Polynomial Function Enclosures and Floating Point Software Verification*

Jan Andrzej Duracz and Michal Konečný

Computer Science, Aston University
Aston Triangle, B4 7ET, Birmingham, UK
{duraczja,m.konecny}@aston.ac.uk

Abstract. Proving partial correctness of floating point programs is a hard verification problem. This is in part because error analysis of finite precision computation is difficult and in part due to the high complexity of the generated verification conditions. Typical verification conditions that arise in this context are predicates with real inequalities as atoms and therefore numerical constraint solvers seem a natural choice for the automation of such proofs.
Freely available numerical constraint solvers are typically based on interval arithmetic due to existence of mature interval algorithms and relatively low computational cost. However, the pathological loss of precision that such arithmetic incurs may limit the applicability of interval based solvers for the type of first order numerical constraints that result from floating point verification problems.
We propose to use function enclosures as the numerical type to base a dedicated solver on. We present a prototype implementation using polynomial bounds with arbitrary precision floating point coefficients and evaluate it on some example verification conditions.

Introduction

We are concerned with partial correctness proofs of annotated floating point code. This is a hard verification problem, since the resulting *verification conditions* (VCs) typically contain complex arithmetic and boolean structures.

We work within the SPARK framework [1] which extends a subset of Ada with an annotation language and toolset comprising a VC generator, simplifier and an interactive prover. Recently, SPARK has moved towards full automation of VC proofs, in line with the *Correctness by Construction* philosophy, which promotes a software development process where programs are built incrementally, interleaving the development and verification of code. One of our aims is to provide SPARK with facilities to express and verify properties of *floating point* (FP) code.

In order to express error analysis and functional properties of FP code, we introduce for each floating point variable a companion ghost variable, thus providing us with the ability to reason about an idealised arithmetic and compare

* This research has been sponsored by Praxis High Integrity Systems Ltd and EPSRC.

```
1  procedure Patriot (X,Y : in Float; N : in Natural; R : out Float);
2  --# derives R from X,Y,N;
3  --# post R = X+Float(N)*Y and
4  --#       abs(R!) <= ((1.0+Eps)**N-1.0)*(abs(X)+Float(N)*abs(Y))+
5  --#                  (1.0+Eps)**N*(abs(X!)+Float(N)*abs(Y!));
```

Fig. 1. `Patriot` program specification.

```
1  procedure Patriot (X,Y : in Float; N : in Natural; R : out Float)
2  is
3      I : Natural;
4      F : Float;
5  begin
6      I := 0;
7      R := X;
8      loop
9          exit when N <= I;
10         --# assert N > I and
11         --#         R = X+Float(I)*Y and
12         --#         abs(R!) <= ((1.0+Eps)**I-1.0)*(abs(X)+Float(I)*abs(Y))+
13         --#                    (1.0+Eps)**I*(abs(X!)+Float(I)*abs(Y!));
14         I := I+1;
15         F := R+Y;
16         R := F;
17     end loop;
18 end Patriot;
```

Fig. 2. `Patriot` program body.

it with the floating point code. The resulting VCs quantify the error in the returned floating point values in terms of the error of the imported values. The approach has previously been described e.g. in [2].

The specification and body of the `Patriot` program, abstracting the famous patriot missile FP error [3], is presented in Figs. 1 and 2, where the error companion variable of a program FP variable `X` is denoted `X!`. A VC generated for the execution path around the loop is shown in Fig. 3, illustrating the complexity of VCs that can arise from even simple functional specifications for FP code.

Programs with nontrivial FP code will, when analysed in this way, generate VCs that contain very large and possibly complex real arithmetic expressions. The VCs concerning FP properties are essentially first-order logic expressions with real inequalities as atoms, that is, *first-order numerical constraint satisfaction problems* (FCSPs). As such, it should be possible to evaluate them using the type of numerical constraint solvers that have enjoyed increasing attention in the past decade. These solvers have become very able at solving certain types of benchmark problems, see e.g. [4]. Nevertheless, to our best knowledge, their performance on FP VCs has not been investigated. Initial tests suggest that such solvers will need to be further developed in order to become useful as part of a software development process.

For the purpose of evaluation, we chose RealPaver (version 0.4) [5] as it is the most advanced freely available *numerical constraint satisfaction problem*

$$
\begin{aligned}
|r!| &\leq \left((1+\varepsilon)^i - 1\right)(|x| + i|y|) + (1+\varepsilon)^i(|x!| + i|y!|) \\
|f!| &\leq \varepsilon|x + iy + y| + (1+\varepsilon)|r! + y!| \\
&\Rightarrow \\
|f!| &\leq \left((1+\varepsilon)^{i+1} - 1\right)(|x| + (i+1)|y|) + (1+\varepsilon)^{i+1}(|x!| + (i+1)|y!|)
\end{aligned}
$$

Fig. 3. Essential part of `Patriot` loop VC.

(NCSP) solver. Although it is possible to express disjunctions within the constraint language of RealPaver, it does not support quantified expressions. VCs for code involving arrays will usually involve quantification over the index set, and therefore, a dedicated solver will need to handle full FCSPs. Specifications detailing functional properties will in general have a nontrivial boolean structure, resulting in VCs with a complex boolean part. A dedicated solver will therefore need to handle VCs with complex boolean, as well as numeric, parts. There is a freely available numeric solver that handles large boolean first-order structures, called RSolver [6], it has however not been optimised for handling complex numerical expressions.

The shortcomings of interval arithmetic based solvers are mainly due to a *precision loss* incurred when using interval arithmetic. The loss has two main sources, the so-called *wrapping effect*, arising from the approximation of geometric objects by axis parallel boxes, and the so called *dependency problem*, which is caused by the treatment of multiple occurrences of a variable as different variables. Wrapping effects were already addressed in [7], and have been studied extensively since. Solutions to the dependency problem have been based on the introduction of relational information to interval arithmetic, see e.g. [8,9].

When a NCSP solver cannot decide the truth value of a predicate over the real numbers (in practice an inequality), it subdivides the domain. Repeated subdivisions incur an exponential computational cost, thus limiting the number of variables that can practically be handled. Previous approaches suggest delaying splitting by use of (interval based) domain narrowing algorithms [5,4,10]. Their efficiency is however often limited, as they still use interval arithmetic, and therefore ultimately suffer from the precision loss described above.

1 Our Position

We propose to use a higher precision numerical type and build a dedicated solver on it. The increased computational complexity of such a type should be outweighed by delaying the intrinsically exponential cost of domain splitting.

Such an approach has previously been been suggested in [11], where so called *Taylor Models* (TMs) are proposed as a way to combat the dependency problem. However, the TM approach has been criticised as a general solution to the

precision loss problem [12], and have so far had documented success mainly in the context of differential equation solving.

We propose to use *polynomial function enclosures* (PFEs) that extend the TM approach in two ways:

1. TMs are essentially intervals centred around polynomial approximations, i.e. they use one polynomial to approximate the function and translate it up and down to obtain a safe enclosure. PFEs use different polynomials as upper and lower bounds for a function.
2. TMs represent polynomials using its coefficients in the power basis $1, x, x^2, \ldots$ while our implementation represents polynomials in the Chebyshev basis of the first kind $1, x, 2x^2 - 1, \ldots$.

We believe that our PFE approach is more promising than TMs for reducing the dependency problem in constraint solving. Using two different polynomials as bounds yields a more accurate multiplication as discussed in Section 2.3. Using the Chebyshev basis should result in smaller truncation error when reducing polynomial order as suggested by Neumaier in [12] based on [13]. However, arithmetic over PFEs is more expensive than arithmetic over TMs.

To our knowledge, our use of PFE arithmetic is novel in the context of numerical constraint solving where interval methods currently dominate. We argue that the computational overhead, when using PFEs instead of interval arithmetic, is worthwhile in the context of verification as long as the overhead is polynomial and the improvement in precision is at least linear. In this case, the improvement in precision will translate to an exponential saving in terms of domain splitting, outweighing the polynomial increase in individual arithmetic operations. We expect that this improvement will be more noticeable with increasing problem dimension, making it particularly relevant to real-life software verification problems because such problems usually involve at least six variables (sometimes much more), which is still outside the scope of existing NCSP solvers.

2 PFE Arithmetic

Assume we are investigating an inequality $f(\boldsymbol{x}) \leq g(\boldsymbol{x})$ where $\boldsymbol{x} = (x_1, \ldots, x_n)$ and for each $1 \leq i \leq n$ the domain of the variable x_i is an interval $I_i \subset \mathbb{R}$. Using the notation $\boldsymbol{I} = \prod_{1 \leq i \leq n} I_i$ for the complete domain, we can simply say $\boldsymbol{x} \in \boldsymbol{I}$.

To prove or disprove the inequality, we use enclosures for both functions f and g. An enclosure of a function f is a pair of simpler functions, in our case polynomials $[p_{\mathrm{L}}, p_{\mathrm{U}}]$ such that

$$p_{\mathrm{L}}(\boldsymbol{x}) \leq f(\boldsymbol{x}) \leq p_{\mathrm{U}}(\boldsymbol{x}) \text{ for all } \boldsymbol{x} \in \boldsymbol{I}$$

We usually require that the polynomials in enclosures do not exceed certain maximum degree. The coefficients are taken from $\mathbb{F}_g$, which is the set of all binary floating point numbers with g-bits long mantissa and g-bits long exponent. We

call the number g the *granularity* of the floating point number. The granularity of the enclosure coefficients is determined from the granularity used to express the endpoints of the domain $\boldsymbol{I}$. 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.

We have limited the polynomial operations on PFEs in our current prototype to: projections (modelling variables), point-wise addition, multiplication, exponentiation and absolute value. To keep the degree within the current limit, we often need to reduce the degree of a polynomial. We will describe this operation first.

2.1 Degree reduction

To easily reduce the degree with a close-to-optimal loss of precision, we always represent the enclosure polynomials using Chebyshev polynomials of the first kind, denoted $T_i(x)$[14].

Most of the nice properties of Chebyshev polynomials apply only when restricted to the domain $[-1, 1]$. Therefore, we use the canonical affine transformation of the domain $\boldsymbol{I}$ to the unit domain $[-1, 1]^n$ and build the enclosures over the unit domain instead of the original domain and compose it with the translation when the correct domain is important, e.g. when defining a projection $\boldsymbol{x} \mapsto x_i$. From now on we will assume that p_{L} and p_{U} are considered on the unit domain and variables in the original expression f are translated to certain affine polynomials over the unit domain.

Thus the general format used for an enclosing polynomial of degree at most m is:

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

Sometimes we will refer to such a polynomial briefly using the notation $P_m\{a_{\boldsymbol{j}}\}$.

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

$$\deg_{\mathrm{m}'}(p)(\boldsymbol{x}) = \left(\sum_{0 \le j_1 + \cdots + j_n \le m'} a_{\boldsymbol{j}} \cdot \prod_{1 \le i \le n} T_{j_i}(x_i) \right) \oplus_{\mathrm{U}} \sum\nolimits_{\mathrm{U}}{}_{m' < j_1 + \cdots + j_n \le m} |a_{\boldsymbol{j}}|$$

where $\oplus_{\mathrm{U}}$ and $\sum_{\mathrm{U}}$ refers to an upwards rounded addition of coefficients in the constant term (recall $T_0(x) = 1$).

This reduction is majoring p thanks to $T_i(x) : [-1, 1] \to [-1, 1]$. Also, this reduction is very close to p thanks to other properties of Chebyshev polynomials[14]. If rounding downwards, the last sum in the formula will be subtracted rounding downwards (i.e. $\ominus_{\mathrm{L}}$) instead of added.

Reduction to degree 1 (affine) is also used when comparing two enclosures corresponding to two sides of an inequality. The lower bound of one side is subtracted from the upper bound of the other side and the resulting polynomial

is reduced to an affine one rounding downwards and checked for positivity on the unit domain.

2.2 Addition

Adding two PFEs is fairly simple, it amounts to adding the upper bound polynomials rounding upwards and adding the lower bound polynomials rounding downwards: $[p_\mathrm{L}, p_\mathrm{U}] + [q_\mathrm{L}, q_\mathrm{U}] = [p_\mathrm{L} \oplus_\mathrm{L} p_\mathrm{L}, p_\mathrm{U} \oplus_\mathrm{U} p_\mathrm{U}]$.

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

$$P_m\{a_{\boldsymbol{j}}\} \oplus_\mathrm{U} P_m\{b_{\boldsymbol{j}}\} = P_m\{a_{\boldsymbol{j}} \oplus_\mathrm{L} b_{\boldsymbol{j}}\} \oplus_\mathrm{U} \sum\nolimits_\mathrm{U} ((a_{\boldsymbol{j}} \oplus_\mathrm{U} b_{\boldsymbol{j}}) \ominus_\mathrm{U} (a_{\boldsymbol{j}} \oplus_\mathrm{L} b_{\boldsymbol{j}}))$$

where each $\oplus_\mathrm{U}$ refers to an upwards rounded addition and each $\oplus_\mathrm{L}$ refers to a downwards rounded addition in the appropriate type. Also the summation symbol and the subtraction are upwards rounding.

This 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 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.

2.3 Multiplication

Multiplication is much more complicated than addition even before any floating-point 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 4. Notice that with multiplication it is important that we use independent polynomials for lower and upper bounds. If we were to use TMs, we would need to use a much worse enclosure in Figure 4, most likely the constant enclosure $[-2, 2]$. 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 multiplication of enclosures as follows:

$$\begin{aligned}[p_\mathrm{L}, p_\mathrm{U}] \cdot [q_\mathrm{L}, q_\mathrm{U}] = \\ &[\min_\mathrm{L}(p_\mathrm{L} \otimes_\mathrm{L} p_\mathrm{L}, p_\mathrm{L} \otimes_\mathrm{L} p_\mathrm{U}, p_\mathrm{U} \otimes_\mathrm{L} p_\mathrm{L}, p_\mathrm{U} \otimes_\mathrm{L} p_\mathrm{U}), \\ &\max_\mathrm{U}(p_\mathrm{L} \otimes_\mathrm{U} p_\mathrm{L}, p_\mathrm{L} \otimes_\mathrm{U} p_\mathrm{U}, p_\mathrm{U} \otimes_\mathrm{U} p_\mathrm{L}, p_\mathrm{U} \otimes_\mathrm{U} p_\mathrm{U})].\end{aligned}$$

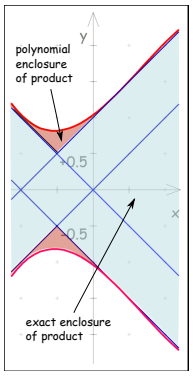

Fig. 4. An approximation of the enclosure product $(x, x+1)\cdot(-1, 1)$ on the unit domain $[-1, 1]$. The exact product is enclosed by the maximum and minimum of all four slanted straight lines.
A TM represetation of the product would be an centered enclosure of width ≥ 4.

To complete the definition, we should say how individual polynomials are multiplied rounding point-wise upwards or downwards and, more crucially, how we estimate the minimum and maximum of multiple polynomials as a polynomial. Nevertheless, the full details of these operations are beyond the scope of this paper. Polynomial multiplication is worked out in a similar way as addition of polynomials, except that it turns out a bit more complex as one has to pairwise multiply all terms using the rule $T_i(x) \cdot T_j(x) = T_{i+j}(x)/2 + T_{|i-j|}(x)/2$. Both maximisation and minimisation is reduced to a maximisation of the type $\max(0, p)$, which is approximated from below and from above using a pair of cubic polynomials. These cubic polynomials are determined by their values and derivatives in the endpoints of the range of p whenever the range contains zero. In the negative endpoint both the value and the derivative of the cubic polynomial is 0 and in the positive endpoint, the value is equal to the endpoint itself and the derivative is 1. In some cases the cubic polynomial exceeds its intended boundaries (e.g. it is slightly negative). When this happens, we shift the polynomial slightly to make it truly maximise or minimise the function $\max(0, p)$.

Both multiplication of polynomials and the maximisation operation require the results to have their polynomial degrees reduced to the current limit.

3 PFE Based Solver

Our prototype solver evaluates inequalities by approximating their numeric expressions with PFEs. If the enclosures are sufficiently separated, then the truth value of the inequality can be determined. If the enclosures are too close or intersecting, then the solver bisects the domain of the expressions along the widest variable domain, increases the precision of the underlying numeric type, and attempts to determine the inequality over the sub-domains. Currently, comparison of PFEs is implemented by linearisation of the function bounds. It is an interesting question whether the increase in precision obtained by reducing the bounds to quadratics would warrant the considerable computational overhead thus incurred.

$$\begin{cases} r! \leq \left((1+\varepsilon)^i - 1\right) iy + (1+\varepsilon)^i iy! \\ f! \leq \varepsilon(iy+y) + (1+\varepsilon)(r!+y!) \\ f! \geq \left((1+\varepsilon)^{i+1} - 1\right)(i+1)y + (1+\varepsilon)^{i+1}(i+1)y! \end{cases}$$

$$y \in [1,10]\ y! \in [0,0.125]\ r!, f! \in [0,100]\ i \in [1,100]$$

Fig. 5. Patriot problem.

$$\begin{cases} (x^2+y^2+12x+9)^2 \leq 4(2x+3)^3 \\ x^2+y^2 \geq 2 \end{cases}$$

$$x, y \in [-2,2]$$

Fig. 6. Problem F2.2

4 Experiments

In Fig. 5 we show a simplified version of the negated loop VC for the `Patriot` program (see also Fig. 3). Neither RealPaver nor our PFE based prover succeeded in exhausting the entire search space, and were aborted after one hour. The PFE based prover discarded over 99.5 percent of the total volume when allowed up to 22 domain subdivisions (i.e. the tree depth), which is the most it reached within the prescribed time limit.

We evaluated RealPaver and our PFE based solver on two benchmarks F2.2 and F2.3 from [15], shown in Figs. 6 and 7 on an Intel Pentium 2.8 GHz dual core with 2GB RAM and 2MB cache. In Fig. 8 we plot the logarithm of execution time as a function of depth for the two benchmarks with RealPaver set to `Bisection mode = paving, number = +oo` and our solver using enclosures of degree zero and one.

In Fig. 9 we present relative inner and outer volumes computed by the PFE based prover for problems F2.2 and F2.3. The titles "depth", "degree", "inner", "outer" and "time" refer to the the maximum number of splittings, the degree of PFE bounds, relative volume of inner boxes (in tenths of a percent), relative volume of outer boxes (in tenths of a percent) and times for each computation (in milliseconds), respectively. Note that the timings include volume computations, decreasing the apparent performance at higher degrees. Fig. 10 illustrates the reduction in the number of splittings for problem F2.2, at degrees zero and two, and depths 10, 12 and 14.

In Fig. 11 we show the minimum depth that was required for our PFE based solver to compute a nonempty set of inner boxes, for degrees zero to three. The timings show that there is an overall improvement in performance for higher degree PFEs, the optimal degree seems, rather unsurprisingly, to be problem dependent. These results imply that higher degree PFEs can yield faster counter example generation.

$$\begin{cases} x^3 + y^3 \geq 3xy \\ x^2 + y^2 \geq 0.1 \\ (x^2 + y^2)\left(y^2 + x(x+1)\right) \leq 4xy^2 \end{cases}$$

$$x, y \in [-3, 3]$$

Fig. 7. Problem F2.3.

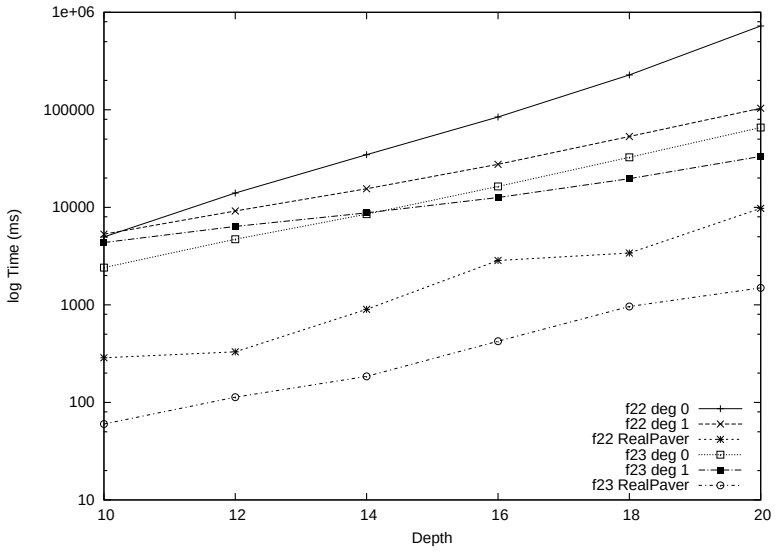

Fig. 8. Depth-log Time plot for F2.2 and F2.3 comparing RealPaver with our PFE based solver using 0 and 1 degree enclosures.

The experiments have shown that increasing the PFE degree can be an effective way of reducing domain splitting in a bisection based algorithm. The most drastic performance increase was observed in identifying counter examples, it may however depend heavily on the choice of search strategy. Extensive experiments are needed to clarify this matter.

5 Conclusions

The main contribution of this paper is a novel implementation of PFE arithmetic, and what to our knowledge is a first implementation of a numerical constraint solver based on PFEs. We believe that we have shown that the approach warrants further investigation. Clearly, our implementation is a first prototype,

depth	degree	F2.2 inner (‰)	F2.2 outer (‰)	F2.2 time (ms)	F2.3 inner (‰)	F2.3 outer (‰)	F2.3 time (ms)
10	0	0	496	4974	0	132	2412
10	1	31	148	5300	0	68	4361
10	2	31	144	4971	0	68	3902
10	3	31	144	5024	0	68	3865
12	0	11	303	14042	0	53	4704
12	1	49	70	9166	1	24	6368
12	2	50	67	8818	1	24	5874
12	3	50	67	8903	1	24	5847
14	0	32	161	34646	0	24	8525
14	1	66	32	15523	3	9	8772
14	2	66	32	15654	3	9	8471
14	3	66	32	15789	3	9	8435
16	0	44	95	84497	1	12	16366
16	1	73	16	27599	4	4	12629
16	2	73	16	28882	4	4	12713
16	3	73	16	29144	4	4	12712
18	0	51	61	228085	3	6	32617
18	1	78	8	53205	5	1	19660
18	2	78	7	57657	5	1	20392
18	3	78	7	58175	5	1	21127

Fig. 9. PFE solver computations of inner and outer volumes.

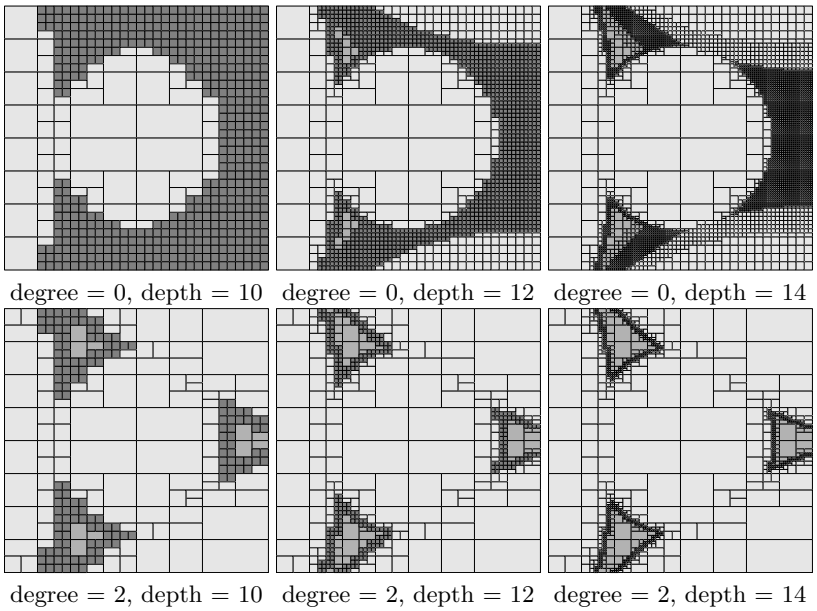

Fig. 10. Solver output for problem F2.2.

F2.2			F2.3		
degree	**depth**	**time**	**degree**	**depth**	**time**
0	11	7.96	0	15	11.18
1	9	4.14	1	11	5.68
2	9	3.93	2	11	5.14
3	9	3.94	3	11	5.11

Fig. 11. Minimum depth at which a counterexample is identified.

and as such cannot be expected to compete with mature solvers on standard benchmarks.

The numerical experiments varying the PFE degree show that there are cases for which there is a clear overall improvement in performance over naive interval arithmetic. It remains to be investigated whether this improvement carries over when domain narrowing techniques, such as constraint inversion, are included in the solver algorithm.

6 Further Work

We have shown that PFEs have the potential to improve the performance of numerical constraint solvers. However, much work remains before this potential can be realised. The primary objectives of future work on PFEs are to implement further PFE operations (mainly division and other elementary functions), as well as a rigorous analysis of the theoretical aspects of PFE computation.

In order to compare the performance of PFE based solvers to the state-of-art interval solvers, it is necessary to discover and implement intelligent handling of FCSPs in the spirit of the NCSP techniques RealPaver is using. At the moment, most of the information contained in the PFE approximations is ignored when evaluating inequalities. It is therefore desirable to develop a different comparison operator for PFEs, i.e. one with a more informative return type, which would enable more intelligent search strategies to be devised. Such improvements would enable the development of analogues to domain narrowing algorithms that, once integrated with advanced search strategies, could move the technology closer to our ultimate goal — the automation of proofs for VCs arising from FP code.

References

1. Barnes, J.: High Integrity Software: The SPARK Approach to Safety and Security. 2 edn. Addison-Wesley (2003)
2. Boldo, S., Filliâtre, J.C.: Formal verification of floating-point programs. In: Proceedings of 18th IEEE International Symposium on Computer Arithmetic, Montpellier, France (2007)

3. Abernethy, K., Allen, T.: Social themes: Risks in numeric computing. `http://cs.furman.edu/digitaldomain/themes/risks/risks_numeric.htm` (2008) Accessed on 21st May 2008.
4. Shcherbina, O., Neumaier, A., Sam-Haroud, D., Vu, X.H., Nguyen, T.V.: Numerical constraint satisfaction problems with non-isolated solutions. In Bliek, C., Jermann, C., Neumaier, A., eds.: Global Optimization and Constraint Satisfaction: Global Optimization and Constraint Satisfaction: First International Workshop on Global Constraint Optimization and Constraint Satisfaction, COCOS 2002, Valbonne-Sophia Antipolis, France, October 2-4, 2002, Revised Selected Papers. Volume LNCS 2861., Springer-Verlag (2003) 211–222
5. Granvilliers, L., Benhamou, F.: Algorithm 852: RealPaver: An interval solver using constraint satisfaction techniques. ACM Transactions on Mathematical Software **32** (2006)
6. Ratschan, S., et al.: RSolver. `http://rsolver.sourceforge.net` (2004) Software Package.
7. Moore, R.E.: Automatic local coordinate transformations to reduce the growth of error bounds in interval computation of solutions of ordinary differential equations. In: volume II of Error in Digital Computations. John Wiley and Sons inc, New York (1965) 103–140
8. de Figueiredo, L.H., Stolfi, J.: Affine arithmetic: Concepts and applications: Scan2002 international conference (guest editors: Rene alt and jean-luc lamotte). Numerical Algorithms **37** (December 2004) 147–158(12)
9. Miné, A.: Relational abstract domains for the detection of floating-point run-time errors. In: ESOP'04. Volume 2986 of LNCS., Springer (2004) 3–17
10. Benhamou, F., Goualard, F., Granvilliers, L., Puget, J.F.: Revising hull and box consistency. In: Proceedings of the 1999 international conference on Logic programming, Cambridge, MA, USA, Massachusetts Institute of Technology (1999) 230–244
11. Makino, K., Berz, M.: Efficient control of the dependency problem based on taylor model methods. Reliable Computing **5** (February 1999) 3–12(10)
12. Neumaier, A.: Taylor forms–use and limits. Reliable Computing **9** (2003) 43–79
13. Kaucher, E., Miranker, W.L.: Validating computation in a function space. (1988) 403–425
14. Mason, J.C., Handscomb, D.C.: Chebyshev Polynomials. CRC Press (2002)
15. Vu, X.H.: Rigorous Solution Techniques for Numerical Constraint Satisfaction Problems. PhD thesis, Swiss Federal Institute of Technology in Lausanne (EPFL), Department of Computer Science (2005)