dots,0).$$

The constant term coefficient $c_{(0,\dots,0)}$ is first computed using the same formula as above, which turns into:

$$\sum\nolimits_{\mathrm{L}}\limits_{0\le|j'|\le m,}\left(0.5\circledast_{\mathrm{L}}a_{j'}\circledast_{\mathrm{L}}b_{j'}\right)$$

when specialised for $j=(0,\dots,0)$. To compensate for the rounding during the computation of each term j, we add to the constant term a number e_j for each j:

$$c_{(0,\dots,0)}=\left(\sum\nolimits_{\mathrm{L}}\limits_{0\le|j'|\le m,}\left(0.5\circledast_{\mathrm{L}}a_{j'}\circledast_{\mathrm{L}}b_{j'}\right)\right)\oplus_{\mathrm{R}}\sum\nolimits_{\mathrm{R}}\limits_{0\le|j|\le 2m}e_j$$

where each e_j is computed as follows:

$$\sum\nolimits_{\mathrm{R}}\limits_{\substack{0\le|j'|\le m\\0\le|j''|\le m\\j'\pm j''\ni j}}\left(0.5\circledast_{\mathrm{R}}(a_{j'}\circledast_{\mathrm{E}}b_{j''})\right)\oplus_{\mathrm{R}}\sum\nolimits_{\mathrm{R}}\limits_{\substack{0\le|j'|\le m\\0\le|j''|\le m\\j'\pm j''\ni j}}\left(0.5\circledast_{\mathrm{E}}(a_{j'}\circledast_{\mathrm{L}}b_{j''})\right)\oplus_{\mathrm{R}}\sum\nolimits_{\mathrm{E}}\limits_{\substack{0\le|j'|\le m\\0\le|j''|\le m\\j'\pm j''\ni j}}\left(0.5\circledast_{\mathrm{L}}a_{j'}\circledast_{\mathrm{L}}b_{j''}\right)$$

This concludes the description of how a safe upper bound on the product of two polynomials in the Chebyshev basis is obtained. The lower bound is obtained dually, i. e. all imprecise operations round upwards instead of downwards, the error term is calculated almost identically but is subtracted from the constant term instead of adding it to it.

3.4 Maximum and minimum

Both maximisation and minimisation are reduced to a maximisation of the type $\max_{\mathrm{R}}(0, p)$ or $\max_{\mathrm{L}}(0, p)$ as follows:

$$\max_{\mathrm{R}}(p,q) = q \oplus_{\mathrm{R}} \max_{\mathrm{R}}(0, p \ominus_{\mathrm{R}} q) \qquad \min_{\mathrm{R}}(p,q) = p \ominus_{\mathrm{R}} \max_{\mathrm{L}}(0, p \ominus_{\mathrm{R}} q)$$

$$\max_{\mathrm{L}}(p,q) = q \oplus_{\mathrm{L}} \max_{\mathrm{L}}(0, p \ominus_{\mathrm{L}} q) \qquad \min_{\mathrm{L}}(p,q) = p \ominus_{\mathrm{L}} \max_{\mathrm{L}}(0, p \ominus_{\mathrm{L}} q)$$

Assume that we know that the range of p is the interval $[a, b]$. When $[a, b]$ does not contain 0, $\max_{\mathrm{L}}(0, p)$ and $\max_{\mathrm{R}}(0, p)$ are either 0 or p. When it does contain zero, we compose p with a cubic univariate polynomial p_{C} that approximates $\max(0, x)$ on $[a, b]$, also shifting the polynomial so that it is fully above or fully below $\max(0, x)$, depending on whether we are computing upper or lower bound. The polynomial p_{C} is uniquely determined by the conditions $p_{\mathrm{C}}(a) = 0$, $p_{\mathrm{C}}'(a) = 0$, $p_{\mathrm{C}}(b) = b$, $p_{\mathrm{C}}'(b) = 1$ where p_{C}' is the derivative of p_{C}.

To safely evaluate $p_{\mathrm{C}}(p)$, one needs interval multiplication, which, in turn, requires maximisation. To break this cycle, we use for $p_{\mathrm{C}}(p)$ a crude version of multiplication, denoted $\circledast^*$. Instead of maximisation, this multiplication wraps the result with a cheap upper bound on the width of the exact enclosure multiplication $q \cdot [p_{\mathrm{L}}, p_{\mathrm{R}}]$. This upper bound is:

$$w = (|q| \circledast_{\mathrm{R}} |p_{\mathrm{R}} - p_{\mathrm{L}}|) \oplus_{\mathrm{R}} (q \circledast_{\mathrm{E}} p_{\mathrm{L}}) \oplus_{\mathrm{R}} (q \circledast_{\mathrm{E}} p_{\mathrm{R}})$$

where $|q|$ is an upper bound on the absolute value of the values of the polynomial q on its domain. Using w, the crude multiplication is defined as follows:

$$q \circledast^* [p_{\mathrm{L}}, p_{\mathrm{R}}] = [(q \circledast_{\mathrm{R}} p_{\mathrm{L}}) - w, (q \circledast_{\mathrm{L}} p_{\mathrm{L}}) + w].$$

3.5 Division

Division is treated as a composition of a product and a reciprocal. The reciprocal is usually not defined for 0. Whenever the interval $[p_{\mathrm{L}}, p_{\mathrm{R}}]$ contains zero for at least one point in the domain, we return the special "no information" interval $[-\infty, \infty]$. Since it is computationally expensive to determine whether zero can be avoided, we do it for a degree 1 (affine) reduction of the interval $[p_{\mathrm{L}}, p_{\mathrm{R}}]$ and thus sometimes return $[-\infty, \infty]$, even though a better approximation exists and could be computed.

Assume we know that $[p_{\mathrm{L}}, p_{\mathrm{R}}]$ is positive or negative throughout its domain. Then the reciprocal is antitonic and thus we have $1/[p_{\mathrm{L}}, p_{\mathrm{R}}] \subseteq [1 \oslash_{\mathrm{L}} p_{\mathrm{R}}, 1 \oslash_{\mathrm{R}} p_{\mathrm{L}}]$. It remains to find a way to approximate the reciprocal of a polynomial from above and from below by a polynomial of a bounded degree. We first show how to do this without consideration of rounding errors.

Assume we want to approximate ${}^1/_p$ for a n-degree polynomial p by a m degree polynomial q. Clearly we cannot find an approximation q with $p \cdot q = 1$, i. e. there is no perfect approximation. Our approach is to take a good polynomial approximation g of ${}^1/_x$ and substitute p into it and reduce its degree.[5] We transform the equation to $x \cdot g = 1$ and relax it to

$$x \cdot g(x) = 1 + c \cdot T_{m+1}(x)$$

so that it is solvable. We choose the Chebyshev polynomial as the remainder in order to distribute the error as evenly as possible modulo the weight $1/x$, which means that errors grow near 0 as expected. The $m + 1$ coefficients of g and c are computed as the solution of a fairly simple regular system of $m + 2$ linear equations (one per degree term in the Chebyshev basis) that can be solved in linear time using the τ method. To get an upper or lower bound we need to shift g by $c/2$ up or down. Finally, we perform these operations with the appropriate rounding mode to obtain a safe approximation despite using a rounded arithmetic.

3.6 Exponentiation and square root

In essence, exponentiation of a PI is performed by carefully applying the polynomial bounds to the standard Taylor expansion of e^x up to a certain degree, as well as an appropriate estimate of the remainder. To reduce the size of the remainder, the polynomial is first transformed using a linear transformation of the form $({}^x/_{2^k}) + c$ so that its range lies in an area where the Taylor series converges relatively fast, i. e. near 0. The result of the Taylor expansion is then transformed back using the exponentiated inverse x^{2^k}/e^c.

One important aspect of this algorithm is in the choice of the size of the Taylor expansion. As is the case with ordinary imprecise arithmetic, when raising the size, at some point the rounding errors start dominating the extra precision gained by the extra terms. It is a subject of further work to conduct a detailed analysis to determine optimal Taylor degrees for various situations. Currently, the degree has to be passed as an additional parameter to the PI exponentiation operator.

The square root is computed in a similar manner, except that instead of terms of the Taylor expansion we use a number of iterations of the Newton method for the equation ${}^1/_{s^2} - p = 0$ to compute $s = {}^1/\sqrt{p}$ and then invert it. The advantage of this method is that the iterations do not involve polynomial division. Nevertheless, we need to carefully transform the PI to make it fit into the stable range of this Newton iteration.

3.7 Integration

Integration of PIs is fairly straightforward using the standard equation:

$$\int_0^x T_n(t)\,\mathrm{d}t = \frac{1}{2}\left(\frac{T_{n+1}(x) - T_{n+1}(0)}{n+1} - \frac{T_{n-1}(x) - T_{n-1}(0)}{n-1}\right),$$

which is an analogy in Chebyshev basis of the standard formula $\int_0^x t^n\,\mathrm{d}t = x^{n+1}/(n+1)$. As with the other operations, extra care has been taken to ensure that rounding is always uniformly upwards or uniformly downwards on the whole domain of the function.

[5] The actual implementation includes further optimisations.

4 Numerical Prover Algorithm

```
input $\varphi, \boldsymbol{x}_{init}$  -- formula and the domain of its variables
$Q \leftarrow singleton(\boldsymbol{x}_{init})$  -- create a queue containing a single element
while $nonempty(Q)$ do
  $\boldsymbol{x} \leftarrow dequeue(Q)$  -- remove the next element from the queue
  if $[\![\varphi]\!]_{\boldsymbol{x}} = \mathtt{f}$ then
    return $false$  -- formula disproved
  if $[\![\varphi]\!]_{\boldsymbol{x}} = \mathtt{u}$ then  -- failed to decide formula
    $\boldsymbol{x}_L, \boldsymbol{x}_R \leftarrow splitLargestFirst(\boldsymbol{x})$  -- split the domain in half
    $enqueue(Q, \boldsymbol{x}_L, \boldsymbol{x}_R)$  -- add both sub-domains to the queue
return $true$  -- formula proved
```

Fig. 7: Top-level algorithm to decide the formula $\forall x_1 \in \boldsymbol{x}_1, \ldots, x_n \in \boldsymbol{x}_n : \varphi$.

Pseudo-code for the main loop of our prover is shown in Figure 7. The algorithm is based on domain subdivision and safe numeric approximation, implementing a straightforward translation of the semantics of interval expressions and relations that has been developed in the preceding sections. The domains are subdivided using the largest-first splitting strategy we introduced at the end of Subsection 2.5. The algorithm takes as input a formula φ and a domain $\boldsymbol{x}_{init} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ over which it is to decide the formula. We equate interval tuples such as $\boldsymbol{x}_{init}$ with their Cartesian products, so that they can be seen as hyper-rectangles within $\mathbb{R}^n$. We use the term *box* for such a hyper-rectangle.

The formal grammar for formulas is presented in Section 4.1 and the valuation, or semantics, $[\![\varphi]\!]_{\boldsymbol{x}}$ of a formula φ over the box $\boldsymbol{x}$ is defined in Section 4.2.

4.1 Correctness theorem language

In [18, 19] we outlined an approach to the specification of FP programs using interval expressions to approximate the FP values computed by the program. An annotated language based on WHILE with procedures [29] and a predicate transformer type logical semantics were given in [16]. The semantics translates the annotated program to a correctness theorem, which is a set of verification conditions (VCs). A proof that the generated VCs imply an intuitive notion of correctness given in terms of operational semantics was also provided in [16]. We will not address the proving of formulas with nested quantifiers, rather we will be proving propositional formulas closed by outermost universal quantification. We also restrict our attention to the functions and relations appearing in the example programs `erf` and `sqrt` given in Sections 5.1 and 5.2, respectively. Thus, the grammars for *numeric terms* τ and *formulas* φ are given by:

$$\begin{aligned}\tau ::= {} & q \mid e \mid \pi \mid x \mid [\tau_1, \tau_2] \mid -\tau \mid \sqrt{\tau} \mid \tau_1 + \tau_2 \mid \tau_1 * \tau_2 \mid \tau_1/\tau_2 \mid \tau_1^{\tau_2} \mid \int_{x \in [\tau_1, \tau_2]} \tau_3 \mid \\ & \varepsilon_{\text{abs}} \mid \varepsilon_{\text{rel}} \mid \boldsymbol{\varepsilon}_{abs} \mid \boldsymbol{\varepsilon}_{rel} \mid \mathtt{exp}(\tau) \mid \tau_1 \oplus \tau_2 \mid \tau_1 \ominus \tau_2 \mid \tau_1 \circledast \tau_2 \mid \tau_1 \oslash \tau_2 \\ \varphi ::= {} & \top \mid \bot \mid \tau_1 < \tau_2 \mid \tau_1 \leq \tau_2 \mid \tau_1 = \tau_2 \mid \tau_1 \geq \tau_2 \mid \tau_1 > \tau_2 \mid \tau_1 \in \tau_2 \mid \tau_1 \subseteq \tau_2 \mid \\ & \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \rightarrow \varphi_2\end{aligned}$$

where $q \in \mathbb{Q}$ and the circled operations $\oplus, \ominus, \circledast$, and $\oslash$ denote rounded versions of the usual numeric operations of addition, subtraction, multiplication, and division, respectively. Correctness theorems are formulas φ in the grammar above with all their free variables closed by a universal quantifier. The domain of each variable is obtained from the FP format in the variable declaration or from a precondition. Thus, correctness theorems have the form:

$$\forall x_1 \in \boldsymbol{x}_1 \cdots \forall x_n \in \boldsymbol{x}_n \,.\, \varphi \tag{8}$$

where $\boldsymbol{x} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ is the domain box of the theorem and all the free variables in φ are among $\{x_1, \ldots, x_m\}$. We call the language of formulas of the form (8) the *correctness theorem* (CT) language. The following section describes how the framework for safe approximation of exact numerical, FP, and Boolean expressions described previously is applied to obtain safe approximations for CT formulas.

4.2 Exact and approximate semantics for CT

The *exact* semantics of the grammar given on the preceding page is expressed in terms of *safe function interval approximations* of predicates. The exact semantics, used to approximate FP expressions, is in turn approximated in the prover by means of *safe* PI *approximations*, as defined in the preceding section. Formally, the *exact* interpretation of a formula φ is given by the map $[\![\cdot]\!] : \mathrm{CT} \to \mathbb{B}_\perp$:

$$[\![\forall x_1 \in \boldsymbol{x}_1 \cdots \forall x_m \in \boldsymbol{x}_m \,.\, \varphi]\!] \;=\; [\![\varphi]\!]_{\boldsymbol{x}} \tag{9}$$

where $\boldsymbol{x} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m)$, and $[\![\varphi]\!]_{\boldsymbol{x}}$ is defined recursively over the grammar of CT as follows:

$$\begin{aligned} [\![\varphi_1 \; c \; \varphi_2]\!]_{\boldsymbol{x}} &= \mathcal{B}_c([\![\varphi_1]\!]_{\boldsymbol{x}}, [\![\varphi_2]\!]_{\boldsymbol{x}}) \\ [\![\neg\varphi]\!]_{\boldsymbol{x}} &= \mathcal{B}_\neg([\![\varphi]\!]_{\boldsymbol{x}}) \end{aligned} \tag{10}$$

for Boolean connectives $c \in \{\wedge, \vee, \rightarrow\}$ with $\mathcal{B}_\neg$ and $\mathcal{B}_c$ given by safe approximations of $\neg$ and c, respectively. The *inclusion* relation $\subseteq$ and the *membership* relation $\in$ both have *exact* semantics $\mathcal{B}_\subseteq = \mathcal{B}_\in$ given by the *reverse refinement* relation $\sqsupseteq$ on function intervals:

$$[\![\tau_1 \subseteq \tau_2]\!]_{\boldsymbol{x}} \;=\; [\![\tau_1 \in \tau_2]\!]_{\boldsymbol{x}} \;=\; [\![\tau_2]\!]_{\boldsymbol{x}} \sqsubseteq [\![\tau_1]\!]_{\boldsymbol{x}} \tag{11}$$

and are *approximated* in terms of PIs by:

$$[\![\tau_2]\!]_{\boldsymbol{x}} \sqsubseteq [\![\tau_1]\!]_{\boldsymbol{x}} \;\gtrsim\; \begin{cases} \mathtt{t} \text{ if } [\![\tau_2]\!]_{\boldsymbol{x}}^{\mathrm{PI}\uparrow} \sqsubseteq^{\mathrm{PI}} [\![\tau_1]\!]_{\boldsymbol{x}}^{\mathrm{PI}\downarrow} \\ \mathtt{f} \text{ if } [\![\tau_2]\!]_{\boldsymbol{x}}^{\mathrm{PI}\downarrow} \not\sqsubseteq^{\mathrm{PI}} [\![\tau_1]\!]_{\boldsymbol{x}}^{\mathrm{PI}\uparrow} \\ \perp \text{ otherwise} \end{cases} \tag{12}$$

where the inner and outer PI semantics $[\![\tau]\!]_{\boldsymbol{x}}^{\mathrm{PI}\uparrow}$ and $[\![\tau]\!]_{\boldsymbol{x}}^{\mathrm{PI}\downarrow}$ of terms are described below, and $\sqsubseteq^{\mathrm{PI}}$ and $\not\sqsubseteq^{\mathrm{PI}}$ are safe PI approximations of $\sqsubseteq$ and $\not\sqsubseteq$, respectively. Order relations $r \in \{<, \leq, =, \geq, >\}$ and Boolean literals have *exact* semantics $\mathcal{B}_r, \mathcal{B}_\perp$, and $\mathcal{B}_\top$ given by:

$$\begin{aligned} [\![\tau_1 \; r \; \tau_2]\!]_{\boldsymbol{x}} &= [\![\tau_1]\!]_{\boldsymbol{x}} \, r_{\mathbb{I}} \, [\![\tau_2]\!]_{\boldsymbol{x}} \\ [\![\perp]\!]_{\boldsymbol{x}} &= \mathtt{f} \\ [\![\top]\!]_{\boldsymbol{x}} &= \mathtt{t} \end{aligned} \tag{13}$$

where the interval relation $r_{\mathbb{I}}$ corresponding to r is *approximated* by:

$$[\![\tau_1]\!]_{\boldsymbol{x}}\, r_{\mathbb{I}}\, [\![\tau_2]\!]_{\boldsymbol{x}} \;\gtrsim\; \begin{cases} \mathtt{t} \text{ if } [\![\tau_1]\!]_{\boldsymbol{x}}^{\text{PI}\downarrow}\, r_{\mathbb{I}}^{\text{PI}}\, [\![\tau_2]\!]_{\boldsymbol{x}}^{\text{PI}\downarrow} \\ \mathtt{f} \text{ if } [\![\tau_1]\!]_{\boldsymbol{x}}^{\text{PI}\downarrow}\, (\neg r)_{\mathbb{I}}^{\text{PI}}\, [\![\tau_2]\!]_{\boldsymbol{x}}^{\text{PI}\downarrow} \\ \bot \text{ otherwise} \end{cases} \tag{14}$$

where $r_{\mathbb{I}}^{\text{PI}}$ is a safe PI approximation of the interval relation $r_{\mathbb{I}}$ and $\neg r$ is the *strict opposite* version of the exact relation r, with *e.g.* $\neg(\leq)$ being $>$.

The refinement and interval order relation approximations in (12) and (14) use the *outer* PI *semantics* $[\![\tau]\!]_{\boldsymbol{x}}^{\text{PI}\downarrow}$. The *exact* outer semantics $\mathcal{B}_{\circledcirc}^{\downarrow}$ and $\mathcal{B}_{\mathtt{exp}}^{\downarrow}$ of *floating-point* operations and functions is given by the generalised outward rounding operator $\mathcal{S}_{out}$ from Section 2.2. It approximates FP expressions from below by pessimistically assuming that maximal rounding errors arise from each constituent FP function. The exact outer semantics $[\![\tau]\!]_{\boldsymbol{x}}^{\downarrow}$ is *approximated* from below by the outer PI semantics $[\![\tau]\!]_{\boldsymbol{x}}^{\text{PI}\downarrow}$ defined for FP terms τ by:

$$\begin{aligned} [\![\tau_1 \circledcirc \tau_2]\!]_{\boldsymbol{x}}^{\text{PI}\downarrow} &= \mathcal{B}_{\circledcirc}^{\text{PI}\downarrow}([\![\tau_1]\!]_{\boldsymbol{x}}^{\text{PI}\downarrow}, [\![\tau_1]\!]_{\boldsymbol{x}}^{\text{PI}\downarrow}) \\ [\![\mathtt{exp}(\tau)]\!]_{\boldsymbol{x}}^{\text{PI}\downarrow} &= \mathcal{B}_{\mathtt{exp}}^{\text{PI}\downarrow}([\![\tau]\!]_{\boldsymbol{x}}^{\text{PI}\downarrow}) \\ [\![\boldsymbol{\varepsilon}]\!]_{\boldsymbol{x}}^{\text{PI}\downarrow} &= [\lambda x. - \varepsilon, \lambda x.\varepsilon] \\ [\![\varepsilon]\!]_{\boldsymbol{x}}^{\text{PI}\downarrow} &= \lambda x.\varepsilon \end{aligned} \tag{15}$$

where $\circledcirc \in \{\oplus, \ominus, \circledast, \oslash\}$, $\boldsymbol{\varepsilon} \in \{\boldsymbol{\varepsilon}_{abs}, \boldsymbol{\varepsilon}_{rel}\}$ and $\varepsilon \in \{\varepsilon_{\text{abs}}, \varepsilon_{\text{rel}}\}$. $\mathcal{B}_{\circledcirc}^{\text{PI}\downarrow}$ and $\mathcal{B}_{\mathtt{exp}}^{\text{PI}\downarrow}$ are given in terms of the natural outer PI extension $\mathcal{S}_{out}^{\text{PI}\downarrow}$ of the exact operator $\mathcal{S}_{out}$:

$$\begin{aligned} \mathcal{B}_{\circledcirc}^{\text{PI}\downarrow}(\boldsymbol{f}_1, \boldsymbol{f}_2) &= \mathcal{S}_{out}^{\text{PI}\downarrow}\Big([\![\varepsilon_{\circledcirc}]\!]_{\boldsymbol{x}}^{\text{PI}\downarrow}, \boldsymbol{f}_1 \circ^{\text{PI}\downarrow} \boldsymbol{f}_2\Big) \\ &= (1 +^{\text{PI}\downarrow} [-\varepsilon_{\text{rel}}, \varepsilon_{\text{rel}}]) *^{\text{PI}\downarrow} \Big((1 +^{\text{PI}\downarrow} [\![\varepsilon_{\circledcirc}]\!]_{\boldsymbol{x}}^{\text{PI}\downarrow} *^{\text{PI}\downarrow} [-1, 1]) \\ &\quad *^{\text{PI}\downarrow}(\boldsymbol{f}_1 \circ^{\text{PI}\downarrow} \boldsymbol{f}_2)\Big) +^{\text{PI}\downarrow} [-\varepsilon_{\text{abs}}, \varepsilon_{\text{abs}}] \end{aligned} \tag{16}$$

where $\circ \in \{+, -, *, /\}$ is the exact operation corresponding to $\circledcirc$ and $\circ^{\text{PI}\downarrow}$ is an *isotonic outer* PI *extension* of $\circ$ in the sense of Sections 2 and 3 and:

$$\mathcal{B}_{\mathtt{exp}}^{\text{PI}\downarrow}(\boldsymbol{f}) \;=\; \mathcal{S}_{out}^{\text{PI}\downarrow}\Big([\![\varepsilon_{\mathtt{exp}}]\!]_{\boldsymbol{x}}^{\text{PI}\downarrow}, \mathcal{B}_{e}^{\text{PI}\downarrow}(\boldsymbol{f})\Big) \tag{17}$$

where $\mathcal{B}_{e}^{\text{PI}\downarrow}$ is an isotonic outer interval extension of the extended real exponential function. In particular, the exact outer semantics approximates *FP functions* over an interval box $\boldsymbol{x}$ with a function interval in $\mathbb{J}(\boldsymbol{x} \to \mathbb{R}^{\pm\infty})$, where $\mathbb{R}^{\pm\infty}$ denotes the poset of *two-point extended reals* $\mathbb{R} \cup \{-\infty, +\infty\}$, with the ordering on $\mathbb{R}$ extended in the obvious way.

The *exact* interpretation $\mathcal{B}_{\gamma}$ of *exact* functions γ is given by the corresponding interval function and *approximated* in terms of PI extensions as follows:

$$\begin{aligned} [\![\gamma(\tau_1, \ldots, \tau_k)]\!]_{\boldsymbol{x}}^{\text{PI}\downarrow} &= \mathcal{B}_{\gamma}^{\text{PI}\downarrow}([\![\tau_1]\!]_{\boldsymbol{x}}^{\text{PI}\downarrow}, \ldots, [\![\tau_k]\!]_{\boldsymbol{x}}^{\text{PI}\downarrow}) \\ \Big[\!\!\Big[\int_{x' \in [\tau_1, \tau_2]} \tau_3\Big]\!\!\Big]_{\boldsymbol{x}}^{\text{PI}\downarrow} &= \Big(\mathcal{B}_{\int}^{\text{PI}\downarrow}([\![\tau_3]\!]_{\boldsymbol{x}'}^{\text{PI}\downarrow}) \circ_{i_{x'}}^{\text{PI}\downarrow} [\![\tau_2]\!]_{\boldsymbol{x}}^{\text{PI}\downarrow}\Big) -^{\text{PI}\downarrow} \Big(\mathcal{B}_{\int}^{\text{PI}\downarrow}([\![\tau_3]\!]_{\boldsymbol{x}'}^{\text{PI}\downarrow}) \circ_{i_{x'}}^{\text{PI}\downarrow} [\![\tau_1]\!]_{\boldsymbol{x}}^{\text{PI}\downarrow}\Big) \\ [\![[\tau_1, \tau_2]]\!]_{\boldsymbol{x}}^{\text{PI}\downarrow} &= \Big[\underline{[\![\tau_1]\!]_{\boldsymbol{x}}^{\text{PI}\downarrow}}, \overline{[\![\tau_2]\!]_{\boldsymbol{x}}^{\text{PI}\downarrow}}\Big] \\ [\![x]\!]_{\boldsymbol{x}}^{\text{PI}\downarrow} &= proj_{i_{\boldsymbol{x}}}(\boldsymbol{x}) \end{aligned} \tag{18}$$

where $\underline{[a,b]} = a$ and $\overline{[a,b]} = b$ denote the lower and upper bound of the interval $[a,b]$, respectively, and $\boldsymbol{x}'$ is the domain box obtained from $\boldsymbol{x}$ by adding the variable x' with the following domain:

$$[\![[\tau_1, \tau_2]]\!]^{\text{PI}\downarrow}_{\boldsymbol{x}} \downarrow 0 \tag{19}$$

and $\circ^{\text{PI}\downarrow}_{i_{x'}}$ denotes *outer* PI *approximation* of *composition* in the variable $i_{x'}$. The outer approximations $\mathcal{B}^{\text{PI}\downarrow}_{\gamma}$ are given by isotonic outer PI extensions of the possibly exact functions $\gamma \in \{\sqrt{}, +, -, *, /, \hat{}, \int\}$, where $\hat{}$ denotes the binary exponentiation operator. The *exact* semantics of exact *literals* $l \in \{e, \pi\} \cup \mathbb{Q}$ is given by the embedding of real numbers within function intervals, $[\![l]\!]_{\boldsymbol{x}} = [\lambda x.l, \lambda x.l]$, and is *approximated* by *outward rounding*:

$$[\![l]\!]^{\text{PI}\downarrow}_{\boldsymbol{x}} \;=\; [\lambda x.l_{\text{L}}, \lambda x.l_{\text{R}}]. \tag{20}$$

Note that the above is compatible with the semantics for FP epsilons given in (15). Since the epsilons are *representable* numbers, their resulting intervals are *thin*.

The containment relation approximations in (12) use both the outer PI semantics $[\![\tau]\!]^{\text{PI}\downarrow}_{\boldsymbol{x}}$ and the *inner* PI semantics $[\![\tau]\!]^{\text{PI}\uparrow}_{\boldsymbol{x}}$ of terms. The latter is defined *dually* to the former. More specifically, a definition of the inner semantics is obtained from (15)–(20) by:

– reversing left and right interval bounds, *i. e.* $\underline{x}$ becomes $\overline{x}$ and vice versa, in the interval constructor equation in (18)
– reversing all downward arrows to upward arrows, except those in (19)
– replacing the isotonic outer interval extensions with *antitonic inner interval extensions*
– using the *conjugate* of the *outward* semantics for interval FP literals:

$$[\![\varepsilon]\!]^{\uparrow}_{\boldsymbol{x}} = [\lambda x.\varepsilon, \lambda x. - \varepsilon] \tag{21}$$

– and replacing the outward rounding $\mathcal{S}_{out}$ by the following *inward rounding* variant:

$$\mathcal{S}_{in}(e, x) \;=\; (1 + [\varepsilon_{\text{rel}}, -\varepsilon_{\text{rel}}])(1 + e[1, -1])x + [\varepsilon_{\text{abs}}, -\varepsilon_{\text{abs}}] \tag{22}$$

4.3 Note on the approximation of $\int_a^b f(x)dx$

Equation (18) describes how outer PI approximations of definite integrals are obtained. First, the integration variable domain is approximated by an *outer interval* approximation of the ranges of the integration bound expressions, as described by (19). that a correct PI approximation of the primitive function is obtained. The domain box $\boldsymbol{x}$ is then extended by adding the integration variable and its domain, denoting the new box $\boldsymbol{x}'$, and an outer PI approximation of the primitive function of the integrand expression is constructed over $\boldsymbol{x}'$. Finally, in the resulting PI the integration variable is substituted with outer approximations of the two bound expressions, yielding two PIs over $\boldsymbol{x}$ and these two are subtracted rounding outwards.

For an *inner* approximation of the integral most operations are computed rounding inwards. Note though that there is one exception, namely that the approximation of the domain of the integration variable is computed with outer rounding so that the box $\boldsymbol{x}'$ certainly covers the range of the integral bound expressions.

Currently, the implementation only supports integration of expressions that lead to *monotone* primitive functions, due to a restriction on the substitution operator. We plan to remove this restriction in the future.

The implementation also provides a parameter for the integration operation, called *integration depth*, intended to provide a means of increasing approximation precision for computational effort. If the integration depth is set to *n*, then the integration variable domain is bisected *n* times and approximations of the integrand expression are computed over each sub-interval, effectively yielding a *piece-wise* PI approximation over the original domain. The piece-wise approximations are then safely combined over the range of each integration bound expression separately, potentially resulting in tighter approximations of the bounds with increasing depth.

In the following section we investigate how solving times for a particular theorem are affected by the *polynomial degree* and *integration depth* parameters. The expectation is that increasing either parameter should initially decrease solving times as improved precision facilitates earlier decisions, and then increase as redundant precision no longer facilitates earlier decisions while incurring additional cost.

5 Examples

This section shows in some detail how our method can be applied on specific verification problems. First, Subsections 5.1 and 5.2 focus on two small examples, namely `erf` given in Figure 8 and `sqrt` given in 11. One example has an integral operator and the other has an interval inclusion in its specification. Thus each of these two examples focuses on one of the two points of extended expressivity in contracts.

Subsection 5.3 provides some detail of solving the `erfRiemann` example from Figure 1 in Section 1. This example is somewhat more involved than the first two as it features both an integral and an inclusion in its specification and a loop in its code. Therefore, we do not document it to the same level of detail as the other two examples. We restrict the discussion to the choice of its loop invariant and state the most difficult VC generated for this program.

Subsection 5.4 provides details of how we tackled one of the more challenging examples from the ProVal project in order to facilitate a comparison with the state-of-the-art methods employed in this project.

Finally, Subsection 5.5 details a verification of a program that comprises multiple dependent sub-programs.

When proving some of the harder VCs, we probe the conjecture that proof obligations for FP properties benefit from higher-degree enclosures to reduce approximation error during solving. An informal argument for this conjecture is that since precision loss in the arithmetic may force subdivision, it follows that precision enhancement at sub-exponential cost may delay subdivision and yield overall reduced solving effort.

Along with VCs for tight functional properties, exception freedom VCs are also generated and checked.

PolyPaver attempts to *decide* a formula over the prescribed domain, meaning that it aims to either prove or disprove it. PolyPaver disproves a formula by identifying a counterexample in the shape of a box where the formula is false. To test this functionality, we simulate a specification error in Subsection 5.1.3.

The experiments in this section were conducted on an Intel Core2 Duo 1.86 GHz machine with 3GB RAM and 6MB cache, running Kubuntu 10.10, and timeouts were set to between one and 24 hours. Proving would also be halted when attempting to split an interval with neighbouring maximum granularity numbers as endpoints. PolyPaver was compiled with the Glasgow Haskell Compiler (GHC) version 6.12.3.

5.1 Verification of `erf`

The *Gaussian error function* is often used to evaluate the cumulative density function of a normally distributed random variable. It is given by the integral:

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt.$$

It is well known that this integral has no closed form solution.

```
procedure erf(in $x \in \mathbb{F}$; out $r \in \mathbb{F}$)
    pre $0 \leq x \ \wedge \ x \leq 4$
    post $\frac{2}{\sqrt{\pi}} \int_{t \in [0,x]} e^{-t^2} - 0.00005 \leq r \ \wedge \ r \leq \frac{2}{\sqrt{\pi}} \int_{t \in [0,x]} e^{-t^2} + 0.00005$
is
    $t_1, t_2, t_3 \in \mathbb{F}$
begin
    -- an encoding of a rational approximation of the error function, found in [2]:
    $t_1$ := $1.0 \oplus 0.47047 \circledast x$;
    $t_2$ := $t_1 \circledast t_1$;
    $t_3$ := $t_1 \circledast t_2$;
    $r$ := $1.0 \ominus$ exp$(-x \circledast x) \circledast (0.3480242 \oslash t_1 \ominus 0.0958798 \oslash t_2 \oplus 0.7478556 \oslash t_3)$;
end
```

Fig. 8: Implementation of the error function using a rational approximation.

Rather than computing the integral directly, the program `erf`, shown in Figure 8, uses a rational approximation. According to [2], this approximation has a maximal absolute error of less than $2.5 \cdot 10^{-5}$. When taking rounding into account, the approximation's error could increase. Therefore we specified the accuracy to be $\pm 5 \cdot 10^{-5}$.

5.1.1 Correctness theorem

The correctness theorem for `erf` is shown as separate VCs (23)–(36) below, with the variable x being universally quantified over the interval [0,4]:

$$0.47047 \circledast x \in \mathbb{F} \tag{23}$$

$$1.0 \oplus (0.47047 \circledast x) \in \mathbb{F} \tag{24}$$

$$T \circledast T \in \mathbb{F} \tag{25}$$

$$T \circledast (T \circledast T) \in \mathbb{F} \tag{26}$$

$$x \circledast x \in \mathbb{F} \tag{27}$$

$$\texttt{exp}(-x \circledast x) \in \mathbb{F} \tag{28}$$

$$0.3480242 \oslash T \in \mathbb{F} \tag{29}$$

$$0.0958798 \oslash (T \circledast T) \in \mathbb{F} \tag{30}$$

$$0.3480242 \oslash T \ominus 0.0958798 \oslash (T \circledast T) \in \mathbb{F} \tag{31}$$

$$0.7478556 \oslash (T \circledast (T \circledast T)) \in \mathbb{F} \tag{32}$$

$$0.3480242 \oslash T \ominus 0.0958798 \oslash (T \circledast T) \oplus 0.7478556 \oslash (T \circledast (T \circledast T)) \in \mathbb{F} \tag{33}$$

$$\texttt{exp}(-x \circledast x) \circledast S \in \mathbb{F} \tag{34}$$

$$1.0 \ominus \texttt{exp}(-x \circledast x) \circledast S \in \mathbb{F} \tag{35}$$

$$\frac{2}{\sqrt{\pi}} \int_{t\in[0,x]} e^{-t^2} - 0.00005 \leq 1.0 \ominus \texttt{exp}(-x \circledast x) \circledast S \leq \frac{2}{\sqrt{\pi}} \int_{t\in[0,x]} e^{-t^2} + 0.00005 \tag{36}$$

where

$$T = 1.0 \oplus (0.47047 \circledast x)$$
$$S = 0.3480242 \oslash T \ominus 0.0958798 \oslash (T \circledast T) \oplus 0.7478556 \oslash (T \circledast (T \circledast T))$$

and $a \leq b \leq c$ abbreviates $a \leq b \wedge b \leq c$.

5.1.2 Proof

The exception freedom VCs (23)–(35) were all proved within 10 seconds for all polynomial degrees. Minimal solving times were obtained for degree 0 intervals, for which all exception freedom VCs were proved within 10 ms. Higher degrees yielded monotonically increasing solving times, indicating that the additional precision and computational cost of higher-degree polynomials was wasted.

The solving times for the functional correctness VC (36) are presented in Figures 9a and 9b on page 28. Figure 9a shows how solving times depend on the chosen polynomial degree and integration depth. Integration depth is explained in the fourth paragraph of Subsection 4.3.

The missing data points in Figure 9a, for integration depths 0 and 1, correspond to solving being terminated due to an attempted splitting of an interval whose endpoints are neighbouring FP numbers, indicating that a prohibitive number of sub-problems may have to be evaluated to decide the formula over the initial domain. We conclude that, for the given parameters, the approximation precision is insufficient, forcing excessive splitting and leading to an exponential growth of the sub-problem queue.

Figure 9b explores in some detail what happens for the missing data points in Figure 9a. For some of these parameter combinations the theorem was not proved on the whole domain $x \in [0,4]$ but it was proved on a subdomain of the form $[0, x_{\max}]$. Figure 9b plots the proportion of the domain $[0,4]$ on which the theorem was proved. The figure clearly shows that the proportion increases monotonically with increasing degree and integration depth.

As expected, the proving durations shown in Figure 9a first decrease and then increase as the polynomial degree increases. The shape is explained in terms of two main sources of computational cost: average evaluation time per sub-problem and the number of computed sub-problems. Increasing the degree leads to increased computational effort and, at least initially, to improved precision. Improved precision leads to earlier decisions, reducing the total amount of generated sub-problems. As long as the number of generated sub-problems decreases faster than the average per-problem computation time, as functions of degree, then the solving time graph should decrease as maximum degree increases. Eventually, an optimal degree should be reached, after which additional degrees add comparatively little precision, while incurring additional computational cost, meaning the graph should level off and then start increasing. We also see that increasing the integration depth can have a beneficial effect, reducing the minimum polynomial degree for which the minimum solving time is achieved.

The conclusion drawn from the above example is that higher-degree polynomial enclosures can improve the overall performance of a bisection search algorithm, at least when compared to interval arithmetic using constant or low-degree polynomial bounds.

5.1.3 Counterexample discovery

To evaluate how PolyPaver performs at counterexample discovery, the postcondition was changed to model what was felt as a reasonable implementation mistake, namely the reversal of signs in a formula. The resulting false version of (36) is shown below.

$$\tfrac{2}{\sqrt{\pi}} \int_{t\in[0,x]} e^{-t^2} + 0.00005 \leq 1.0 \ominus \mathtt{exp}(-x \circledast x) \circledast S \leq \tfrac{2}{\sqrt{\pi}} \int_{t\in[0,x]} e^{-t^2} - 0.00005 \qquad (37)$$

The solving efforts for (37) are presented in Figures 10a and 10b, showing solving time and the number of computed sub-problems, respectively, as functions of degree. The solving time graphs in Figure 10a are similar to the ones in Figure 9a but shifted towards lower degrees. This is to be expected, since a counterexample can be identified anywhere, while proving requires searching the entire domain.

The shapes of the graphs are again roughly convex, reflecting the characteristics of the PolyPaver algorithm as described in the preceding section.

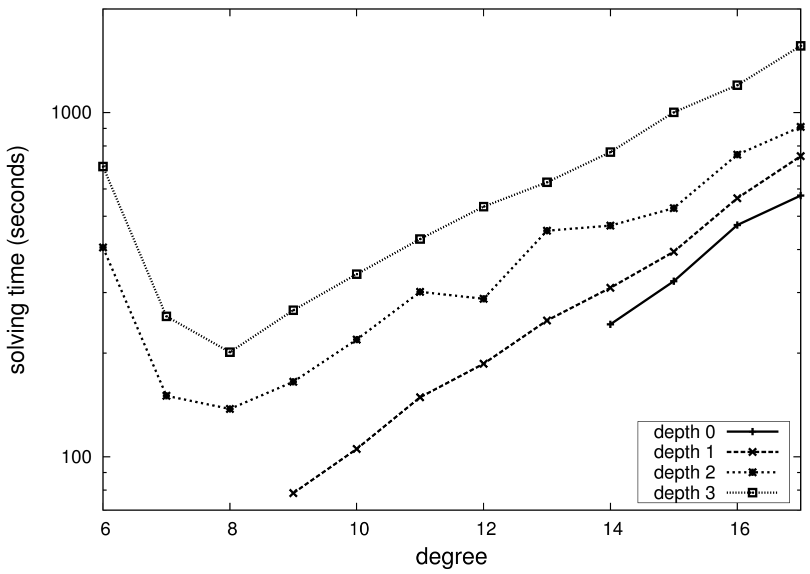
The number of computed sub-problems until counterexample identification are plotted in Figure 10b. Since the number of generated sub-problems depends on the approximation precision, we can interpret the graphs as showing the initial benefit of increasing approximation effort and with it precision. As approximation effort is increased further, there is a decrease in the improvement. At this point, added approximation effort yields little or no improvement and the additional effort translates to computational overhead.

5.2 Verification of `sqrt`

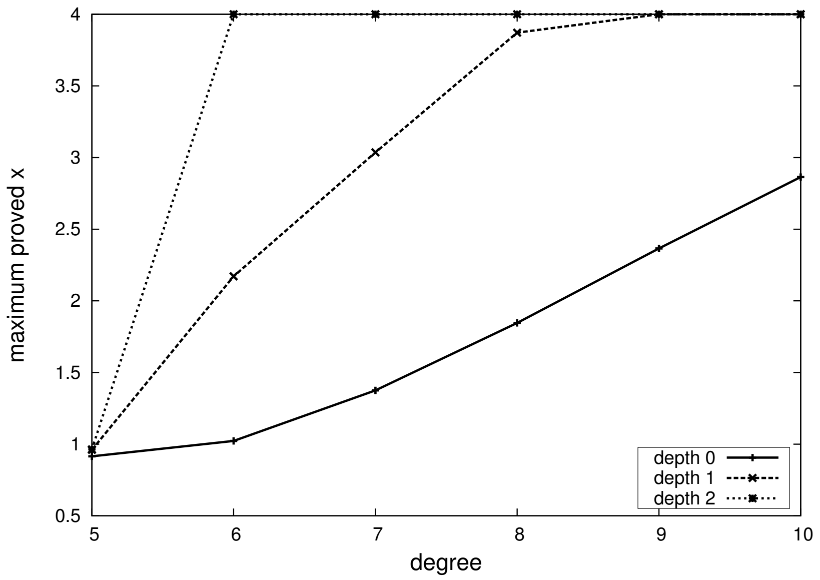
In this section we describe a program that features interval terms and interval inclusions in its specification. The chosen algorithm is an instance of Newton's algorithm generating a sequence of values converging to the square root of a real number $x_0 > 0$. The sequence is defined inductively as:

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{x_0}{x_n}\right) \qquad (38)$$

for non-negative integers n.

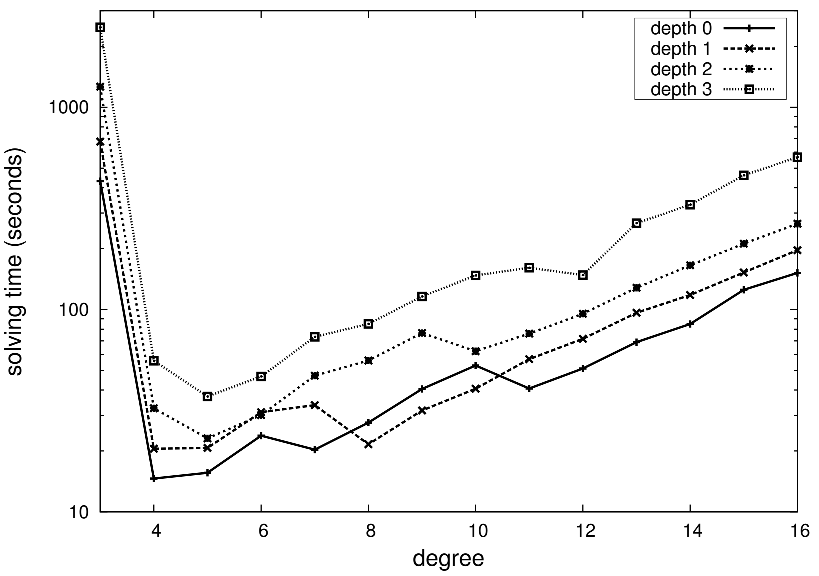


(a) The effect of polynomial degree on proving time for different integration depths.

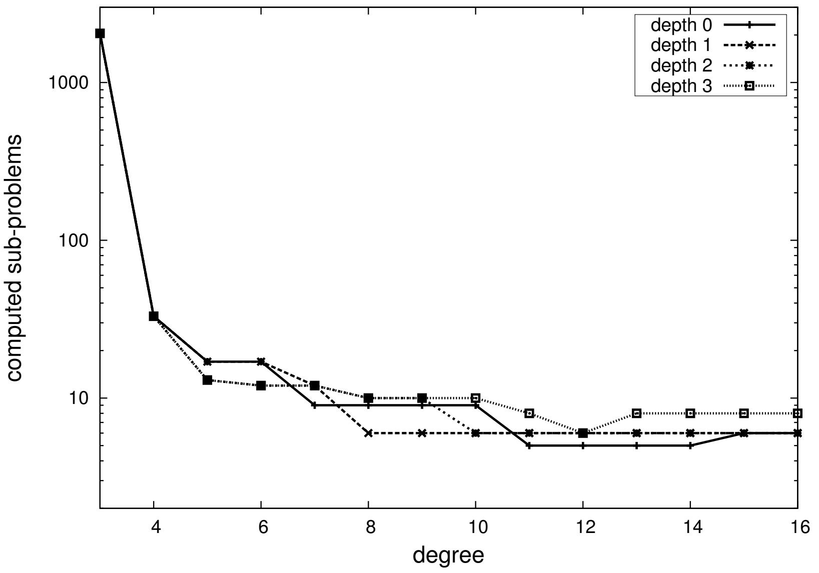


(b) The effect of polynomial degree on the fraction of problem domain for which proof was completed, for different integration depths.

Fig. 9: Performance of PolyPaver depending on polynomial degree and integration depth. Proving VC (36), derived from a correctly specified `erf` program.



(a) The effect of polynomial degree on counterexample discovery time for different integration depths.



(b) The effect of polynomial degree on the number of subproblems generated during counterexample discovery, for different integration depths.

Fig. 10: Performance of PolyPaver depending on polynomial degree and integration depth. Counterexample discovery for VC (37), derived from an incorrectly specified `erf` program.

```
procedure sqrt(in $x \in \mathbb{F}$; out $r \in \mathbb{F}$)
    pre   $x \in [0.5, 2]$  -- the input $x_0$ is restricted to a stable range
    post  $r \in (1 + 4\varepsilon_{rel})\sqrt{x}$  -- r is the square root of $x_0$ modulo a small relative error
is
    $s \in \mathbb{F}$  -- the previous value in the Newton iteration
begin
    $s$ := $x$;  -- initially, the previous value is $x_0$
    $r$ := $(0.5 \circledast x) \oplus 0.5$;  -- ... while the current value is $x_1$
    while $r \neq s$ assert $r \in [-\frac{x^2}{4} + x, \frac{x^2}{4} + 1]$ do
                                    -- the loop invariant provides bounds for r
        $s$ := $r$;  -- the current value becomes the previous value
        $r$ := $0.5 \circledast (s \oplus (x \oslash s))$;  -- the Newton step gives the new value
    od
end
```

Fig. 11: Implementation of the square root function.

5.2.1 Implementation

Figure 11 contains an implementation of the algorithm we just described and its specification. We have chosen to restrict the range for x in the precondition to eliminate the troublesome case for x near 0. This is perfectly reasonable, since any call to `sqrt` may be preceded with a conditioning step, where the values in the original range for x are multiplied with an appropriate power of 4, resulting in a call to `sqrt` with $2^{2k}x$ rather than x, for some integer k. The result will then lie near $2^k\sqrt{x}$, which is multiplied with 2^{-k} to obtain the sought approximation of $\sqrt{x}$. Note that when using FP arithmetic in base β, multiplication and division with powers of β is exact, given that no overflow or underflow occurs. Therefore the conditioning and normalisation steps introduce the least possible error.

Changing the input range for the program eliminates potential denormalised values for x, which makes it possible to have a postcondition without an absolute error term.

In order to propagate the positivity property for r through successive iterations of the loop, we need to provide a loop invariant encoding this information. As a first idea we may choose $r > 0$, but this choice does not help to rule out overflows during division. Another choice is $r \in [x, 2]$ which introduces a less obvious problem, namely that of placing the fixpoint of the loop body, occurring for $x = 1$ and $r = 1$, on the boundary of the loop invariant. The problem with this is that VCs covering computation in the loop require us to show that states satisfying the loop invariant are mapped to states that satisfy the loop invariant. With a fixpoint on the boundary, this statement leads to a formula with touching, which our approximation method cannot prove. The solution is to use a loop invariant that has no fixpoints on its boundary. A sufficient condition is that the support of the invariant is mapped *properly* into itself by the loop body. One such region is given by numbers r that are bounded from above by $\frac{x^2}{4} + 1$ and from below by $-\frac{x^2}{4} + x$, yielding the loop invariant $r \in [-\frac{x^2}{4} + x, \frac{x^2}{4} + 1]$.

Our final concern is to ensure that the initial values of r and s validate the post condition whenever they satisfy the loop exit condition, and that they otherwise validate the loop invariant. This is achieved by setting the initial value of s to x and that of r to $0.5 \circledast x \oplus 0.5$, effectively unrolling the first iteration of the loop.

5.2.2 Termination

The program in Figure 11 typically terminates because the exit condition of the loop is an equality that eventually holds thanks to the fact that the algorithm is stable and the numbers are rounded to a finite set of FP values. Nevertheless, our method currently does not verify termination as we target *partial* rather than *total correctness*. Instead, we *assume* termination, and seek to specify *accuracy* properties of the returned value.

It is possible to extend our method so that it can verify termination of loops. In such an extension it would be quite hard to deal with the exit condition from Figure 11 because an equality of FP expressions is very hard to prove using approximations.

The identification of classes of programs for which termination proofs may be tractable is an interesting and important research direction. Progress in this area could make it safe to employ natural expressions of algorithms, such as the one in Figure 11, in safety-critical applications.

5.2.3 Correctness theorem

The correctness theorem Vcs for `sqrt` are presented below:

$$0.5 \circledast x \oplus 0.5 \neq x \rightarrow 0.5 \circledast x \oplus 0.5 \in [-\frac{x^2}{4} + x, \frac{x^2}{4} + 1] \tag{39}$$

$$0.5 \circledast x \oplus 0.5 = x \rightarrow 0.5 \circledast x \oplus 0.5 \in (1 + 4\varepsilon_{rel}) \sqrt{x} \tag{40}$$

$$r \in [-\frac{x^2}{4} + x, \frac{x^2}{4} + 1] \rightarrow x \oslash r \in \mathbb{F} \tag{41}$$

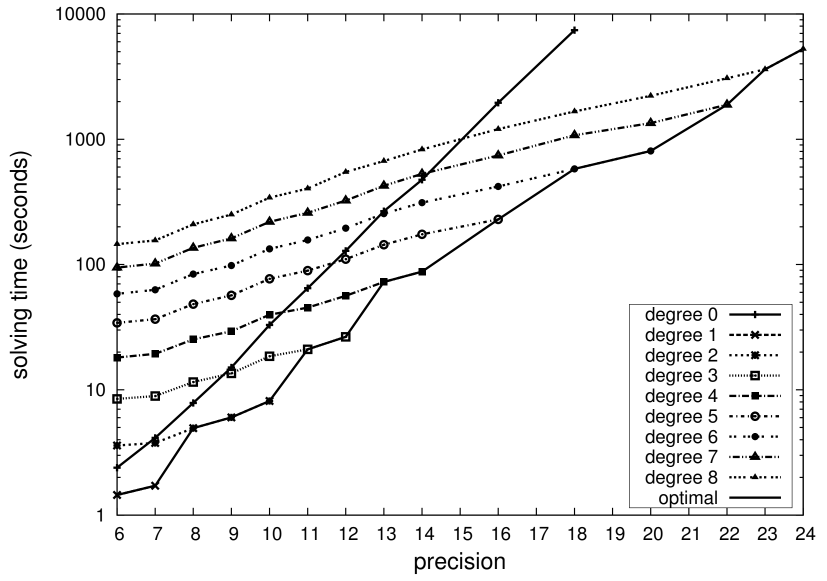
$$r \in [-\frac{x^2}{4} + x, \frac{x^2}{4} + 1] \rightarrow r \oplus (x \oslash r) \in \mathbb{F} \tag{42}$$

$$r \in [-\frac{x^2}{4} + x, \frac{x^2}{4} + 1] \rightarrow 0.5 \circledast (r \oplus (x \oslash r)) \in \mathbb{F} \tag{43}$$

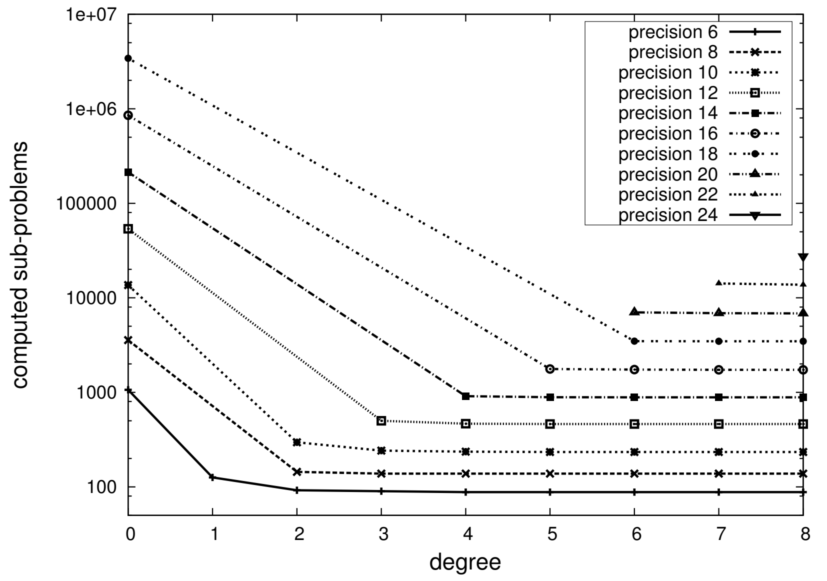
$$r \in [-\frac{x^2}{4} + x, \frac{x^2}{4} + 1] \wedge 0.5 \circledast (r \oplus (x \oslash r)) \neq r \rightarrow 0.5 \circledast (r \oplus (x \oslash r)) \in [-\frac{x^2}{4} + x, \frac{x^2}{4} + 1] \tag{44}$$

$$r \in [-\frac{x^2}{4} + x, \frac{x^2}{4} + 1] \wedge 0.5 \circledast (r \oplus (x \oslash r)) = r \rightarrow 0.5 \circledast (r \oplus (x \oslash r)) \in (1 + 4\varepsilon_{rel}) \sqrt{x} \tag{45}$$

Note that boundedness checks appearing as conclusions for the exception freedom VCs (41)–(43) also appear as hypotheses in each following VC in conjunction with the hypothesis $r \in [-\frac{x^2}{4} + x, \frac{x^2}{4} + 1]$ of (41). We therefore omit these checks from the hypotheses of (42)–(45) to improve readability.



(a) The effect of FP precision on proving time, for different polynomial degrees.



(b) The effect of polynomial degree on the number of computed sub-problems, for different FP precisions.

Fig. 12: Performance of PolyPaver depending on polynomial degree and problem tightness. Proving VC (45) derived from the `sqrt` program with various FP precisions. The higher is the precision, the tighter is the VC.

5.2.4 Proof

Solving was performed for values of x in $[0.5, 2]$ and values of r in $[0, 3]$. PolyPaver successfully discharged (39)–(44) in well under a second using degree 0 intervals. Using higher degrees decided (39), corresponding to computation from start to the loop invariant, without performing splitting and slightly outperforming solving with constant intervals. Affine intervals did show consistently lower solving times for (41)–(43), however the difference was within measurement error. With the exception of (45), the remaining cases showed higher degrees leading to monotonically increasing solving times, which was expected from reasoning analogous to that in Section 5.1.2.

The remaining VC (45), corresponding to computation from the loop invariant to termination, is the one requiring the most from the approximations used in PolyPaver. It states that on exit from the loop, *i. e.* when the loop action makes a negligible change to the value of r, the value is very near $\sqrt{x}$. This statement is formalised in the postcondition constraint and we expected its proof to show the most benefits of using higher degrees.

Solving with degree 0 intervals either timed out after several hours, or given 24 hour timeout ran out of memory. To obtain data for analysis we simulated lower precision FP types by increasing ε_{rel} to reflect a smaller mantissa bit size than the 23 bits of single precision IEEE-754 FP numbers. Proofs were attempted for precisions 6 to 23 and degrees 0 to 8 and the results for the highest and even and precisions are summarised in Figures 12a and 12b. The missing data points in the graphs denote that the proof was terminated prematurely as subdivision had generated a domain box with intervals whose endpoints are double precision FP neighbours.

Figure 12a shows the graphs of solving time, as a function of FP precision, for each degree. The final graph, labelled "optimal", chooses the minimum solving time for the given precision. Clearly, it is possible to choose a positive degree for each precision for which PolyPaver performs better than when using constant intervals. This validates our expectations that the (45) requires very precise approximation and that higher degrees give an overall benefit to the prover algorithm. The improvement in solving time is due to the ability of higher-degree enclosures to decide the VC over larger sub-domains, leading to less splitting and thus less computation. Since the additional cost of computing with higher-degree polynomials is lesser than the exponential cost of splitting, the overall performance was improved.

Figure 12b shows the number of computed sub-problems for each precision, as a function of degree. Since this number is proportional to the space complexity of the proof, the graph gives a picture of the asymptotic cost of precision and the ability of higher degree enclosures to mitigate it. We see that, for each precision, using higher degrees leads to significantly smaller numbers of sub-problems, with improvements ranging between one and three orders of magnitude. As precision is increased, higher degrees are required for successful completion of the proof, with the full 23 bit precision proof only succeeding when using degree 8 enclosures.

5.3 Verification of `erfRiemann`

Here we return to the program given in Figure 1 in Section 1. To be able to apply PolyPaver on this program, we need to supply a loop invariant and determine domains for its parameters. These are shown in the version of the program in Figure 13.

The padding by $(n + 1) \cdot \delta$ around the model error terms in the right-hand-side of the postcondition is used to model the accumulation of rounding errors with increasing n. To control the tightness of the constraint, we scale the padding with a small positive δ. The minimum δ for which the contract holds depends on the precision of the FP format used to execute the program.

```
-- This procedure computes an upper Riemann sum of the error function
-- scaled by sqrt(pi)/2, using a uniform partition of size n.
procedure erfRiemann(in $x \in \mathbb{F}$; in $n \in \mathbb{N}$; out $r \in \mathbb{F}$)
    pre  $x \in [0,4]$, $n \in [1,100]$
    post $\left(r - \int_0^x e^{-t^2} dt\right) \in \left[ -(n+1)\cdot\delta,\ (1 - e^{-x^2})x/n + (n+1)\cdot\delta \right]$
    -- a bound on the total model and rounding errors
is
    $s \in \mathbb{F}$  -- size of one segment in the partition
    $i \in \mathbb{N}$  -- number of the current segment, starting from 0
    $y \in \mathbb{F}$  -- the starting point of the current segment
begin
    $s$ := $x \oslash n$;  -- the integration range is 0..x
    $r$ := 0;  -- initialise the accumulator of the Riemann sum
    $i$ := 0;  -- start with the first segment
    while $i < n$ do  -- iterate over segments
       $y$ := $i \circledast s$;  -- calculate the current segment starting point
       -- a loop invariant to facilitate induction and to bound local variables:
       assert $\left(r - \int_0^y e^{-t^2} dt\right) \in \left[ -(i+1)\cdot\delta,\ (1 - e^{-(ix/n)^2})x/n + (i+1)\cdot\delta \right]$
              and $r \in [-10, 100]$ and $i \in [0, n-1]$;
       $r$ := $r \oplus (s \circledast$ exp(-($y \circledast y$)));  -- add an upper volume for this segment
       $i$ := $i + 1$;
    od
end
```

Fig. 13: Implementation of the error function using a Riemann sum.

The loop invariant includes a version of the postcondition where n is replaced by i and x by y in one instance and by ix/n in another. The remainder of the loop invariant specifies ranges for the local variables as PolyPaver cannot determine them automatically.

Verifying this program entailed proving 19 VCs. Only two of these VCs are not proved within one second, namely those associated with the path from the loop invariant around the loop back to itself and the path that leads from the loop invariant to the end of the program.

The relevant parts of these VCs are the following:

$$
\begin{aligned}
&\left(r - \int_0^{(x \oslash n) \circledast i} e^{-t^2} dt\right) \in \left[-(\delta(i+1)), (1 - e^{-(xi/n)^2})\tfrac{x}{n} + \delta(i+1)\right] \\
&\implies \\
&\left(r \oplus x \oslash n \circledast (\mathtt{exp}(-i \circledast (x \oslash n) \circledast i \circledast (x \oslash n))) - \int_0^{(x \oslash n) \circledast (i+1)} e^{-t^2} dt\right) \\
&\qquad\qquad\qquad \in \left[-(\delta(i+2)), (1 - e^{-(x(i+1)/n)^2}) x/n + \delta(i+2)\right] \\
&\wedge \\
&r \oplus x \oslash n \circledast (\mathtt{exp}(-i \circledast (x \oslash n) \circledast i \circledast (x \oslash n))) \in [-10, 100].
\end{aligned}
\tag{46}
$$

With $\delta = 0.1$, PolyPaver took less than 2 hours to prove the second conclusion in the above VC and it took less than 2 days to prove the first one, after splitting the domain into 2,670,879 boxes.

5.4 Selected ProVal verified programs

We borrow two examples from the ProVal project[31] to facilitate a comparison of our verification method with theirs. Figure 14 shows an adapted source code of the example programs. These two examples are among three of the most complex verified FP programs published on the ProVal portal. The third one is concerned with very precise details of rounding (the Veltkamp/Dekker algorithm for computing the exact error of FP multiplication) and thus is a very different kind of problem than those we target.

Both original programs can be verified automatically using a combination of the Why VC generator, the Gappa prover and the Coq interval tactic. Our version has fewer auxiliary annotations, while still featuring the main postconditions. Using our method we get 4 VCs for `proval_cosine` and 28 VCs for `proval_sqrt`. PolyPaver proves each of the 4 `proval_cosine` VCs in less than 1s. We have thus shown that our annotation method and PolyPaver can automatically verify a cleaner, more readable and easier to produce version of a ProVal program.

Among the 28 VCs for `proval_sqrt`, PolyPaver proves all but three of these VCs within 2s. After some simplification, the hardest four VCs are show below, all featuring the two variables $x \in [1/2, 2]$ and $r \in [1/2, 3/2]$.

$$|r - 1/\sqrt{x}| <= 2^{-6}/\sqrt{x} \implies \left|r \oslash 2 \circledast (3 \ominus x \circledast r \circledast r) - 1/\sqrt{x}\right| <= 2^{-11}/\sqrt{x} \tag{sqrt-1}$$

$$|r - 1/\sqrt{x}| <= 2^{-11}/\sqrt{x} \implies \left|r \oslash 2 \circledast (3 \ominus x \circledast r \circledast r) - 1/\sqrt{x}\right| <= 2^{-21}/\sqrt{x} \tag{sqrt-2}$$

$$|r - 1/\sqrt{x}| <= 2^{-21}/\sqrt{x} \implies \left|r \oslash 2 \circledast (3 \ominus x \circledast r \circledast r) - 1/\sqrt{x}\right| <= 2^{-41}/\sqrt{x} \tag{sqrt-3}$$

$$|r - 1/\sqrt{x}| <= 2^{-41}/\sqrt{x} \implies |x \circledast r - \sqrt{x}| <= 2^{-43}/\sqrt{x} \tag{sqrt-4}$$

PolyPaver proves `sqrt-1` in less than 2s but runs out of time when attempting to decide the other three statements. It is not feasible for PolyPaver to decide these statements because the supporting sets where the hypothesis and conclusion hold are both very thin. In order to prove that one is inside the other the domain has to be split finely enough so that the boundaries of the boxes separate the two thin sets. An improved arithmetic such as ours cannot prevent this kind of splitting. In the case of `sqrt-2` the minimal depth of the tree of split boxes is over 22, requiring approximately 2^{22} boxes to be investigated. The actual depth

```
procedure proval_cosine(in $x \in \mathbb{F}$; out $r \in \mathbb{F}$)
  pre  $|x| \le 2^{-5} + 2^{-20}$  -- $x$ is not far from 0
  post $|r - \cos(x)| \le 2^{-24}$  -- $r$ is very close to $\cos(x)$
is
begin
  $r$ := $1.0 \ominus x \circledast x \circledast 0.5$;  -- a Taylor expansion for $\cos(x)$
end
```

```
procedure proval_sqrt(in $x \in \mathbb{F}$; out $r \in \mathbb{F}$)
  pre  $0.5 \le x \le 2$  -- $x$ is not far from 1
  post $|r - \sqrt{x}| \le 2^{-43} \cdot \sqrt{x}$  -- $r$ is very close to $\sqrt{x}$
is
  $t \in \mathbb{F}$  -- intermediate approximation of $1/\sqrt{x}$
begin
  sqrt_init($x$, $t$);  -- set $t$ to a value where Newton iteration converges

  $t$ := $0.5 \circledast t \circledast (3 \ominus t \circledast t \circledast x)$;  -- Newton iteration
  assert $|r - 1/\sqrt{x}| \le 2^{-11} \cdot 1/\sqrt{x}$;  -- bound on total error after iteration 1
  $t$ := $0.5 \circledast t \circledast (3 \ominus t \circledast t \circledast x)$;  -- Newton iteration
  assert $|r - 1/\sqrt{x}| \le 2^{-21} \cdot 1/\sqrt{x}$;  -- bound on total error after iteration 2
  $t$ := $0.5 \circledast t \circledast (3 \ominus t \circledast t \circledast x)$;  -- Newton iteration
  assert $|r - 1/\sqrt{x}| \le 2^{-41} \cdot 1/\sqrt{x}$;  -- bound on total error after iteration 3

  $r$ := $x \circledast t$;  -- conversion $1/\sqrt{x} \rightarrow \sqrt{x}$
end

procedure sqrt_init(in $x \in \mathbb{F}$; out $r \in \mathbb{F}$)
  pre $0.5 \le x \le 2$  -- $x$ is not far from 1
  post $|r - 1/\sqrt{x}|$ <= $2^{-6} \cdot 1/\sqrt{x}$  -- $r$ is close to $1/\sqrt{x}$
...
```

Fig. 14: Two example annotated programs from [31] translated from C to our language and without some assertions that serve as hints to Why and Gappa but do not help PolyPaver. The three assertions were added to break down one difficult VC into four easier ones.

is usually over 30 due to the approximation errors of the arithmetic, making an exhaustive search over all boxes infeasible.

Among the VCs generated by the ProVal tools for the cosine example is one that is in essence as follows:

$$\begin{aligned}&\Big(\forall x_1 \in [-2^{-4}, 2^{-4}], x_2 \in [-2^{-5}, 2^{-5}]\Big)\Big(x_1 - x_2 \in [-2^{-20}, 2^{-20}]\\ &\quad \Longrightarrow \; \big|(1 \ominus (x_1 \circledast x_1) \oslash 2) - 1 + {}^{x_2^2}\!/_2\big| - |x_1 - x_2| \le {}^{131201}\!/_{2^{41}}\Big).\end{aligned} \qquad \texttt{(cosine-2)}$$

This VC is very tight in the same way as the above square root VCs (i. e. close to being as strong as possible). Nevertheless, it is significantly simpler than the square root VC. PolyPaver times out on this VC in the standard mode described in this paper but it can prove it in 10s with an experimental extension that, while splitting, rotates and skews the boxes, ensuring coverage of the original domain while allowing for thin boxes that better fit in the thin gaps between the supporting sets of the hypothesis and the conclusion. This VC is representative of properties near the boundary of what our numerical proving approach can manage to prove automatically in reasonable time. We believe that the best way to address this limitation is to combine our numerical proving approach with symbolic manipulations such as those employed in Gappa or MetiTarski.

5.5 Checking interpolated values

Next, we show how our method applies on a slightly larger example, which is shown in Figure 15 and whose full implementation in SPARK and VCs generated by SPARK tools are available on the PolyPaver portal[17]. It is an adaptation of code present in a commercial safety-critical project. Its task is to compute a bound on the maximum of a quadratic polynomial interpolating three consecutive data points in a time-series of non-negative FP values with uniform time step.

For simplicity, we set the time step to 1 and thus interpolate the points $(-1, y_1)$, $(0, y_2)$, and $(1, y_3)$, yielding the coefficients $a = \frac{y_1 - 2y_2 + y_3}{2}$, $b = \frac{y_3 - y_1}{2}$, and $c = y_2$ of the polynomial $q(x) = ax^2 + bx + c$. These coefficients are computed by the procedure `coeffs`. When there is a critical point, the value of the polynomial at the critical point is given by $c - \frac{b^2}{4a}$. This value is computed by the procedure `max_quad`. When a is negative, this value is the global maximum of the polynomial. The postcondition of `max_quad` expresses this fact, albeit restricted to the domain $[-1, 1]$. The postcondition needs a universally quantified variable x. Since we do not support explicit quantification, we had to make x a dummy program variable. Note that, instead of $a < 0$, we used the stronger precondition $a < -0.05$ to make the VCs less tight and feasibly provable by the current PolyPaver technique.

The procedure `max_unit` is returns a number that is close to the maximum of the interpolated polynomial over the unit interval $[-1, 1]$. One undesirable consequence of strengthening the precondition of `max_quad` was the need to correspondingly strengthen the test in the body of `max_unit` that guards the call to `max_quad`. This complicates the specification of `max_unit` because it ignores potential peaks near 0 with $a \in [-0.05, 0]$. The postcondition includes a buffer to accommodate for the peaks that the program ignores, effectively providing a bound on how high such peaks could be. The postcondition is also somewhat weakened so that the VCs are sufficiently loose to be proved by PolyPaver in a reasonable time. There were only 3 non-trivial VCs. The hardest VC has 8 variables and PolyPaver proved it in around 10min after splitting the domain into 507911 segments.

All *exception freedom* VCs were trivial for PolyPaver — no splitting was required to prove them with degree 0. If we are interested in exception freedom only, we can drop all postconditions from the program except those that give explicit numerical ranges for output values.

Having verified this specification, we attempted to verify a tighter specification. While we were not successful in this goal, PolyPaver was still useful in discovering subtle errors in our attempts at a much tighter specification such as swapped polynomial coefficients or an inverted branching condition. When it disproved a specification, it provided a counterexample in the form of values for the parameters for which the buggy program deviates from the specification. Thus we have experienced PolyPaver as an effective tool for early detection of numerical programming bugs that are outside the scope of other currently available static analyses.

```
-- Compute an upper bound over $x \in [-1,1]$ for the quadratic polynomial
-- that interpolates the points given by the values $y_1, y_2, y_3$ for $x = -1, 0, 1$, respectively.
procedure max_unit(in $y_1, y_2, y_3 \in \mathbb{F}$; out $r \in \mathbb{F}$)
  pre   $y_1, y_2, y_3 \in [0,1]$  -- a range restriction on the input points
  post  $r \geq y_1 - 0.2 \ \wedge\ r \geq y_2 - 0.2 \ \wedge\ r \geq y_3 - 0.2$
        -- the result (almost) dominates the given points
is
  $a, b, c, m_1, m_2 \in \mathbb{F}$  -- coefficients of the quadratic polynomial and intermetiate maxima
begin
  max($y_1$, $y_3$, $m_1$);  -- i.e. $m_1 = \max(y_1, y_3)$
  coeffs($y_1$, $y_2$, $y_3$, $a$, $b$, $c$);  -- quadratic polynomial interpolation
  if $a < -0.05 \ \wedge\ 2.0 \circledast a \leq b \ \wedge\ b \leq -2.0 \circledast a$ then  -- is a maximum inside $[-1,1]$?
    max_quad($a$, $b$, $c$, $0.0$, $m_2$);  -- compute the maximum of the polynomial inside $[-1,1]$
    max($m_1$, $m_2$, $r$);  -- ensure the result not below $y_1$ and $y_2$
  else
    $r$ := $m_1$;  -- the maximum is close to $-1$ or $1$
  fi;
end
```

```
-- Compute the coefficients of a quadratic polynomial that interpolates
-- the three points given by $y_1, y_2, y_3$ for $x = -1, 0, 1$, respectively.
procedure coeffs(in $y_1, y_2, y_3 \in \mathbb{F}$; out $a, b, c \in \mathbb{F}$)
  pre   $y_1, y_2, y_3 \in [0,1]$  -- a range restriction on the input points
  post  $a - b + c - y_1 \in [-10^{-6}, 10^{-6}] \ \wedge$  -- the polynomial is near point 1
        $c = y_2 \ \wedge$  -- the polynomial is near point 2
        $a + b + c - y_3 \in [-10^{-6}, 10^{-6}] \ \wedge$  -- the polynomial is near point 3
        $a \in [-1.1, 1.1] \ \wedge\ b \in [-1.1, 1.1] \ \wedge\ c \in [-1.1, 1.1]$  -- the coefficients fit in $[-1.1, 1.1]$
is
begin
   -- the following formulas are obtained by solving a simple linear system:
  $a$ := $0.5 \circledast (y_1 \ominus 2.0 \circledast y_2 \oplus y_3)$;
  $b$ := $0.5 \circledast (y_3 \ominus y_1)$;
  $c$ := $y_2$;
end
```

```
-- Compute an upper bound on a quadratic polynomial with the given coefficients
-- over the interval $[-1,1]$, assuming the polynomial is convex upward.
-- The parameter $x$ is used only to formulate the postcondition. It's value is irrelevant.
procedure max_quad(in $a, b, c, x \in \mathbb{F}$; out $r \in \mathbb{F}$)
  pre   $a \in [-1.2, -0.05] \ \wedge\ b \in [-1.2, 1.2] \ \wedge\ c \in [-1.2, 1.2] \ \wedge$
        $x \in [-1, 1]$
  post  $r \geq ax^2 + bx + c - 0.05 \ \wedge$  -- $r$ (almost) dominates the polynomial for any $x \in [-1,1]$
        $r \leq 10$  -- a bound on $r$ following from the bounds on the coefficients
is
begin
  $r$ := $c \ominus 0.25 \circledast b \circledast b \oslash a$;  -- $= c - b^2/4a$, i.e. the value at point $x = -b/2a$
end
```

```
procedure max(in $x, y \in \mathbb{F}$; out $r \in \mathbb{F}$)
  post  $(r = x \ \wedge\ x \geq y) \ \vee\ (r = y \ \wedge\ y \geq x)$  -- $r$ is the larger one among $x$ and $y$
...
```

Fig. 15: A procedure computing the maximum of a quadratic interpolation of three points

6 Comparison with other numerical provers

In the previous section there are examples for which PolyPaver proves non-linear error bounds for numerical programs, in some cases fairly tight ones. PolyPaver and our method of deriving verification conditions match in several aspects. Nevertheless, in many cases it is possible to apply other automated theorem provers on the verification conditions generated by our method, sometimes with some level of symbolic manipulation to circumvent the limitations of various provers. The goal of this section is to compare PolyPaver with RealPaver 0.4, RSolver 3.01, Gappa 0.14.1 and MetiTarski 1.9 in terms of ability to prove numerical theorems, in particular theorems arising as VCs for numerical programs using our method or other similar methods. Subsection 6.1 reports on how the provers compare when solving one of the hardest VCs encountered in Section 5. To enable comparison on a numerical scale, Subsection 6.2 reports on solving some parametrised benchmark problems that do not arise as VCs.

RealPaver [23] is similar to PolyPaver in its basic concept but uses plain interval arithmetic together with an array of interval computation and constraint propagation techniques. RealPaver's expression language features a limited set of real functions and has no explicit interval operators. The same is true for RSolver, Gappa and MetiTarski, but the set of real functions is different for each prover. Moreover, RealPaver formulae are limited to conjunctions of inequalities over real expressions.

RealPaver computes *outer approximations* of the set of solutions of a given constraint system, *i. e.* a union of boxes containing the solution set. For our purposes we need to know that a theorem is true *everywhere* in its domain and therefore apply RealPaver to the *negation* of the theorem. If an empty outer approximation is computed, it has been proved that the negation completely lacks solutions, or equivalently, that the theorem is true over the entire domain.

RSolver[34] is also similar to PolyPaver in its basic concept but its focus is on proving first-order formulae with existential and universal quantifiers.

Gappa[10] uses a combination of symbolic and numerical proving, treating interval evaluation and domain splitting as some of the available proof rules. It natively supports the FP rounding operator but is poor on common numerical functions, covering only field operations and the square root.

MetiTarski[3] also uses a combination of symbolic and numerical proving, but its numerical axiom base includes a fixed selection of tight polynomial and rational approximations of elementary functions. It is therefore quite different from the other provers in that it does not perform any interval evaluation. On the other hand, it is similar to PolyPaver in its use of tight polynomial bounds for numerical expressions that appear in theorems.

6.1 Square root VC

The first property we benchmark the provers on is the VC in equation (45) on page 32. Due to the lack of support for interval literals in the provers, we had the choice of either

1. eliminating interval literals symbolically by translating an interval expression to a pair of bounds, leading us to translate problem (45) to:

$$\begin{aligned}&\Big(-\tfrac{x^2}{4}+x\le r \quad\wedge\quad r\le\tfrac{x^2}{4}+1 \quad\wedge\quad L\le r \quad\wedge\quad r\le R\Big) \implies\\ &\qquad\Big((1-4\varepsilon_{\mathrm{rel}})\sqrt{x}\le L \quad\wedge\quad R\le(1+4\varepsilon_{\mathrm{rel}})\sqrt{x}\Big)\\ &\text{where}\\ &L=(1-\varepsilon_{\mathrm{rel}})\tfrac{1}{2}\Big((1-\varepsilon_{\mathrm{rel}})(r+((1-\varepsilon_{\mathrm{rel}})\tfrac{x}{r}-\varepsilon_{\mathrm{abs}}))-\varepsilon_{\mathrm{abs}}\Big)-\varepsilon_{\mathrm{abs}}\\ &R=(1+\varepsilon_{\mathrm{rel}})\tfrac{1}{2}\Big((1+\varepsilon_{\mathrm{rel}})(r+((1+\varepsilon_{\mathrm{rel}})\tfrac{x}{r}+\varepsilon_{\mathrm{abs}}))+\varepsilon_{\mathrm{abs}}\Big)+\varepsilon_{\mathrm{abs}}\\ &\varepsilon_{\mathrm{rel}}=2^{-23}\qquad \varepsilon_{\mathrm{abs}}=2^{-126}\end{aligned}\tag{47}$$

2. or replacing each occurrence of an interval literal with a fresh variable whose bounds correspond to the interval endpoints, leading us to translate problem (45) to:

$$\begin{aligned}&\Big(-\tfrac{x^2}{4}+x\le r \quad\wedge\quad r\le\tfrac{x^2}{4}+1 \quad\wedge\quad r=E\Big) \implies\\ &\qquad\Big((1-4\varepsilon_{\mathrm{rel}})\sqrt{x}\le E \quad\wedge\quad E\le(1+4\varepsilon_{\mathrm{rel}})\sqrt{x}\Big)\\ &\text{where}\\ &E=(1+\varepsilon_{\mathrm{r3}})\tfrac{1}{2}\Big((1+\varepsilon_{\mathrm{r2}})(r+((1+\varepsilon_{\mathrm{r1}})\tfrac{x}{r}+\varepsilon_{\mathrm{a1}}))+\varepsilon_{\mathrm{a2}}\Big)+\varepsilon_{\mathrm{a3}}\\ &\varepsilon_{\mathrm{r1}},\varepsilon_{\mathrm{r2}},\varepsilon_{\mathrm{r3}}\in[-\varepsilon_{\mathrm{rel}},\varepsilon_{\mathrm{rel}}]\qquad \varepsilon_{\mathrm{a1}},\varepsilon_{\mathrm{a2}},\varepsilon_{\mathrm{a3}}\in[-\varepsilon_{\mathrm{abs}},\varepsilon_{\mathrm{abs}}]\end{aligned}\tag{48}$$

The issue with choice 1 is that this translation is generally nontrivial and difficult to automate. The translation involves monotonicity analysis and when the operators in the expressions are not monotone, the number of cases to consider grows exponentially with expression size. In our example we were lucky because all functions involved are monotone over the given domain. (An expression for which the number of cases grows can be found in [18].) The issue with choice 2 is that the dimension of the problem domain increases considerably, normally leading to an exponential slowdown. If we could tell the provers not to split the domain of the newly introduced variables, the splittable domain dimension would stay unchanged and we could make a better comparison. Unfortunately, only RSolver gives such an option. However, as RSolver does not support the square root we cannot apply it to this problem.

As stated earlier, the RealPaver theorem language also lacks disjunction and implication and thus we have to modify the problems further for RealPaver. The solution pointed to in the RealPaver manual is to use the minimum function as follows:

$$a\le b\vee c\le d \quad\Leftrightarrow\quad \min(a-b,c-d)\le 0.$$

Also having to negate the original VC (45), we end up giving RealPaver the following problem when using choice 1

$$\begin{aligned}&r\ge-\tfrac{x^2}{4}+x \quad\wedge\quad r\le\tfrac{x^2}{4}+1 \quad\wedge\quad r\ge L \quad\wedge\quad r\le R \quad\wedge\\ &\qquad\min\Big(R-(1-4\varepsilon_{\mathrm{rel}})\sqrt{x},(1+4\varepsilon_{\mathrm{rel}})\sqrt{x}-L\Big)\le 0\end{aligned}\tag{49}$$

for $x\in[0.5,2]$ and $r\in[0,3]$, and using choice 2:

$$\begin{aligned}&r\ge-\tfrac{x^2}{4}+x \quad\wedge\quad r\le\tfrac{x^2}{4}+1 \quad\wedge\quad r=E \quad\wedge\\ &\qquad\min\Big(E-(1-4\varepsilon_{\mathrm{rel}})\sqrt{x},(1+4\varepsilon_{\mathrm{rel}})\sqrt{x}-E\Big)\le 0\end{aligned}\tag{50}$$

for $x\in[0.5,2]$, $r\in[0,3]$, $\varepsilon_{\mathrm{r1}},\varepsilon_{\mathrm{r2}},\varepsilon_{\mathrm{r3}}\in[-\varepsilon_{\mathrm{rel}},\varepsilon_{\mathrm{rel}}]$, and $\varepsilon_{\mathrm{a1}},\varepsilon_{\mathrm{a2}},\varepsilon_{\mathrm{a3}}\in[-\varepsilon_{\mathrm{abs}},\varepsilon_{\mathrm{abs}}]$.

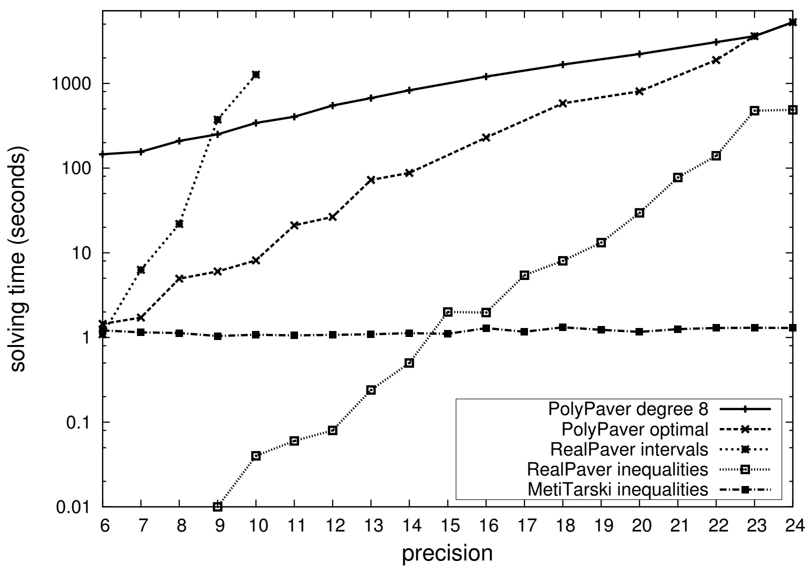
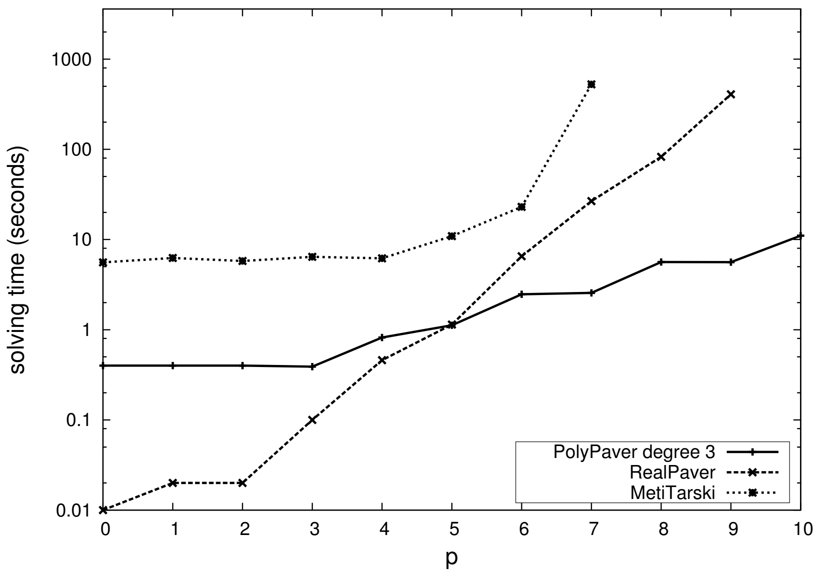


Fig. 16: The effect of FP precision (mantissa bitsize) on proving times for a translation of problem (45) by various provers.

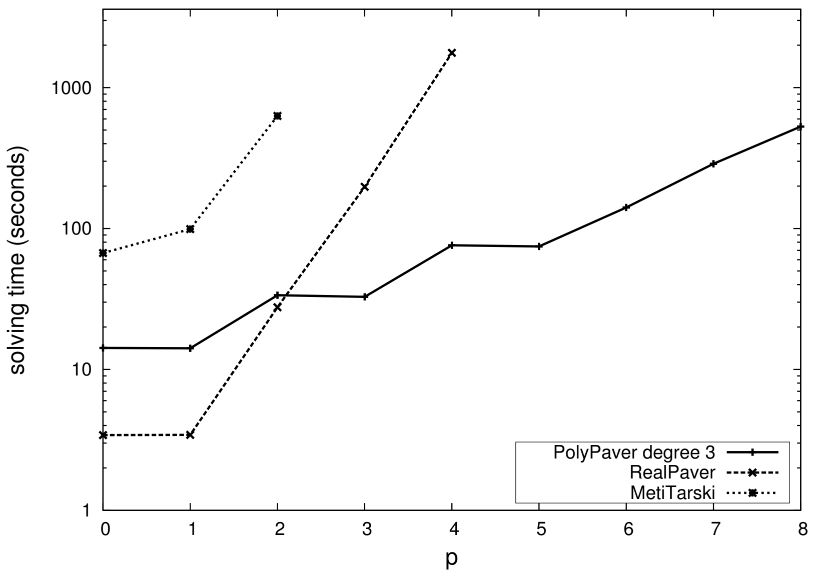
Figure 16 shows how well the three provers that we could apply on the VC did. On the 2-variable translation (47) MetiTarski clearly wins for higher precisions, probably because it has a good enough bound for square root and all other necessary rules to solve the problem using more or less the same proof irrespective of precision up to a certain, fairly high precision. RealPaver also outperforms PolyPaver in this test. Nevertheless, the inequality-based approach to expressing inclusions used to obtain (47) does not scale well when the functions are not monotone on the domain of interest. Using the 8-variable translation (48), which is easier to obtain, RealPaver was clearly lagging behind PolyPaver and MetiTarski gave up even for precision 6.

6.2 Parametrised numerical benchmarks

For the purpose of further comparing the performance of the provers, we use two families of benchmarks parametrised by *dimension* and *tightness*. As such benchmarks, we chose familiar identities between real expressions that are invariant under permutation of variables, yielding benchmarks that scale *uniformly* with dimension. Then, the chosen identity is turned into an inequality and one of the expressions is shifted by a small distance, separating the identical surfaces given by the syntactically distinct but functionally equivalent expressions. The distance controls the tightness of the inequality.

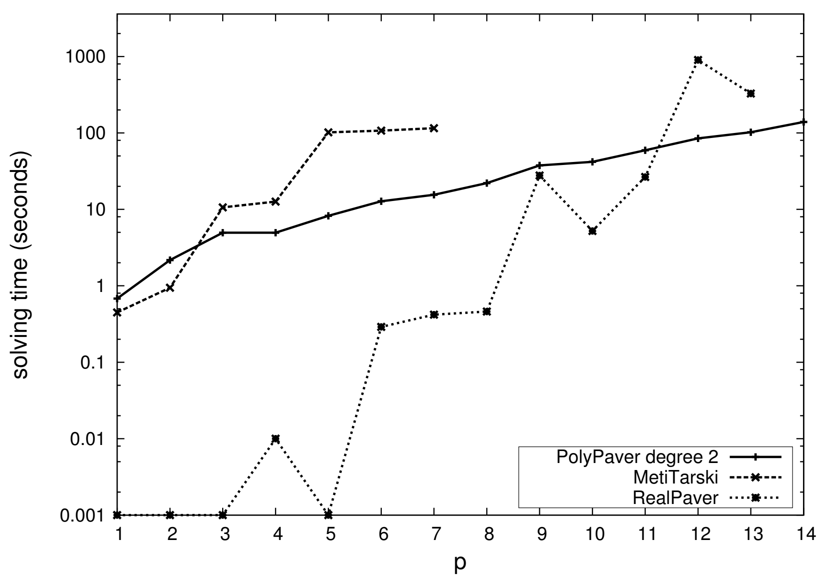


(a) The effect of 2D exponential benchmark tightness on solving time for different provers.

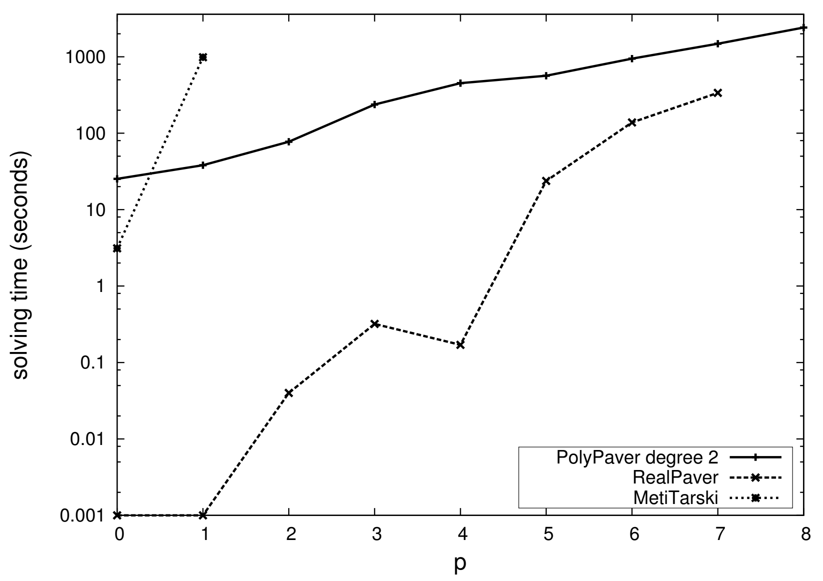


(b) The effect of 3D exponential benchmark tightness on solving time for different provers.

Fig. 17: Solving times of the provers on the benchmark problems $\varphi^{exp}_{2,p}$ in (a) and $\varphi^{exp}_{3,p}$ in (b). Any missing data points correspond to solving time exceeding 1 hour. Graphs show how long it took to prove a problem as a function of its p parameter.



(a) The effect of 2D square root benchmark tightness on solving time for different provers.



(b) The effect of 3D square root benchmark tightness on solving time for different provers.

Fig. 18: Solving times of the provers on the benchmark problems $\varphi^{sqrt}_{2,p}$ in (a) and $\varphi^{sqrt}_{3,p}$ in (b). Any missing data points correspond to solving time exceeding 1 hour. Graphs show how long it took to prove a problem as a function of its p parameter.

Formally, these benchmarks are given by real predicates $\varphi^{exp}_{d,p}$, $\varphi^{sqrt}_{d,p}$, parametrised by natural numbers $d, p \in \mathbb{N}$, with $d \geq 1$, defined below.

$$\varphi^{exp}_{d,p} \equiv \forall x_1, \ldots, x_d \in [0,1] : \prod_{i=1}^{d} e^{x_i} < 2^{-p} + e^{\sum_{i=1}^{d} x_i} \tag{51}$$

$$\varphi^{sqrt}_{d,p} \equiv \forall x_1, \ldots, x_d \in [1,2] : \sqrt{\prod_{i=1}^{d} x_i} < 2^{-p} + \prod_{i=1}^{d} \sqrt{x_i} \tag{52}$$

RSolver and Gappa were unable to prove these properties even for $d = 2, p = 0$. The timings for RealPaver, MetiTarski, and PolyPaver are shown in Figures 17 and 18 for benchmark (51) and (52), respectively.

MetiTarski does not score well in these tests because it relies on a finite set of axioms, mainly polynomial bounds, that suffice to prove properties only up to a certain precision. If the two identities that are embedded in these properties would be among MetiTarski's axioms, MetiTarski would certainly outperform the other provers. This illustrates the fact that it is hard to engineer a set of axioms that do not overwhelm the reasoning engine with too much choice and also cover most problems in the target application domain.

As both PolyPaver and RealPaver are based on domain subdivision, they necessarily suffer from an exponential cost in the number of variables in a given theorem. (MetiTarski is sensitive to the number of variables even more due to using QEPCAD-B to compare polynomials.) Thus, the *dimension* parameter d allows for a uniform scaling of the toughness of a proof as a function of dimension. Both these provers decide inequalities by finding boxes separating the surfaces given by the expressions in the relation. Thus, the *precision* parameter p allows for a uniform scaling of the toughness of a proof as a function of *tightness*. In these benchmarks PolyPaver outperforms RealPaver on tighter problems and RealPaver outperforms PolyPaver on looser problems. It is also worth noting the complicated and less predicable behaviour of RealPaver, which we attribute to the number and complexity of the different techniques inside it. PolyPaver, on the other hand, is relatively simple and predictable.

7 Scalability

To study the current limits of our method, we conducted tests on randomly generated programs and specifications. Each of the programs is a single FP expression built using variables and binary operators. The derived specification is a tight functional specification that links the values of the FP expression with the intended exact interpretation of the expression.

In each test, the generated expressions increase in size or complexity in some way. We focus our study on the impact of each of the following parameters:

- Expression dimension, i. e. the number of free variables
- Expression depth, i. e. the length of the longest chain of nested operations
- Expression size, i. e. the overall number of binary operations

Another parameter that strongly impacts the performance of PolyPaver is the specification tightness. We study the impact of changing tightness in Subsection 6.2. Here we only conduct a small number of tightness tests to determine a reasonable tightness to use in the remaining tests.

7.1 Benchmark generation

While random generation of expressions is simple, it is highly nontrivial to randomly generate tight specifications for the generated programs, which have a good chance of being both correct and feasibly provable. To keep the task manageable, we generate single-expression programs. An analogous investigation of a richer class of programs is beyond the scope of this work.

In summary, a specification for an expression is generated by sampling various tuples of input values and executing the program in different precision arithmetics, from which an estimate of the actual absolute error of the program is obtained. We could try to prove that the estimate is a correct error bound, but it is very unlikely that this would be true. Instead, we scale the estimate by a tightness parameter and then try to prove that this is a correct error bound.

The parameters for generating an expression are its arity n, its depth d and whether the expression should be a *balanced* binary tree such as $((y * x) + (y + z)) * ((x + x) + (y + z))$ (depth 3), or a *deep non-branching* binary tree such as $x + (y * (z + (x * z)))$ (depth 4). Further parameters $\tau > 1$ and $N \in \mathbb{N}$ will be introduced later.

An expression e is obtained by random sampling of variables and binary operations while constructing a tree of the selected shape and size. The variables are drawn from the set $\{x_1, \ldots, x_n\}$. We take care to exclude expressions where some variable happens not to occur, to guarantee that the expression will have the specified arity. Binary operations are drawn from the set comprising the operations $x + y, x - y, x * y, \sqrt{1 + x^2 + y^2}$. Note that these real operations are total and relatively stable.

The *idealised meaning* $[\![e]\!]^{\mathbb{R}}$ of an expression e is defined by interpreting the values of the variables as real numbers and operations as real functions. The *actual meaning* $[\![e]\!]^{\mathbb{F}}$ of e is obtained by interpreting the values of variables as values in the FP format $\mathbb{F}$, and the operations as rounded to fit into $\mathbb{F}$.

The specification for a program obtained from expression e with free variables $x_1, \ldots, x_n$ is

$$x_1 \in [-1, 1], \ldots, x_n \in [-1, 1] \Rightarrow [\![e]\!]^{\mathbb{F}} \in [\![e]\!]^{\mathbb{R}} + \tau \cdot [-\delta, \delta]$$

where $\tau > 1$ is the tightness parameter, $\delta = \max\{|\delta_1|, \ldots, |\delta_N|\}$ and each value δ_i is obtained using the following steps:

1. Select a random tuple v of FP values in $[-1, 1]^n$.
2. Estimate $[\![e]\!]^{\mathbb{R}}(v)$ by evaluating e in v using a very high precision interval arithmetic.
3. Evaluate $[\![e]\!]^{\mathbb{F}}(v)$ using ordinary single-precision FP arithmetic.
4. Set $\delta_i = [\![e]\!]^{\mathbb{R}} - [\![e]\!]^{\mathbb{F}}$.

7.2 Tests

We set $N = 1000$ and calibrated the tightness τ by running a sample of verification problems with 16 operations for $\tau = 10, 100, 1000, 10000$. We chose the value $\tau = 1000$ because it was the least value that led to good results.

We ran three sets of tests according to the three parameters outlined earlier, namely expression arity, depth, and number of operations. The other parameters were set as follows:

- Arity tests: balanced expressions with 16 operations.
- Size tests: balanced expressions of arity 2.
- Depth tests: deep non-branching expressions of arity 2.

Each test was run for 20 randomly generated programs. The timeout was set to 10 hours. PolyPaver was executed single-threaded on an Intel i7-2600 3.40GHz machine with 16GB RAM and 8MB cache per core, running Kubuntu 12.04.

7.3 Analysis

Figure 19 shows the outcome of our three tests, organised in three groups of two plots. The plots on the left show the proportions of different types of overall result. The result "stopped" means that the prover has reached a bisection depth beyond a threshold of 1000 or the depth-first queue reached a threshold of 50, implying that the proof, if successful could take over a year to complete. The plots on the right show averages and standard deviations of proof durations for those programs that were successfully proved.

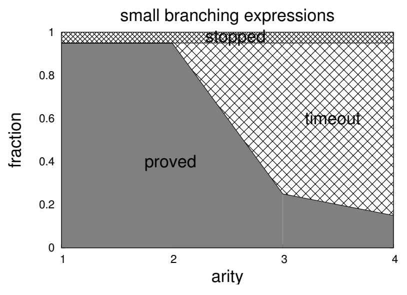
We conjecture that in a majority of the cases where the prover stopped, the chosen tightness was too high for PolyPaver. When the same happens in practice, the programmer loosens the specification until PolyPaver succeeds in reasonable time.

As may be observed from Figure 19, we found that arity matters most and that PolyPaver can effectively verify most of the generated programs with arity two. This is the reason why we chose to fix arity at two for the other tests.

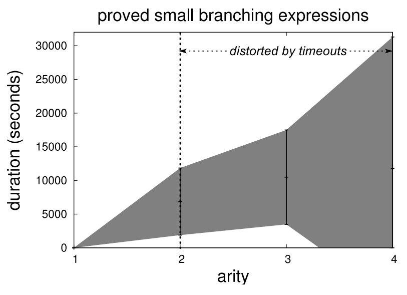
We set the timeout to ten hours and determined sizes for which reasonably few timeouts occurred. The duration data for tests with many timeouts is distorted by the fact that a significant proportion of the problems that could be proved are missing because their proof takes too long. This fact is clearly indicated in the plots where this applies.

The results seem to indicate that proving duration is close to linear in the size of the program. A larger sample size and much longer timeout would be needed to substantiate a confident verdict.

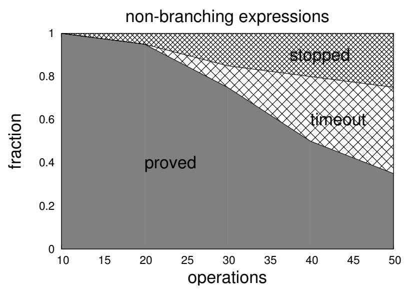
We conclude that arity is the biggest bottleneck. As expected, the proof duration is approximately exponential in the program arity. This is in line with results reported by all other tools based on subdivision, which is currently the dominating approach in automated non-linear real-number theorem proving.



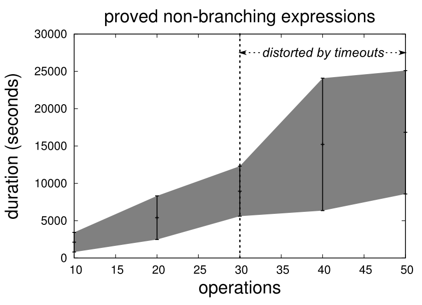
(a) Impact of arity on result



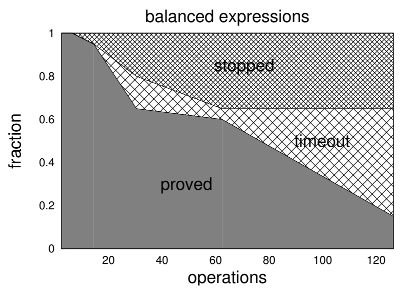
(b) Impact of arity on proof duration



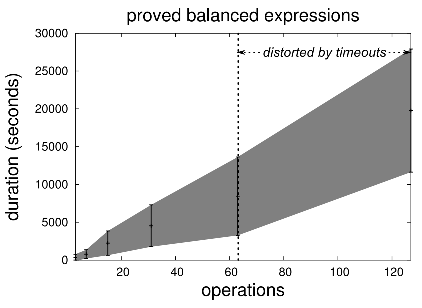
(c) Impact of size on result



(d) Impact of size on proof duration



(e) Impact of size on result



(f) Impact of size on proof duration

Fig. 19: Impact of expression complexity on proof success rate and effort. The annotation “distorted by timeouts” indicates that the true average duration in the annotated region is likely to be substantially higher than the experiments show. In (a), (c) and (e) the impact on the proportion of different results and in (b), (d) and (f) the impact on duration of successful proofs are shown for balanced expressions with 16 operations and given arity, for deep expressions with given number of operations and for balanced expressions with given number of operations, respectively. The duration plots are centred around the average duration and their widths indicate the standard deviation of the duration.

8 Conclusion

We have given a formal account of the main concepts behind our numerical theorem prover PolyPaver, which is based on a novel polynomial interval arithmetic that automatically computes tight polynomial bounds for expressions.

While PolyPaver can be applied in various contexts, it was developed with verification of floating-point programs in mind. We showed that PolyPaver can facilitate an automatic verification of floating-point programs against tight functional specifications with the integral operator and interval inclusions. The ability to naturally deal with the integral operator and interval expressions and inclusions is unique to PolyPaver.

While being relatively simple and implemented in a high-level functional language with no focus on optimisation, PolyPaver performs comparably to other published state-of-the-art numerical provers.

Our experiments demonstrate that higher-degree polynomial intervals can significantly outperform lower-degree polynomials, such as affine and constant interval forms, when used to approximate numerical expressions in subdivision-based numerical constraint solving.

Finally, we have probed the extent to which PolyPaver scales to larger problems using randomly generated program-specification pairs of various sizes. It appears that PolyPaver scales well with the size of program but not well with the number of variables in the program. Other published non-linear numerical theorem provers and solvers also do not scale well with the number of variables.

PolyPaver, as described here and with extensions not mentioned here, is BSD-licensed and its sources available on its development portal at [17].

8.1 Further work

Improved automation. The parameters of PolyPaver, such as polynomial degree and splitting bounds, need to be set manually or scripted in a problem-dependent way. In order to fully automate the solving process, we need to develop heuristics for setting these parameters. This includes automatically finding close-to-optimal degrees and other effort indicators in a problem- or even box-dependant manner.

Improved power. Our solver does not support quantification, which is essential for expressing and proving properties of programs with arrays. The work of Ratschan [33, 34] on first order numerical constraint solving and Fränzle et al. [21] on integrating SMT and NCSP solvers provide a direction, in which we plan to extend our approach so that it can handle programs with arrays.

Combination with symbolic reasoning. We believe that the power of PolyPaver will be best utilised in combination with automated or interactive symbolic proving techniques. The symbolic provers can supply what PolyPaver lacks, such as ability to prove equalities and touching inequalities, and automatic problem decomposition. PolyPaver can aid symbolic provers by identifying counter-examples to false statements, quickly rating sub-problems in terms of their likely validity and tightness and prove small but hard non-linear inequalities.

One starting point of our future work will be to symbolically pre-process theorems to remove as much unnecessary touching as can feasibly be done.

We foresee that an integration with an SMT solver will help us not only to deal with quantifiers as mentioned earlier but also to eliminate many simple cases of touching.

Another possible avenue for combining the main aspects of our method with symbolic reasoning is to act as an oracle for MetiTarski to provide further polynomial bounds on demand.

Documentation. We plan to rigorously document and formally analyse the PI arithmetic implementation used by the solver. This will enable us to produce a soundness proof for the solver implementation.

References

1. Ada Reference Manual, ISO/IEC 8652:2007(E) Ed. 3. Ada Europe (2007). URL `http://www.adaic.org/standards/05rm/html/RM-TTL.html`
2. Abramowitz, M., Stegun, I.A.: Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables, ninth dover printing, tenth gpo printing edn. Dover, New York (1964)
3. Akbarpour, B., Paulson, L.C.: Metitarski: An automatic theorem prover for real-valued special functions. J. Autom. Reasoning **44**(3), 175–205 (2010)
4. Amey, P.: Correctness by construction: Better can also be cheaper. CrossTalk Magazine pp. 24–28 (2002)
5. Barnes, J.: The spark way to correctness is via abstraction. Ada Lett. **XX**(4), 69–79 (2000). DOI http://doi.acm.org/10.1145/369264.369271
6. Barnes, J.: High Integrity Software: The SPARK Approach to Safety and Security, 2 edn. Addison-Wesley (2003)
7. Barnes, J., Chapman, R., Johnson, R., Widmaier, J., Cooper, D., Everett, B.: Engineering the tokeneer enclave protection software. In: Proceedings of IEEE International Symposium on Secure Software Engineering (2006)
8. Boldo, S.: Floats & Ropes: a case study for formal numerical program verification. In: 36th International Colloquium on Automata, Languages and Programming, *Lecture Notes in Computer Science - ARCoSS*, vol. 5556, pp. 91–102. Springer, Rhodos, Greece (2009)
9. Boldo, S.: How to compute the area of a triangle: a formal revisit. In: Proceedings of the 21th IEEE Symposium on Computer Arithmetic. Austin, Texas, USA (2013). URL `http://hal.inria.fr/hal-00790071`
10. Boldo, S., Filliâtre, J.C., Melquiond, G.: Combining coq and gappa for certifying floating-point programs. In: Calculemus '09/MKM '09: Proceedings of the 16th Symposium, 8th International Conference. Held as Part of CICM '09 on Intelligent Computer Mathematics, pp. 59–74. Springer-Verlag, Berlin, Heidelberg (2009). DOI http://dx.doi.org/10.1007/978-3-642-02614-0_10
11. Boldo, S., Lelay, C., Melquiond, G.: Improving Real Analysis in Coq: a User-Friendly Approach to Integrals and Derivatives. In: C. Hawblitzel, D. Miller (eds.) Proceedings of the The Second International Conference on Certified Programs and Proofs, *Lecture Notes in Computer Science*, vol. 7679, pp. 289–304. Kyoto, Japan (2012). DOI 10.1007/978-3-642-35308-6_22. URL `http://hal.inria.fr/hal-00712938`
12. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTREÉ analyzer. In: S. Sagiv (ed.) ESOP, *Lecture Notes in Computer Science*, vol. 3444, pp. 21–30. Springer (2005)
13. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-c: a software analysis perspective. In: Proceedings of the 10th international conference on Software Engineering and Formal Methods, SEFM'12, pp. 233–247. Springer-Verlag, Berlin, Heidelberg (2012). DOI 10.1007/978-3-642-33826-7_16. URL `http://dx.doi.org/10.1007/978-3-642-33826-7_16`
14. Daumas, M., Melquiond, G.: Certification of bounds on expressions involving rounded operators. ACM Trans. Math. Softw. **37**(1), 1–20 (2010). DOI http://doi.acm.org/10.1145/1644001.1644003
15. Delmas, D., Goubault, E., Putot, S., Souyris, J., Tekkal, K., Védrine, F.: Towards an industrial use of fluctuat on safety-critical avionics software. In: Proceedings of the 14th International Workshop on Formal Methods for Industrial Critical Systems, FMICS '09, pp. 53–69. Springer-Verlag, Berlin, Heidelberg (2009)

16. Duracz, J.: Verification of floatin point programs. Ph.D. thesis, Aston University (2010)
17. Duracz, J., Konečný, M.: PolyPaver development portal. `http://code.google.com/p/polypaver/`. Last accessed on 28th April 2013
18. Duracz, J.A., Farjudian, A., Konečný, M.: Enclosure constraints for floating point software verification. In: Proceedings of CFV 2009 in Grenoble (2009)
19. Duracz, J.A., Konečný, M.: Polynomial function enclosures and floating point software verication. In: Proceedings of CFV 2008 in Sydney, pp. 56–67 (2008)
20. Filliâtre, J.C., Paskevich, A.: Why3 – Where Programs Meet Provers. In: ESOP'13 22nd European Symposium on Programming, *LNCS*, vol. 7792. Springer, Rome, Italie (2013). URL `http://hal.inria.fr/hal-00789533`
21. Fränzle, M., Herde, C., Teige, T., Ratschan, S., Schubert, T.: Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. Journal on Satisfiability, Boolean Modeling and Computation **1**, 209–236 (2007)
22. Goubault, E., Putot, S.: Static analysis of numerical algorithms. In: K. Yi (ed.) SAS, *Lecture Notes in Computer Science*, vol. 4134, pp. 18–34. Springer (2006)
23. Granvilliers, L., Benhamou, F.: Algorithm 852: RealPaver: An interval solver using constraint satisfaction techniques. ACM Transactions on Mathematical Software **32**(1) (2006). URL `http://www.lina.sciences.univ-nantes.fr/Publications/2006/GB06`
24. Kaucher, E.: Interval analysis in the extended interval space ir. Computing, Suppl. **2**, 33–49 (1980)
25. Makino, K., Berz, M.: Efficient control of the dependency problem based on taylor model methods. Reliable Computing **5**, 3–12(10) (February 1999). URL `http://www.ingentaconnect.com/content/klu/reom/1999/00000005/00000001/00204749`
26. Mason, J.C., Handscomb, D.C.: Chebyshev Polynomials. CRC Press (2002)
27. Neher, M., Jackson, K.R., Nedialkov, N.S.: On taylor model based integration of odes. SIAM J. Numer. Anal. **45**(1), 236–262 (2007). DOI http://dx.doi.org/10.1137/050638448
28. Neumaier, A.: Taylor forms–use and limits. Reliable Computing **9**(1), 43–79 (2003)
29. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag New York, Inc., Secaucus, NJ, USA (1999)
30. Owre, S., Rushby, J.M., , Shankar, N.: PVS: A prototype verification system. In: D. Kapur (ed.) 11th International Conference on Automated Deduction (CADE), *Lecture Notes in Artificial Intelligence*, vol. 607, pp. 748–752. Springer-Verlag, Saratoga, NY (1992). URL `http://www.csl.sri.com/papers/cade92-pvs/`
31. ProVal team: Proval web portal. `http://proval.lri.fr/index.en.html`. Last accessed on 1st September 2011
32. Putot, S., Goubault, E., Martel, M.: Static analysis-based validation of floating-point computations. LNCS **2991**, 306–313 (2004). URL `http://www.springeronline.com/3-540-21260-4`
33. Ratschan, S.: Efficient solving of quantified inequality constraints over the real numbers. ACM Transactions on Computational Logic **7**(4), 723–748 (2006)
34. Ratschan, S., et al.: RSolver. `http://rsolver.sourceforge.net` (2004). Software Package