A Mixed Real and Floating-Point Solver

Rocco Salvia[1], Laura Titolo[2], Marco A. Feliú[2], Mariano M. Moscato[2], César A. Muñoz[3], and Zvonimir Rakamarić[1]

[1] University of Utah*,
{rocco,zvonimir}@cs.utah.edu
[2] National Institute of Aerospace**,
{laura.titolo,marco.feliu,mariano.moscato}@nianet.org
[3] NASA Langley Research Center,
cesar.a.munoz@nasa.gov

Abstract. Reasoning about mixed real and floating-point constraints is essential for developing accurate analysis tools for floating-point programs. This paper presents FPRoCK, a prototype tool for solving mixed real and floating-point formulas. FPRoCK transforms a mixed formula into an equisatisfiable one over the reals. This formula is then solved using an off-the-shelf SMT solver. FPRoCK is also integrated with the PRECiSA static analyzer, which computes a sound estimation of the round-off error of a floating-point program. It is used to detect infeasible computational paths, thereby improving the accuracy of PRECiSA.

1 Introduction

Floating-point numbers are frequently used as an approximation of real numbers in computer programs. A round-off error originates from the difference between a real number and its floating-point representation, and accumulates throughout a computation. The resulting error may affect both the computed value of arithmetic expressions as well as the control flow of the program. To reason about floating-point computations with possibly diverging control flows, it is essential to solve mixed real and floating-point arithmetic constraints. This is known to be a difficult problem. In fact, constraints that are unsatisfiable over the reals may hold over the floats and vice-versa. In addition, combining the theories is not trivial since floating-point and real arithmetic do not enjoy the same properties.

Modern *Satisfiability Modulo Theories* (SMT) solvers, such as Mathsat [3] and Z3 [11], encode floating-point numbers with bit-vectors. This technique is usually inefficient due to the size of the binary representation of floating-point numbers. For this reason, several abstraction techniques have been proposed to approximate floating-point formulas and to solve them in the theory of real numbers. Approaches based on the *counterexample-guided abstraction refinement* (CEGAR) framework [2,14,18] simplify a floating-point formula and solve it in a

* Partially supported by NSF awards CCF 1346756 and CCF 1704715.
** Research by the first four authors was supported by the National Aeronautics and Space Administration under NASA/NIA Cooperative Agreement NNL09AA00A.

proxy theory that is more efficient than the original one. If a model is found for the simplified formula, a check on whether this is also a model for the original formula is performed. If it is, the model is returned, otherwise, the proxy theory is refined. Realizer [9] is a framework built on the top of Z3 to solve floating-point formulas by translating them into equivalent ones in real arithmetic. Molly [14] implements a CEGAR loop where floating-point constraints are lifted in the proxy theory of mixed real and floating-point arithmetics. To achieve this, it uses an extension of Realizer that supports mixed real and floating-point constraints. However, this extension is embedded in Molly and cannot be used as a standalone tool. The Colibri [10] solver handles the combination of real and floating-point constraints by using disjoint floating-point intervals and difference constraints. Unfortunately, the publicly available version of Colibri does not support all the rounding modalities and the negation of Boolean formulas. JConstraints [7] is a library for constraint solving that includes support for floating-points by encoding them into reals.

This paper presents a prototype solver for mixed real and floating-point constraints called FPRoCK.[4] It extends the transformation defined in Realizer [9] to mixed real/floating-point constraints. Given a mixed real-float formula, FPRoCK generates an equisatisfiable real arithmetic formula that can be solved by an external SMT solver. In contrast to Realizer, FPRoCK supports mixed-precision floating-point expressions and different ranges for the input variables. FPRoCK is also employed to improve the accuracy of the static analyzer PRECiSA [16]. In particular, it identifies spurious execution traces whose path conditions are unsatisfiable, which allows PRECiSA to discard them.

2 Solving Mixed Real/Floating-Point Formulas

A *floating-point number* [8], or simply a *float*, can be represented by a tuple (s, m, exp) where s is a sign bit, m is an integer called the *significand* (or *mantissa*), and exp is an integer *exponent*. A float (s, m, exp) encodes the real number $(-1)^s \cdot m \cdot 2^{exp}$. Henceforth, $\mathbb{F}$ represents the set of floating-point numbers. Let $\tilde{v}$ be a floating-point number that represents a real number r. The difference $|\tilde{v} - r|$ is called the *round-off error* (or *rounding error*) of $\tilde{v}$ with respect to r. Each floating-point number has a format f that specifies its dimensions and precision, such as single or double. The expression $F_f(r)$ denotes the floating-point number in format f *closest* to r assuming a given rounding mode.

Let $\mathbb{V}$ and $\widetilde{\mathbb{V}}$ be two disjoint sets of variables representing real and floating-point values respectively. The set $\mathbb{A}$ of mixed arithmetic expressions is defined by the grammar

$$A ::= d \mid x \mid \tilde{d} \mid \tilde{x} \mid A \odot A \mid A \widetilde{\odot} A \mid F_f(A),$$

where $d \in \mathbb{R}$, $x \in \mathbb{V}$, $\odot \in \{+, -, *, /, |\cdot|\}$ (the set of basic real number arithmetic operators), $\tilde{d} \in \mathbb{F}$, $\tilde{x} \in \widetilde{\mathbb{V}}$, $\widetilde{\odot} \in \{\tilde{+}_f, \tilde{-}_f, \tilde{*}_f, \tilde{/}_f\}$ (the set of basic floating-point arithmetic operators) and $f \in \{single, double\}$ denotes the desired precision for the

[4] The FPRoCK distribution is available at `https://github.com/nasa/FPRoCK`.

result. The rounding operator F_f is naturally extended to arithmetic expressions. According to the IEEE-754 standard [8], each floating-point operation is computed in exact real arithmetic and then rounded to the nearest float, i.e., $A \widetilde{\odot}_f A = F_f(A \odot A)$. Since floats can be exactly represented as real numbers, an explicit transformation is not necessary. The set of mixed real-float Boolean expressions $\mathbb{B}$ is defined by the grammar

$$B ::= \mathit{true} \mid \mathit{false} \mid B \wedge B \mid B \vee B \mid \neg B \mid A < A \mid A = A,$$

where $A \in \mathbb{A}$.

The input to FPRoCK is a formula $\tilde{\phi} \in \mathbb{B}$ that may contain both real and floating-point variables and arithmetic operators. Each variable is associated with a type (real, single or double precision floating-point) and range that can be either bounded, e.g., $[1, 10]$, or unbounded, e.g., $[-\infty, +\infty]$. The precision of a mixed-precision floating-point arithmetic operation is automatically detected and set to the maximum precision of its arguments. Given a mixed formula $\tilde{\phi} \in \mathbb{B}$, FPRoCK generates a formula ϕ over the reals such that $\tilde{\phi}$ and ϕ are equisatisfiable. Floating-point expressions are transformed into equivalent real-valued expressions using the approach presented in [9], while the real variables and operators are left unchanged. It is possible to define $x \widetilde{\odot} y$ as

$$x \widetilde{\odot} y = \left(\frac{\rho\left(\frac{x \odot y}{2^{exp}} \cdot 2^p\right)}{2^p} \right) \cdot 2^{exp}, \tag{2.1}$$

where p is the precision of the format, $exp = max\{i \in \mathbb{Z} \mid 2^i \leq |x \odot y|\}$, and $\rho : \mathbb{R} \to \mathit{Int}$ is a function implementing the rounding modality [9]. Therefore, given a floating-point formula $\tilde{\phi}$, an equisatisfiable formula without floating-point operators is obtained by replacing every occurrence of $x \widetilde{\odot} y$ using Equation (2.1). This is equivalent to replacing the occurrences of $x \widetilde{\odot} y$ with a new fresh real-valued variable v and imposing $v = x \widetilde{\odot} y$. From Equation (2.1) it follows that $v \cdot 2^{p-exp} = \rho((x \odot y) \cdot 2^{p-exp})$. Thus, the final formula ϕ is

$$\phi := \tilde{\phi}[v / x \widetilde{\odot} y] \wedge v \cdot 2^{p-exp} = \rho((x \odot y) \cdot 2^{p-exp}), \tag{2.2}$$

where $\tilde{\phi}[v / x \widetilde{\odot} y]$ denotes the Boolean formula $\tilde{\phi}$ where all the occurrences of $x \widetilde{\odot} y$ are replaced by v. The precision p is a constant that depends on the chosen floating-point format, while exp is an integer representing the exponent of the binary representation of $x \widetilde{\odot} y$.

To find an assignment for the exponent exp, FPRoCK performs in parallel a sequential and binary search over the dimension of $x \widetilde{\odot} y$, as opposed to the simple sequential search implemented in Realizer. The implementation of the function ρ depends on the selected rounding mode and can be defined using floor and ceiling operators (see [9] for details). Therefore, the transformed formula ϕ does not contain any floating-point operators, and hence it can be solved by any SMT solver that supports the fragment of real/integer arithmetics including floor and ceiling operators. FPRoCK uses three off-the-shelf SMT solvers as back-end procedures to solve the transformed formula: Mathsat [3], Z3 [11], and

CVC4 [1]. Optionally, the constraint solver Colibri [10] is also available for use within FPRoCK. FPRoCK provides the option to relax the restriction on the minimum exponent to handle subnormal floats. This solution is sound in the sense that it preserves the unsatisfiability of the original formula. However, if this option is used, it is possible that FPRoCK finds an assignment to a float that is not representable in the chosen precision, and therefore is not a solution for the original formula. Furthermore, FPRoCK currently does not support special floating-point values such as *NaN* and *Infinity*.

3 Integrating FPRoCK in PRECiSA

PRECiSA[5] (Program Round-off Error Certifier via Static Analysis) [16] is a static analyzer based on abstract interpretation [4]. PRECiSA accepts as input a floating-point program and automatically generates a sound over-approximation of the floating-point round-off error and a proof certificate in the Prototype Verification System (PVS) [13] ensuring its correctness. For every possible combination of real and floating-point execution paths, PRECiSA computes a *conditional error bound* of the form $\langle \eta, \widetilde{\eta} \rangle \twoheadrightarrow (r, e)$, where η is a symbolic path condition over the reals, $\widetilde{\eta}$ is a symbolic path condition over the floats, and r, e are symbolic arithmetic expressions over the reals. Intuitively, $\langle \eta, \widetilde{\eta} \rangle \twoheadrightarrow (r, e)$ indicates that if the conditions η and $\widetilde{\eta}$ are satisfied, the output of the program using exact real number arithmetic is r and the round-off error of the floating-point implementation is bounded by e.

PRECiSA initially computes round-off error estimations in symbolic form so that the analysis is modular. Given the initial ranges for the input variables, PRECiSA uses the Kodiak global optimizer [12] to maximize the symbolic error expression e. Since the analysis collects information about real and floating-point execution paths, it is possible to consider the error of taking the incorrect branch compared to the ideal execution using real arithmetic. This happens when the guard of a conditional statement contains a floating-point expression whose round-off error makes the actual Boolean value of the guard differ from the value that would be obtained assuming real arithmetic. When the floating-point computation diverges from the real one, it is said to be *unstable*.

For example, consider the function $sign(\tilde{x}) = \mathit{if}\ \tilde{x} \geq 0\ \mathit{then}\ 1\ \mathit{else}\ -1$. PRECiSA computes a set of four different conditional error bounds: $\{\langle \chi_r(\tilde{x}) \geq 0, \tilde{x} \geq 0 \rangle \twoheadrightarrow (r = 1, e = 0), \langle \chi_r(\tilde{x}) < 0, \tilde{x} < 0 \rangle \twoheadrightarrow (r = -1, e = 0), \langle \chi_r(\tilde{x}) \geq 0, \tilde{x} < 0 \rangle \twoheadrightarrow (r = -1, e = 2), \langle \chi_r(\tilde{x}) < 0, \tilde{x} \geq 0 \rangle \twoheadrightarrow (r = 1, e = 2)\}$. The function $\chi_r : \widetilde{\mathbb{V}} \to \mathbb{V}$ associates with the floating-point variable $\tilde{x}$ a variable $x \in \mathbb{V}$ representing the real value of $\tilde{x}$. The first two elements correspond to the cases where real and floating-point computational flows coincide. In these cases, the error is 0 since the output is an integer number with no rounding error. The other two elements model the unstable paths. In these cases, the error is 2, which corresponds to the difference between the output of the two branches. PRECiSA may produce

[5] The PRECiSA distribution is available at `https://github.com/nasa/PRECiSA`.

conditional error bounds with unsatisfiable symbolic conditions (usually unstable), which correspond to execution paths that cannot take place. The presence of these spurious elements affects the accuracy of the computed error bound. For instance, in the previous example, if $|\chi_r(\tilde{x}) - \tilde{x}| \leq 0$ both unstable cases can be removed, and the overall error would be 0 instead of 2.

Real and floating-point conditions can be checked separately using SMT solvers that support real and/or floating-point arithmetic. However, the inconsistency often follows from the combination of the real and floating-point conditions. In fact, the floating-point expressions occurring in the conditions are implicitly related to their real arithmetic counterparts by their rounding error. Therefore, besides checking the two conditions separately, it is necessary to check them in conjunction with a set of constraints relating each arithmetic expression $\widetilde{expr}$ occurring in the conditions with its real number counterpart $R_{\mathbb{A}}(\widetilde{expr})$. $R_{\mathbb{A}}(\widetilde{expr})$ is defined by simply replacing in $\widetilde{expr}$ each floating-point operation with the corresponding real one and by applying χ_r to floating-point variables.

FPRoCK is suitable for solving such constraints thanks to its ability to reason about mixed real and floating-point formulas. Given a set ι of ranges for the input variables, for each conditional error bound $c = \langle \eta, \widetilde{\eta} \rangle_t \twoheadrightarrow (r, e)$ computed by PRECiSA, the following formula ψ modeling the information contained in the path conditions is checked using FPRoCK:

$$\psi \coloneqq \eta \wedge \widetilde{\eta} \wedge \bigwedge \{ |\widetilde{expr} - R_{\mathbb{A}}(\widetilde{expr})| \leq \epsilon \mid \widetilde{expr} \text{ occurs in } \widetilde{\eta}, \widetilde{expr} \notin \widetilde{\mathbb{V}}, \widetilde{expr} \notin \mathbb{F}, \epsilon = max(e)|_\iota \} \tag{3.1}$$

The value $max(e)|_\iota$ is the round-off error of $\widetilde{expr}$ assuming the input ranges in ι, and it is obtained by maximizing the symbolic error expression e with the Kodiak global optimizer. If ψ is unsatisfiable, then c is dropped from the solutions computed by PRECiSA. Otherwise, a counterexample is generated that may help to discover cases for which the computation is diverging or unsound.

Since FPRoCK currently supports only the basic arithmetic operators, while PRECiSA supports a broader variety of operators including transcendental functions, a sound approximation is needed for converting PRECiSA conditions into a valid input for FPRoCK. The proposed approach replaces in ψ each floating-point (respectively real) arithmetic expression with a fresh floating-point (respectively real) variable. This is sound but not complete, meaning it preserves just the unsatisfiability of the original formula. In other words, if $\psi[v_i/\widetilde{expr}_i]_{i=1}^n$ is unsatisfiable it follows that ψ is unsatisfiable, but if a solution is found for $\psi[v_i/\widetilde{expr}_i]_{i=1}^n$ there is no guarantee that an assignment satisfying ψ exists. This is enough for the purpose of eliminating spurious conditional bounds since it assures that no feasible condition gets eliminated. In practice, it is accurate enough to detect spurious unstable paths. When a path condition is deemed unsatisfiable by FPRoCK, PRECiSA states such unsatisfiability in the PVS formal certificate. For simple path conditions, this property can be automatically checked by PVS. Unfortunately, there are cases where human intervention is required to verify this part of the certificates.

Table 1. Experimental results showing absolute round-off error bounds and execution time in seconds (best results in bold).

Benchmark	PRECiSA		PRECiSA+FPRoCK		Rosa	
	Error	Time(s)	Error	Time(s)	Error	Time(s)
cubicSpline	2.70E+01	**0.07**	2.70E+01	97.8	**2.50E-01**	24.1
eps_line	2.00E+00	**0.02**	**1.00E+00**	48.8	2.00E+00	15.5
jetApprox	1.51E+01	**12.79**	8.11E+00	263.3	**4.97E+00**	924.8
linearFit	1.08E+00	**0.06**	5.42E-01	259.7	**3.19E-01**	12.4
los	2.00E+00	**0.02**	**1.00E+00**	46.2	not supported	n/a
quadraticFit	3.68E+00	**0.90**	3.68E+00	259.8	**1.27E-01**	82.4
sign	2.00E+00	**0.02**	**1.00E+00**	32.1	2.00E+00	4.7
simpleInterpolator	2.25E+02	**0.03**	1.16E+02	93.8	**3.33E+01**	6.3
smartRoot	**1.75E+00**	**0.32**	**1.75E+00**	0.6	not supported	n/a
styblinski	9.35E+01	**1.06**	6.66E+01	260.1	**6.55E+00**	77.0
tau	8.40E+06	**0.03**	**8.00E+06**	101.8	8.40E+06	20.7

Table 1 compares the original version of PRECiSA with the enhanced version that uses FPRoCK to detect the unsatisfiable conditions, along with the analysis tool Rosa [6] which also computes an over-approximation of the round-off error of a program. All the benchmarks are obtained by applying the transformation defined in [17] to code fragments from avionics software and the FPBench library [5]. A transformed program is guaranteed to return either the result of the original floating-point program, when it can be assured that both its real and floating-point flows agree, or a warning when these flows may diverge. The results show that FPRoCK helps PRECiSA improving the computed round-off error in 8 out of 11 benchmarks total. FPRoCK runs all search encoding (linear, binary) plus solver (MathSAT5, CVC4, Z3) combinations in parallel. It waits for all solvers to finish and performs a check on the consistency of the solutions.

4 Conclusions

This paper presents FPRoCK, a prototype tool for solving mixed real and floating-point formulas. FPRoCK extends the technique used in Realizer by adding support for such mixed formulas. FPRoCK is integrated into PRECiSA to improve its precision. Similarly, it could be integrated into other static analyzers, such as FPTaylor [15]. The current version of FPRoCK has some limitations in terms of expressivity and efficiency. Support for a vast range of operators, including transcendental functions, is contingent on the expressive power of the underlying SMT solvers. The performance of FPRoCK can be improved by returning a solution as soon as the first solver finalizes its search. However, finding an assignment for the exponent of each floating-point variable is still the major bottleneck of the analysis. The use of a branch-and-bound search to divide the state-space may help to mitigate this problem.

References

1. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Proceedings of the 23rd International Conference on Computer Aided Verification (CAV). pp. 171–177. Springer (2011)
2. Brillout, A., Kroening, D., Wahl, T.: Mixed abstractions for floating-point arithmetic. In: Proceedings of the 9th International Conference on Formal Methods in Computer-Aided Design (FMCAD). pp. 69–76. IEEE (2009)
3. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 95–109. Springer (2013)
4. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). pp. 238–252. ACM (1977)
5. Damouche, N., Martel, M., Panchekha, P., Qiu, C., Sanchez-Stern, A., Tatlock, Z.: Toward a standard benchmark format and suite for floating-point analysis. In: Proceedings of the 9th International Workshop on Numerical Software Verification (NSV). pp. 63–77. Springer (2016)
6. Darulova, E., Kuncak, V.: Sound compilation of reals. In: Proceedings of the 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). pp. 235–248. ACM (2014)
7. Howar, F., Jabbour, F., Mues, M.: JConstraints: A library for working with logic expressions in Java. In: Essays Dedicated to Bernhard Steffen on the Occasion of His 60th Birthday (2019), to appear
8. IEEE: IEEE standard for binary floating-point arithmetic. Tech. rep., Institute of Electrical and Electronics Engineers (2008)
9. Leeser, M., Mukherjee, S., Ramachandran, J., Wahl, T.: Make it real: Effective floating-point reasoning via exact arithmetic. In: Proceedings of the 17th Design, Automation & Test in Europe Conference & Exhibition, (DATE). pp. 1–4. IEEE (2014)
10. Marre, B., Bobot, F., Chihani, Z.: Real behavior of floating point numbers. Proceedings of the 15th International Workshop on Satisfiability Modulo Theories (SMT). (2017)
11. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 337–340. Springer (2008)
12. Narkawicz, A., Muñoz, C.: A formally verified generic branching algorithm for global optimization. In: Proceedings of the 5th International Conference on Verified Software: Theories, Tools, Experiments (VSTTE). pp. 326–343. Springer (2013)
13. Owre, S., Rushby, J., Shankar, N.: PVS: A prototype verification system. In: Proceedings of the 11th International Conference on Automated Deduction (CADE). pp. 748–752. Springer (1992)
14. Ramachandran, J., Wahl, T.: Integrating proxy theories and numeric model lifting for floating-point arithmetic. In: Proceedings of the 16th International Conference on Formal Methods in Computer-Aided Design, (FMCAD). pp. 153–160. FMCAD Inc (2016)
15. Solovyev, A., Jacobsen, C., Rakamaric, Z., Gopalakrishnan, G.: Rigorous estimation of floating-point round-off errors with Symbolic Taylor Expansions. In: Pro-

ceedings of the 20th International Symposium on Formal Methods (FM). pp. 532–550. Springer (2015)
16. Titolo, L., Feliú, M., Moscato, M., Muñoz, C.: An abstract interpretation framework for the round-off error analysis of floating-point programs. In: Proceedings of the 19th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI). pp. 516–537. Springer (2018)
17. Titolo, L., Muñoz, C., Feliú, M., Moscato, M.: Eliminating unstable tests in floating-point programs. In: Proceedings of the 28th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR). pp. 169–183. Springer (2018)
18. Zeljic, A., Backeman, P., Wintersteiger, C.M., Rümmer, P.: Exploring approximations for floating-point arithmetic using UppSAT. In: Proceedings of the 9th International Joint Conference on Automated Reasoning (IJCAR). pp. 246–262. Springer (2018)