Make it Real: Effective Floating-Point Reasoning via Exact Arithmetic

Miriam Leeser, Saoni Mukherjee
Electrical and Computer Engineering
Northeastern University
{mel|saoni}@coe.neu.edu

Jaideep Ramachandran, Thomas Wahl
College of Computer and Information Science
Northeastern University
{jaideep|wahl}@ccs.neu.edu

***Abstract*—Floating-point arithmetic is widely used in scientific computing. While many programmers are subliminally aware that floating-point numbers only approximate the reals, few are cognizant of the dangers this entails for programming. Such dangers range from tolerable rounding errors in sequential programs, to unexpected, divergent control flow in parallel code.**

To address these problems, we present a decision procedure for floating-point arithmetic (FPA) that exploits the proximity to real arithmetic (RA), via a loss-less reduction from FPA to RA. Our procedure does not involve any form of bit-blasting or bit-vectorization, and can thus generate much smaller back-end decision problems, albeit in a more complex logic. This trade-off is beneficial for the exact and reliable analysis of parallel scientific software, which tends to give rise to large but benignly structured formulas. We have implemented a prototype decision engine and present encouraging results analyzing such software for numerical accuracy.

I. Introduction

Floating-point arithmetic (FPA) is the most widely used form of approximating real-valued calculations implemented on computers today. A real number is encoded in the format $(-1)^s \cdot m \cdot 2^e$ with a *sign* bit s, a *mantissa* m and an *exponent* e. The components m and e are stored in fixed-width bitvectors, an (unavoidable) limitation that impacts both the precision and the range of real numbers that can be represented as floating-point numbers. Any real number that does not fit into this format is *rounded*. The precise rules for rounding are formulated in the IEEE 754 Floating-Point Standard, first published in 1985, with a major revision in 2008 [11].[1]

FPA computations are subject to rounding errors, which lead not only to imprecision in the result, but also to potentially surprising side effects of apparently harmless code manipulations, especially in parallel scientific computing applications. Platforms such as OpenMP, CUDA and OpenCL come with compilers that distribute complex computations onto many nodes of a parallel cluster. In the process of doing so, an expression to be evaluated may be *reordered*, in a way that changes its value under floating-point semantics. For instance, if an OpenCL compiler implements the (ambiguous) source code expression $a + b + c + d$ in a reduction tree style as $(a + b) + (c + d)$, rather than sequentially as $((a + b) + c) + d$, the results can differ since floating-point addition is not associative. For long summations, the difference can be significant. Note that the Standard leaves expression evaluation rules to the programming language implementation. Such dependencies on compiler behavior challenge the promise of code portability that platforms like OpenCL make.

To address floating-point inaccuracies, several approaches to formal reasoning with expressions that involve FPA have been developed. Procedures based on arithmetic over real intervals or other abstract domains (e.g., Gappa [5]) determine value ranges for the result of a computation. For efficiency, these procedures typically overapproximate this range. In contrast, more expensive model-exploration based techniques aim at *precise* floating-point reasoning [3], [10]. These techniques encode floating-point expressions into propositional or bit-vector formulas, which are easy to reason about but incur a large increase in formula size, as was clearly demonstrated in [3]. This increase can be attributed to the "logical distance" between the language of the encoding and real arithmetic.

In this paper, we propose an approach that exploits the proximity of floating-point to exact real arithmetic. The Standard essentially mandates that FPA is to be implemented as exact arithmetic followed by rounding. For designers of floating-point hardware, this specification poses a challenge (which is *not* the topic of this paper): given the limited resources on a processor, the FPU cannot first compute the exact arithmetic result. In contrast, a logical reasoning engine for FPA can encode floating-point expressions in two layers: an infinite-precision layer (exact arithmetic) underneath an "adjustment layer" that models the aspects in which FPA and RA differ. The outer layer is primarily responsible for the rounding step.

We show in this paper that such a dual-layer approach is not only feasible, but can lead to a competitive floating-point decision procedure. We present the formal details of the floatingpoint-to-real reduction along with a summary of its implementation in our prototype tool REALIZER.

II. Background: Floating-Point Arithmetic

We review the basics of floating-point arithmetic. We use the symbols $\mathbb{N}, \mathbb{Z}, \mathbb{R}$ to denote the natural, integral and real numbers, respectively. The operators $\lceil \cdot \rceil$ and $\lfloor \cdot \rfloor$ round a real number up and down, respectively, to the next integer. E.g. $\lceil 2.7 \rceil = 3$, $\lfloor -2.3 \rfloor = -3$.

This research is supported by the US National Science Foundation under grant no. CCF-1218075
978-3-9815370-2-4/DATE14/©2014 EDAA

[1] We refer to the 2008 revision simply as *the Standard* in this paper.

A. Floating-Point Numbers

Floating-point numbers are approximations of real numbers suitable for manipulation on a computer. A finite subset of the reals are directly representable as floating-point numbers; other reals are *rounded* to some nearby value. In addition, there are floating-point data that indicate computational results outside the reals, such as positive and negative infinities (e.g. resulting from overflows), and *not-a-number (NaN)* (e.g. resulting from a division $0/0$).

The *binary floating-point format* defined by the 2008 IEEE 754 Floating-Point Standard [11] formalizes floating-point data as triples consisting of a *sign* $s \in \{0, 1\}$, an integer-valued *exponent* e, and a rational-valued *mantissa* m; the triple (s, e, m) represents the real value $(-1)^s \cdot m \cdot 2^e$. The three components are encoded using bit-vectors, of widths 1, r ("range"), and p ("precision"), respectively; common instances are $(r, p) = (8, 23)$ and $(r, p) = (11, 52)$, called *single* and *double* representations. We denote by $\mathrm{FP}_{(r,p)}$ the set containing the real values representable in floating-point format (r, p), and the three symbols $\pm\infty$ and NaN. The real-valued subset $\mathrm{FP}_{(r,p)} \setminus \{\pm\infty, \mathrm{NaN}\}$ equals the set of values $(-1)^s \cdot m \cdot 2^e$ (where e and m fit within the bit-vector dimensions given by r and p) except that some bit patterns in the encoding $(s.e.m)$ are reserved by the Standard for special purposes.

In this paper, we suppress issues involving *subnormal* numbers, and non-real computational results. We focus mostly on the numerical behavior of expression evaluation. The occurrence of infinities and NaN's in computations can often be caught in programs via exception handling and is therefore usually less of interest than problems arising from rounding.

B. Floating-Point Arithmetic

Floating-point arithmetic (FPA) is the logic (parameterized by r, p) of formulas whose atoms are floating-point (in-)equality relations. Such relations are built from expressions whose atoms are floating-point data, or floating-point valued variables. In this paper, we use the fragment of floating-point arithmetic formulas ϕ defined by the grammar

$$\begin{array}{rcl} \phi & ::= & t ⊜ t \mid t ⧀ t \mid \neg\phi \mid \phi \vee \phi \\ t & ::= & c \mid v \mid \ominus t \mid t \oplus t \mid t \ominus t \mid t \otimes t \mid t \oslash t \end{array} \tag{1}$$

where t denotes a floating-point term, c a floating-point literal, and v a floating-point valued variable. A floating-point literal is an element of $\mathrm{FP}_{(r,p)}$. As usual, expressions like $t_1 ⧀ t_2$ and $\phi_1 \wedge \phi_2$ are convenient abbreviations for $\neg(t_1 ⧀ t_2) \wedge \neg(t_1 ⊜ t_2)$ and $\neg(\neg\phi_1 \vee \neg\phi_2)$, respectively.

The Standard requires that an operation be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and that this intermediate result be rounded, if necessary, to fit in the destination's format. This can be formalized as follows (we borrow part of the notation from [3]). For $x \in \mathbb{R}$, define

$$\begin{array}{rcl} \lfloor x \rfloor_{(r,p)} & := & \max\{f \in \mathrm{FP}_{(r,p)} : f \leq x\}, \quad \text{and} \\ \lceil x \rceil_{(r,p)} & := & \min\{f \in \mathrm{FP}_{(r,p)} : f \geq x\}. \end{array} \tag{2}$$

The values $\lfloor x \rfloor_{(r,p)}$ and $\lceil x \rceil_{(r,p)}$ are the two floating-point numbers in $\mathrm{FP}_{(r,p)}$ nearest to x. To *round* x means to map x to one of $\lfloor x \rfloor_{(r,p)}$ and $\lceil x \rceil_{(r,p)}$; which one is determined by the *rounding mode*.[2]

For example, in the "directed" rounding mode *roundTowardPositive*, x is always rounded up to $\lceil x \rceil_{(r,p)}$. For a given rounding mode μ, rounding defines a mathematical function, which we denote by $rd^{\mu}_{(r,p)} \colon \mathbb{R} \to \mathrm{FP}_{(r,p)}$. We write $\mathrm{FPA}_{(r,p,\mu)}$ for floating-point arithmetic (1) over floating-point constants from $\mathrm{FP}_{(r,p)}$ and rounding mode μ. According to the above Standard specification, each binary floating-point operator $\circledcirc \in \{\oplus, \ominus, \otimes, \oslash\}$ in $\mathrm{FPA}_{(r,p,\mu)}$ is defined to return the rounded result of the corresponding real operator $\circ \in \{+, -, \times, /\}$:

$$x \circledcirc y := rd^{\mu}_{(r,p)}(x \circ y)\,. \tag{3}$$

The Standard extends this definition to non-real operands (and results), e.g. $+\infty \oplus -\infty := \mathrm{NaN}$.

Operations that produce Boolean rather than floating-point results, e.g. ⊜ and ⧀, require no rounding. On the subset $\mathrm{FP}_{(r,p)} \setminus \{\pm\infty, \mathrm{NaN}\}$ of real-valued floating-point numbers, they behave like their real counterparts, i.e. we can treat ⊜ as $=$, and ⧀ as $<$. The Standard defines how the non-real floating-point data compare against others, e.g. $\neg(\mathrm{NaN} ⊜ \mathrm{NaN})$, $-\infty ⧀ \infty$.

We observe that, for a given parameter tuple r, p, μ, the set $\mathrm{FP}_{(r,p)}$ of floating-point literals is finite. All operations of $\mathrm{FPA}_{(r,p,\mu)}$ are effectively computable, since both the underlying arithmetic operations and the rounding function $rd^{\mu}_{(r,p)}$ are computable. Satisfiability for $\mathrm{FPA}_{(r,p,\mu)}$ is thus decidable, for instance (in principle) by enumeration.

We also frequently refer to *real arithmetic* (RA) in this paper, which is the standard mathematical theory of the real numbers with classical arithmetic operations. This theory is decidable [8], but complete decision procedures would be too complex to be useful; existing solvers handle fragments. The combination of the reals with integers and classical linear and non-linear arithmetic is known as *mixed real/integer arithmetic* (RIA) and is undecidable [8].

III. A Floating-Point Decision Procedure Based on Exact Arithmetic

The goal of this section is to describe a procedure to translate a formula $f \in \mathrm{FPA}_{(r,p,\mu)}$ into a formula f' in some logic L such that (i) f and f' are equi-satisfiable, and (ii) if satisfiable, an L-valued assignment satisfying f' can be mapped to an assignment of variables over $\mathrm{FP}_{(r,p)}$ that satisfies f. We call formulas f and f' with these two properties *satisfiability-equivalent*. In addition, we want L to be such that efficient solvers exist, and we expect L's background theory to be close to real arithmetic.

A. Reducing Floating-Point to Exact Arithmetic

The *real-valued result* of a binary floating-point operation $\circledcirc$ on real-valued floating-point numbers $x, y \in \mathrm{FP}_{(r,p)}$ can be computed according to the following steps:

[2] In this paper, we use the five IEEE 754-2008 mode names, which are *roundToward*{*Positive*|*Negative*|*Zero*},*roundTiesTo*{*Even*|*Away*}.

1) **calculate** $z := x \circ y$, the real-valued result of the corresponding real-arithmetic operation $\circ$;

2) **normalize** z, i.e. represent z uniquely in the form

$$z = m \cdot 2^e$$

with $1 \le |m| < 2$, and $emin \le e \le emax$ for $emax = 2^{r-1} - 1$ and $emin = 1 - emax$.[3]
Value m is the *signed* mantissa of z, e its exponent ;

3) **round** m to p bits precision (p bits following the radix point), subject to rounding mode rules, obtaining a rounded signed mantissa $\overline{m}$;

4) **return** the real number $\overline{m} \cdot 2^e$.

The pre-rounding normalization step 2 is required to obtain the mantissa m, on which the rounding step depends. In contrast, there is no re-normalization after rounding, although the signed mantissa $\overline{m}$ and exponent e of the final result may not be normalized: the rounding in step 3 may result in $\overline{m} = 2$, e.g. for $p = 4$, $m = 1.111101$ (in binary) and mode *roundTowardPositive*. However, re-normalization does not change the *real-valued result* of the operation, so it is not required in our technique.

Step 2 can be accomplished in real arithmetic as follows. Multiplying $1 \le |m| < 2$ by 2^e we obtain $2^e \le |z| < 2^{e+1}$, so

$$2^e = \max\{2^i \mid i \in \mathbb{Z} \wedge 2^i \le |z|\}. \tag{4}$$

We can now compute the signed mantissa m of step 2 as $z/2^e$. In order to round m to p bits of precision (step 3), we first define five functions — one for each rounding mode — with signature $rd\colon \mathbb{R} \to \mathbb{Z}$, which round their real-valued argument to an *integral* value:

roundTowardPositive	$: rd(x) = \lceil x \rceil$
roundTowardNegative	$: rd(x) = \lfloor x \rfloor$
roundTowardZero	$: rd(x) = sign(x) \cdot \lfloor \lvert x \rvert \rfloor$
roundTiesToEven	$: rd(x) = rd\_tte(x)$
roundTiesToAway	$: rd(x) = sign(x) \cdot \lfloor \lvert x \rvert + 0.5 \rfloor$

where $rd\_tte(x)$ is defined as follows: compute both $\lfloor x + 0.5 \rfloor$ and $\lceil x - 0.5 \rceil$. If these are the same, return that value. Otherwise they differ by exactly 1; return whichever is even.

The rounded signed mantissa $\overline{m}$ can now be computed as

$$\overline{m} = rd(m \cdot 2^p)\,/\,2^p,$$

where rd is the rounding function chosen according to the rounding mode. For example, to round m to p bits of precision in mode *roundTowardPositive*, we shift the radix point in m to the right by p positions, then round up to the next integral value, then shift the radix point back p positions. The general procedure is summarized in Algorithm 1.

According to Algorithm 1, we can express $x \circledcirc y$ explicitly in terms of $x \circ y$ as the following closed formula:

$$x \circledcirc y = \left(\frac{rd\left(\frac{x \circ y}{2^e} \cdot 2^p\right)}{2^p} \right) \cdot 2^e \ . \tag{5}$$

Algorithm 1 Compute $x \circledcirc y$

```
Require: $x, y \in \mathbb{R}$, $p \in \mathbb{N}$
$z := x \circ y$
$exp\_e := \max\{2^i \mid i \in \mathbb{Z} \wedge 2^i \le |z|\}$ ,    // $2^e$ as in eq. (4)
$m := z \,/\, exp\_e$
$\overline{m} := rd(m \cdot 2^p)\,/\,2^p$
return $\overline{m} \cdot exp\_e$
```

Suppose now $f \in \text{FPA}_{(r,p,\mu)}$ is a floating-point formula containing an expression $x \circledcirc y$. We can replace all occurrences of $x \circledcirc y$ in f by the right-hand side term of equation (5) and obtain a satisfiability-equivalent formula f'. A more compact and division-free formulation can be obtained by first replacing f equivalently by $f|_{x \circledcirc y \to v} \wedge v = x \circledcirc y$; the first conjunct here denotes the substitution of a fresh real-valued variable v for each occurrence of $x \circledcirc y$. We can now rewrite $v = x \circledcirc y$ equivalently as $v \cdot 2^{p-e} = rd((x \circ y) \cdot 2^{p-e})$. Putting it all together, our approach is to translate floating-point formula f into the new formula

$$f' \ :: \ f|_{x \circledcirc y \to v} \ \wedge \ v \cdot 2^{p-e} = rd((x \circ y) \cdot 2^{p-e}). \tag{6}$$

This replacement applies to all binary computational operations $\circledcirc$ with their corresponding real counterparts $\circ$. Other (e.g. unary) operations that return a numeric result can be replaced similarly. Repeating this step for all occurrences of floating-point operations results in a formula f' with the following properties:

1) f' is **floating-point free**: after full replacement, the expression in (6) is rid of all occurrences of any $\circledcirc$.

2) f' and f are **satisfiability-equivalent**: this follows from the fact that all floating-point terms have been replaced by equal-valued exact-arithmetic terms. The Boolean structure of f has not been altered, except that f' has additional conjuncts that involve fresh variables v: a satisfying assignment for f can be extended to one for f' using the explicit formula (5), with v in place of $x \circledcirc y$.

B. The Decision Procedure

We now illustrate how to turn the mathematical machinery described above into an effective decision procedure for floating-point arithmetic. Consider the translated formula f' defined by equation (6). This formula is free of occurrences of $x \circledcirc y$. The formula contains parameter p, which is a constant (e.g., $p = 23$ for single-precision floating-point arithmetic). According to this equation, 2^e is the largest power of 2 that is not larger than $|z|$. We can find value of 2^e via a case analysis over the magnitude of z (assumed non-zero). The constraints cover the entire range of z. This requires about $\log rg(z)$ cases, where $rg(z)$ is the size of that range.

The right-hand side of equation (6) contains expressions of the form $x \circ y$, which are real-arithmetic terms. Depending on the rounding mode, (6) may also contain absolute-value and sign operators, which can be defined via simple if-then-else expressions. Finally, the equation may also contain the operators $\lceil \cdot \rceil$ and $\lfloor \cdot \rfloor$, which round a real number up or down to the nearest integer. These operators are typically available

[3] This representation is not possible if $|z|/2^{emin} < 1$, in which case z is *subnormal* or zero. It is also not possible if $|z|/2^{emax} \ge 2$, in which case z *overflows*; the result of normalization then depends on the rounding mode. These cases are easy to catch; we ignore them in this description.

in solvers that support mixed real/integer arithmetic, such as `z3` [6].

To summarize, formula f' resulting from the translation of the given floating-point formula f can be expressed in Boolean logic with real arithmetic as background theory, extended by ceiling and floor functions. That logic is a (small) fragment of mixed real/integer arithmetic (RIA) and is thus amenable to deciding by solvers that support RIA. The size of f' is dominated by the constraints corresponding to (4) that determine 2^e, whose number is logarithmic in the size of the range of $|z|$ and hence linear in the exponent range of the given floating-point format. Considering this number a (possibly large) constant, we have $|f'| = \mathcal{O}(|f|)$. We have implemented the procedure described so far in a tool called REALIZER. This tool takes as input an FPA formula to be analyzed (as defined in (1), but in a LISP-like prefix notation), and a configuration file that specifies the precision parameter, the lower and upper range for each floating-point value in the formula, the decision objective (satisfiability or validity), and the rounding mode. Our way of specifying the range parameter deviates from, but is more flexible than, the way this is formalized in the Standard (e.g. it permits user-specified ranges that go beyond those of *double* or *float*).

REALIZER translates the input formula into a satisfiability-equivalent real-arithmetic formula expressed in the SMT2 language [1] and passes it to the Z3 theorem prover [6]. If satisfiable, the output is an assignment to the input variables of real values that are representable in the chosen floating-point format. REALIZER is available from our project website

```
http://www.ccs.neu.edu/home/wahl/
  Research/fpa-heterogeneous.html
```

The website also contains experimental results obtained using REALIZER on formulas arising from *reduction sums*, a common technique for adding large arrays of floating-point numbers on parallel platforms.

IV. RELATED WORK AND CLOSING REMARKS

Prior decision procedures for floating-point arithmetic are primarily based on propositional or bit-vector encodings of floating-point constraints. "Bit-blasting" approaches are implemented in CBMC [3] and in MATHSAT [4]. With increasing size and complexity of FPA constraints, the resulting propositional encoding becomes very large, which is problematic especially if the input formula itself is large, such as when it represents a lengthy reduction-style computation. An attempt was made to alleviate this problem by applying a combination of under- and over-approximations to the same formula [3].

Decision procedures for FPA have been developed and used in the field of Constraint Satisfaction Problems for generating test vectors [13]. The problem is formalized as a Floating-point Constraint System, and heuristic local consistency algorithms are used to identify parts of the search space which do not contain a solution and interval analysis is then used to verify that no solution exists in that space.

Various formalizations and libraries for FPA have been developed in the domain of theorem proving [12]. Such provers have been used extensively to prove correctness properties of floating-point algorithms for hardware [14], an objective that is completely orthogonal to ours: we analyze software *under the assumption* that the hardware correctly implements IEEE floating-point arithmetic.

ASTREE, a static analysis engine, can soundly abstract floating-point operations using intervals, octagons and ellipsoids [2]. Goubault and Putot [9] present abstract domains and methods to bound the difference between floating-point and real-arithmetic interpretations of the program. These have been incorporated into FLUCTUAT, and can be used for test-case generation. Abstract interpretation and interval arithmetic techniques provide clear efficiency benefits over model exploration approaches such as ours, and feature a high level of automation. They have been successfully applied in industrial contexts. On the other hand, they are approximate and may not suffice when accurate analysis is paramount. Mapping an abstract assignment into the concrete (floating-point) domain can be non-trivial. In contrast, in our approach there is an easy and immediate mapping between floating-point values and values in the model domain (= reals). Recently, a framework for lifting the Conflict Driven Clause Learning (CDCL) approach to abstract domains was proposed [7] and implemented for floating-point numbers [10], using real intervals as abstract domains.

In closing, we mention future work, which includes a customized decision procedure for the fragment of mixed real/integer arithmetic that arises from floating-point encodings. The motivation is that off-the-shelf generic real arithmetic solvers such as Z3, lacking domain knowledge, necessarily embed this fragment into full mixed real/integer arithmetic, which is not even decidable.

REFERENCES

[1] C. Barrett, A. Stump, and C. Tinelli, "The satisfiability modulo theories library (SMT-LIB)," `www.SMT-LIB.org`, 2010.

[2] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, "A static analyzer for large safety-critical software," in *PLDI*, 2003, pp. 196–207.

[3] A. Brillout, D. Kroening, and T. Wahl, "Mixed abstractions for floating-point arithmetic," in *FMCAD*, 2009, pp. 69–76.

[4] A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani, "The MathSAT5 SMT solver."

[5] M. Daumas and G. Melquiond, "Certification of bounds on expressions involving rounded operators," *Transactions on Mathematical Software*, vol. 37, no. 1, pp. 1–20, 2010.

[6] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *TACAS*, 2008, pp. 337–340.

[7] V. D'Silva, L. Haller, and D. Kroening, "Abstract conflict driven learning," in *POPL*, 2013, pp. 143–154.

[8] H. B. Enderton, *A mathematical introduction to logic*. Academic Press, 1972.

[9] E. Goubault and S. Putot, "Static analysis of finite precision computations," in *VMCAI*, 2011, pp. 232–247.

[10] L. Haller, A. Griggio, M. Brain, and D. Kroening, "Deciding floating-point logic with systematic abstraction," in *FMCAD*, 2012, pp. 131–140.

[11] Institute of Electrical and Electronics Engineers (IEEE), "754-2008 — IEEE standard for floating-point arithmetic," *IEEE*, pp. 1–58, 2008.

[12] G. Melquiond, "Floating-point arithmetic in the Coq system," *Inf. Comput.*, vol. 216, pp. 14–23, 2012.

[13] C. Michel, M. Rueher, and Y. Lebbah, "Solving constraints over floating-point numbers," in *CP*, 2001, pp. 524–538.

[14] D. M. Russinoff, "A mechanically checked proof of correctness of the AMD K5 floating point square root microcode," *Formal Methods in System Design*, vol. 14, no. 1, pp. 75–125, 1999.