Invertibility Conditions for Floating-Point Formulas*

Martin Brain[3], Aina Niemetz[1], Mathias Preiner[1], Andrew Reynolds[2], Clark Barrett[1], and Cesare Tinelli[2]

[1] Stanford University
[2] The University of Iowa
[3] University of Oxford and City, University of London

Abstract. Automated reasoning procedures are essential for a number of applications that involve bit-exact floating-point computations. This paper presents conditions that characterize when a variable in a floating-point constraint has a solution, which we call invertibility conditions. We describe a novel workflow that combines human interaction and a syntax-guided synthesis (SyGuS) solver that was used for discovering these conditions. We verify our conditions for several floating-point formats. One implication of this result is that a fragment of floating-point arithmetic admits compact quantifier elimination. We implement our invertibility conditions in a prototype extension of our solver CVC4, showing their usefulness for solving quantified constraints over floating-points.

1 Introduction

Satisfiability Modulo Theories (SMT) formulas including either the theory of floating-point numbers [12] or universal quantifiers [24,32] are widely regarded as some of the hardest to solve. Problems that combine universal quantification over floating-points are rare — experience to date has suggested they are hard for solvers and would-be users should either give up or develop their own incomplete techniques. However, progress in theory solvers for floating-point [11] and the use of expression synthesis for handling universal quantifiers [29,27] suggest that these problems may not be entirely out of reach after all, which could potentially impact a number of interesting applications.

This paper makes substantial progress towards a scalable approach for solving quantified floating-point constraints directly in an SMT solver. Developing procedures for quantified floating-points requires considerable effort, both foundationally and in practice. We focus primarily on establishing a foundation for lifting to quantified floating-point formulas a procedure for solving quantified bit-vector formulas by Niemetz et al. [26]. That procedure relies on so-called *invertibility conditions*, intuitively, formulas that state under which conditions an argument of a given operator and predicate in an equation has a solution. Building on this concept and a state-of-the-art expression synthesis engine [29], we generate invertibility conditions for a majority of operators and predicates in the theory of floating-point numbers. In the context of quantifier-free floating-point formulas, floating-point invertibility conditions may enable us to lift the

* This work was supported in part by DARPA (award no. FA8650-18-2-7861), ONR (award no. N68335-17-C-0558) and NSF (award no. 1656926).

propagation-based local search approach for bit-vectors in [25] to the theory of floating-point numbers.

This work demonstrates that invertibility conditions exist and show promise for solving quantified floating-point constraints. More specifically, it makes the following contributions:

- In Section 3, we present invertibility conditions for the majority of operators and predicates in the SMT-LIB standard theory of floating-point numbers.
- In Section 4, we present a custom methodology based on syntax-guided synthesis and decision tree learning that we developed for the purpose of synthesizing the invertibility conditions presented here.
- In Section 5, we present a quantifier elimination procedure for a fragment of the theory that is based on invertibility conditions, and give experimental evidence of its potential, based on quantified floating-point problems coming from a verification application.

Related Work To our knowledge, no previous work specifically discusses techniques for solving universally quantified floating-point formulas. Brain et al. [11] provide a comprehensive review of decision procedures for quantifier-free bit-exact floating-point using both SMT-based as well as other approaches. They identify four groups of techniques: bit-blasting approaches that use floating-point circuits to generate bit-vector formulas [13,16,20,33], interval techniques that use partitioning and interval propagation [10,31,23,22], optimization and numerical approaches that work with complete valuations [4,18,7,21], and axiomatic techniques that use partial or total axiomatizations of the theory of floating-point numbers in other theories such as real arithmetic [15,14].

On the other hand, approaches for universal quantification have been developed in modern SMT solvers that target other background theories, including linear arithmetic [29,17,8] and bit-vectors [32,27,26]. At a high level, these approaches use model-based refinement loops that lazily add instances of universal quantifiers until they reach a conflict at the quantifier-free level, or otherwise saturate with a model.

2 Preliminaries

We assume the usual notions and terminology of many-sorted first-order logic with equality (denoted by $\approx$). Let Σ be a *signature* consisting of a set Σ^s of sort symbols and a set Σ^f of interpreted (and sorted) function symbols. Each function symbol f has a sort $\tau_1 \times ... \times \tau_n \to \tau$, with arity $n \geq 0$ and $\tau_1, ..., \tau_n, \tau \in \Sigma^s$. We assume that Σ includes a Boolean sort Bool and the Boolean constants $\top$ (true) and $\bot$ (false). We further assume the usual definition of well-sorted terms, literals, and (quantified) formulas with variables and symbols from Σ, and refer to them as Σ-terms, Σ-atoms, and so on. For a Σ-term or Σ-formula e, we denote the *free variables* of e (defined as usual) as $FV(e)$ and use $e[x]$ to denote that the variable x occurs free in e. We write $e[t]$ for the term or formula obtained from e by replacing each occurrence of x in e by t.

A *theory* T is a pair (Σ, I), where Σ is a signature and I is a non-empty class of Σ-interpretations (the *models* of T) that is closed under variable reassignment, i.e., every

Σ-interpretation that only differs from an $\mathcal{I} \in I$ in how it interprets variables is also in I. A Σ-formula φ is *T-satisfiable* (resp. *T-unsatisfiable*) if it is satisfied by some (resp. no) interpretation in I; it is *T-valid* if it is satisfied by all interpretations in I. We will sometimes omit T when the theory is understood from context.

We briefly recap the terminology and notation of Brain et al. [12] which defines an SMT-LIB theory T_{FP} of floating-point numbers based on the IEEE-754 2008 standard [3]. The signature of T_{FP} includes a parametric family of sorts $\mathbb{F}_{\varepsilon,\sigma}$ where ε and σ are integers greater than or equal to 2 giving the number of bits used to store the exponent e and significand s, respectively. Each of these sorts contains five kinds of constants: normal numbers of the form $1.s * 2^e$, subnormal numbers of the form $0.s * 2^{-2^{\sigma-1}-1}$, two zeros ($+0$ and -0), two infinities ($+\infty$ and $-\infty$) and a single not-a-number (NaN). We assume a map $\mathrm{v}_{\varepsilon,\sigma}$ for each sort, which maps these constants to their value in the set $\mathbb{R}^* = \mathbb{R} \cup \{+\infty, -\infty, \mathrm{NaN}\}$. The theory also provides a rounding-mode sort RM, which contains five elements $\{\mathsf{RNE}, \mathsf{RNA}, \mathsf{RTP}, \mathsf{RTN}, \mathsf{RTZ}\}$.

Table 1 lists all considered operators and predicate symbols of theory T_{FP}. The theory contains a full set of arithmetic operations $\{|\ldots|, +, -, \cdot, \div, \sqrt{}, \mathsf{max}, \mathsf{min}\}$ as well as rem (remainder), rti (round to integral) and fma (combined multiply and add with just one rounding). The precise semantics of these operators is given in [12] and follows the same general pattern: $\mathrm{v}_{\varepsilon,\sigma}$ is used to project the arguments to $\mathbb{R}^*$, the normal arithmetic is performed in $\mathbb{R}^*$, then the rounding mode and the result are used to select one of the adjoints of $\mathrm{v}_{\varepsilon,\sigma}$ to convert the result back to $\mathbb{F}_{\varepsilon,\sigma}$. Note that the full theory in [12] includes several additional operators which we omit from discussion here, such as floating-point minimum/maximum, equality with floating-point semantics (fp.eq), and conversions between sorts.

Theory T_{FP} further defines a set of ordering predicates $\{<, >, \leq, \geq\}$ and a set of classification predicates $\{\mathsf{isNorm}, \mathsf{isSub}, \mathsf{isInf}, \mathsf{isZero}, \mathsf{isNaN}, \mathsf{isNeg}, \mathsf{isPos}\}$. In the following, we denote the rounding mode of an operation above the operator symbol, e.g., $a \overset{\mathsf{RTZ}}{+} b$ adds a and b and rounds the result towards zero. We use the infix operator style for isInf ($\ldots \approx \pm\infty$), isZero ($\ldots \approx \pm 0$), and isNaN ($\ldots \approx \mathsf{NaN}$) for conciseness. We further use $\mathrm{min_n}$/$\mathrm{max_n}$ and $\mathrm{min_s}$/$\mathrm{max_s}$ for floating-point constants representing the minimum/maximum normal and subnormal numbers, respectively. We will omit rounding mode and floating-point sorts if they are clear from the context.

3 Invertibility Conditions for Floating-Point Formulas

In this section, we adapt the concept of invertibility conditions introduced by Niemetz et al. in [26] to our theory T_{FP}. Intuitively, an invertibility condition ϕ_c for a literal $l[x]$ is the exact condition under which $l[x]$ has a solution for x, i.e., ϕ_c is equivalent to $\exists x.\, l[x]$ in T_{FP}.

Definition 1. *(Floating-Point Invertibility Condition) Let $l[x]$ be a Σ_{FP}-literal. A quantifier-free Σ_{FP}-formula ϕ_c is an* invertibility condition *for x in $l[x]$ if $x \notin FV(\phi_\mathrm{c})$ and $\phi_\mathrm{c} \Leftrightarrow \exists x.\, l[x]$ is T_{FP}-valid.*

As a simple example of an invertibility condition, given literal $|x| \approx t$ where $|x|$ denotes the absolute value of x, a solution for x exists if and only if t is not negative, i.e., if

Symbol	SMT-LIB Syntax	Sort
isNorm, isSub	fp.isNormal, fp.isSubnormal	$\mathbb{F}_{\varepsilon,\sigma} \to \mathsf{Bool}$
isPos, isNeg	fp.isPositive, fp.isNegative	$\mathbb{F}_{\varepsilon,\sigma} \to \mathsf{Bool}$
isInf, isNaN, isZero	fp.isInfinite, fp.isNaN, fp.isZero	$\mathbb{F}_{\varepsilon,\sigma} \to \mathsf{Bool}$
$\approx, <, >, \leq, \geq$	=, fp.lt, fp.gt, fp.leq, fp.geq	$\mathbb{F}_{\varepsilon,\sigma} \times \mathbb{F}_{\varepsilon,\sigma} \to \mathsf{Bool}$
$\lvert\ldots\rvert, -$	fp.abs, fp.neg	$\mathbb{F}_{\varepsilon,\sigma} \to \mathbb{F}_{\varepsilon,\sigma}$
rem	fp.rem	$\mathbb{F}_{\varepsilon,\sigma} \times \mathbb{F}_{\varepsilon,\sigma} \to \mathbb{F}_{\varepsilon,\sigma}$
$\surd$, rti	fp.sqrt, fp.roundToIntegral	$\mathsf{RM} \times \mathbb{F}_{\varepsilon,\sigma} \to \mathbb{F}_{\varepsilon,\sigma}$
$+, -, \cdot, \div$	fp.add, fp.sub, fp.mul, fp.div	$\mathsf{RM} \times \mathbb{F}_{\varepsilon,\sigma} \times \mathbb{F}_{\varepsilon,\sigma} \to \mathbb{F}_{\varepsilon,\sigma}$
fma	fp.fma	$\mathsf{RM} \times \mathbb{F}_{\varepsilon,\sigma} \times \mathbb{F}_{\varepsilon,\sigma} \times \mathbb{F}_{\varepsilon,\sigma} \to \mathbb{F}_{\varepsilon,\sigma}$

Table 1: Considered floating-point predicates/operators, with SMT-LIB 2 syntax.

$\neg\mathsf{isNeg}(t)$ holds. We introduce additional terminology for the sake of the discussion. We define the *dimension* of an invertibility condition problem $\exists x.\, l[x]$ as the number of free variables it contains. For example, if s and t are variables, then the dimension of $\exists x.\, x + s \approx t$ is two, the dimension of $\exists x.\, \mathsf{isZero}(x + s)$ is one, and the dimension of $\exists x.\, \mathsf{isZero}(|x|)$ is zero. A literal $l[x]$ is *fully invertible* if its invertibility condition is $\top$. A term e is an (unconditional) *inverse* for x in $l[x]$ if $l[e]$ is equivalent to $\top$. For example, the literal $-x \approx t$ is fully invertible and $-t$ is an inverse for x in this literal. We say that e is a *conditional inverse* for $l[x]$ if $l[e]$ is an invertibility condition for $l[x]$.

Our primary goal in this work is to establish invertibility conditions for all floating-point constraints that contain exactly one operator and one predicate. These conditions collectively suffice to characterize when any literal $l[x]$ containing exactly one occurrence of x, the variable to solve for, has a solution. In total, we were able to establish 167 out of 188 invertibility conditions (counting commutative cases only once) using a syntax-guided synthesis framework which we describe in more detail in Section 4. In this section, we present a subset of these invertibility conditions, highlighting the most interesting cases where we succeeded (or failed) to establish an invertibility condition. Due to space restrictions, we omit the conditions for the remaining cases.[4]

Table 2 lists the invertibility conditions for equality with the operators $\{+, -, \cdot, \div, \mathsf{rem}, \surd, |\ldots|, -, \mathsf{rti}\}$, parameterized over a rounding mode R (one of RNE, RNA, RTP, RTN, or RTZ). Note that operators $\{+, \cdot\}$ and the multiplicative step of fma are commutative, and thus the invertibility conditions for both variants are identical.

Each of the first six invertibility conditions in this table follows a pattern. The first two disjuncts are instances of the literal to solve for, where a term involving rounding modes RTP and RTN is substituted for x. These disjuncts are then followed by disjuncts for handling special cases for infinity and zero. From the structure of these conditions, e.g., for $+$, we can derive the insight that if there is a solution for x in the equation $x \overset{\mathsf{R}}{+} s \approx t$ and we are not in a corner case where $s = t$, then either $t \overset{RTP}{-} s$ or $t \overset{RTN}{-} s$ must be a solution. Based on extensive runs of our syntax-guided synthesis procedure, we believe this condition is close to having minimal term size. From this, we

[4] Available at https://cvc4.cs.stanford.edu/papers/CAV2019-FP .

Literal	Invertibility Condition
$x \overset{\mathsf{R}}{+} s \approx t$	$t \approx (t \overset{\mathsf{RTP}}{-} s) \overset{R}{+} s \vee t \approx (t \overset{\mathsf{RTN}}{-} s) \overset{R}{+} s \vee s \approx t$
$x \overset{\mathsf{R}}{-} s \approx t$	$t \approx (s \overset{\mathsf{RTP}}{+} t) \overset{R}{-} s \vee t \approx (s \overset{\mathsf{RTN}}{+} t) \overset{R}{-} s \vee (s \not\approx t \wedge s \approx \pm\infty \wedge t \approx \pm\infty)$
$s \overset{\mathsf{R}}{-} x \approx t$	$t \approx s \overset{R}{+} (t \overset{\mathsf{RTP}}{-} s) \vee t \approx s \overset{R}{+} (t \overset{\mathsf{RTN}}{-} s) \vee s \approx t$
$x \overset{\mathsf{R}}{\cdot} s \approx t$	$t \approx (t \overset{\mathsf{RTP}}{\div} s) \overset{R}{\cdot} s \vee t \approx (t \overset{\mathsf{RTN}}{\div} s) \overset{R}{\cdot} s \vee (s \approx \pm\infty \wedge t \approx \pm\infty) \vee (s \approx \pm 0 \wedge t \approx \pm 0)$
$x \overset{\mathsf{R}}{\div} s \approx t$	$t \approx (s \overset{\mathsf{RTP}}{\cdot} t) \overset{R}{\div} s \vee t \approx (s \overset{\mathsf{RTN}}{\cdot} t) \overset{R}{\div} s \vee (s \approx \pm\infty \wedge t \approx \pm 0) \vee (t \approx \pm\infty \wedge s \approx \pm 0)$
$s \overset{\mathsf{R}}{\div} x \approx t$	$t \approx s \overset{R}{\div} (s \overset{\mathsf{RTP}}{\div} t) \vee t \approx s \overset{R}{\div} (s \overset{\mathsf{RTN}}{\div} t) \vee (s \approx \pm\infty \wedge t \approx \pm\infty) \vee (s \approx \pm 0 \wedge t \approx \pm 0)$
$x \,\mathsf{rem}\, s \approx t$	$t \approx t \,\mathsf{rem}\, s$
$s \,\mathsf{rem}\, x \approx t$	?
$\overset{\mathsf{R}}{\sqrt[n]{x}} \approx t$	$t \approx \overset{R}{\sqrt[n]{(t \overset{\mathsf{RTP}}{\cdot} t)}} \vee t \approx \overset{R}{\sqrt[n]{(t \overset{\mathsf{RTN}}{\cdot} t)}} \vee t \approx \pm 0$
$\lvert x \rvert \approx t$	$\neg \mathsf{isNeg}(t)$
$-x \approx t$	$\top$
$\overset{\mathsf{R}}{\mathsf{rti}}(x) \approx t$	$t \approx \overset{R}{\mathsf{rti}}(t)$

Table 2: Invertibility conditions for floating-point operators (excl. fma) with $\approx$.

conclude that an efficient yet complete method for solving $x \overset{\mathsf{R}}{+} s \approx t$ checks whether $t - s$ rounding towards positive or negative is a solution in the non-trivial case when s and t are disequal, and otherwise concludes that no solution exists. A similar insight can be derived for the other invertibility conditions of this form.

We found that t is a conditional inverse for the case of $\overset{\mathsf{R}}{\mathsf{rti}}(x) \approx t$ and $x \,\mathsf{rem}\, s \approx t$, that is, substituting t for x is an invertibility condition. For the latter, we discovered an alternative invertibility condition:

$$\lvert t \overset{\mathsf{RTP}}{+} t \rvert \leq \lvert s \rvert \vee \lvert t \overset{\mathsf{RTN}}{+} t \rvert \leq \lvert s \rvert \vee \mathsf{ite}(t \approx \pm 0, s \not\approx \pm 0, t \not\approx \pm\infty) \tag{1}$$

In contrast to the condition from Table 2, this version does not involve rem. It follows that certain applications of floating-point remainder, including those whose first argument is an unconstrained variable, can be eliminated based on this equivalence. Interestingly, for $s \,\mathsf{rem}\, x \approx t$, we did not succeed in finding an invertibility condition. This case appears to not admit a concise solution; we discuss further details below.

Table 3 gives the invertibility conditions for $\geq$. Since these constraints admit more solutions, they typically have simpler invertibility conditions. In particular, with the exception of rem, all conditions only involve floating-point classifiers.

When considering literals with predicates, the invertibility conditions for cases involving $x + s$ and $s - x$ are identical for every predicate and rounding mode. This is due to the fact that $s - x$ is equivalent to $s + (-x)$, independent from the rounding mode. Thus, the negation of the inverse value of x for an equation involving $x + s$ is the inverse value of x for an equation involving $s - x$. Similarly, the invertibility conditions for $x \cdot s$ and $s \div x$ over predicates $\{<, \leq, >, \geq, \mathsf{isInf}, \mathsf{isNaN}, \mathsf{isNeg}, \mathsf{isZero}\}$ are identical for all rounding modes.

For all predicates except $\{\approx, \mathsf{isNorm}, \mathsf{isSub}\}$, the invertibility conditions for operators $\{+, -, \div, \cdot\}$ contain floating-point classifiers only. All of these conditions are

Literal	Invertibility Condition
$x \overset{\mathsf{R}}{+} s \geq t$	$(\mathsf{isPos}(s) \vee \mathsf{ite}(s \approx \pm\infty, (t \approx \pm\infty \wedge \mathsf{isNeg}(t)), \mathsf{isNeg}(s))) \wedge t \not\approx \mathsf{NaN}$
$x \overset{\mathsf{R}}{-} s \geq t$	$\mathsf{ite}(\mathsf{isNeg}(s), t \not\approx \mathsf{NaN}, \mathsf{ite}(s \approx \pm\infty, (t \approx \pm\infty \wedge \mathsf{isNeg}(t)), (\mathsf{isPos}(s) \wedge t \not\approx \mathsf{NaN})))$
$s \overset{\mathsf{R}}{-} x \geq t$	$(\mathsf{isPos}(s) \vee \mathsf{ite}(s \approx \pm\infty, (t \approx \pm\infty \wedge \mathsf{isNeg}(t)), \mathsf{isNeg}(s))) \wedge t \not\approx \mathsf{NaN}$
$x \overset{\mathsf{R}}{\cdot} s \geq t$	$(\mathsf{isNeg}(t) \vee t \approx \pm 0 \vee s \not\approx \pm 0) \wedge s \not\approx \mathsf{NaN} \wedge t \not\approx \mathsf{NaN}$
$x \overset{\mathsf{R}}{\div} s \geq t$	$(\mathsf{isNeg}(t) \vee t \approx \pm 0 \vee s \not\approx \pm\infty) \wedge s \not\approx \mathsf{NaN} \wedge t \not\approx \mathsf{NaN}$
$s \overset{\mathsf{R}}{\div} x \geq t$	$(\mathsf{isNeg}(t) \vee t \approx \pm 0 \vee s \not\approx \pm 0) \wedge s \not\approx \mathsf{NaN} \wedge t \not\approx \mathsf{NaN}$
$x \,\mathsf{rem}\, s \geq t$	$\mathsf{ite}(\mathsf{isNeg}(t), s \not\approx \mathsf{NaN}, (\lvert t \overset{\mathsf{RNE}}{+} t \rvert \leq \lvert s \rvert \wedge t \not\approx \pm\infty)) \wedge s \not\approx \pm 0$
$s \,\mathsf{rem}\, x \geq t$	?
$\overset{\mathsf{R}}{\sqrt{x}} \geq t$	$t \not\approx \mathsf{NaN}$
$\lvert x \rvert \geq t$	$t \not\approx \mathsf{NaN}$
$-x \geq t$	$t \not\approx \mathsf{NaN}$
$\overset{\mathsf{R}}{\mathsf{rti}}(x) \geq t$	$t \not\approx \mathsf{NaN}$

Table 3: Invertibility conditions for floating-point operators (excl. fma) with $\geq$.

also independent from the rounding mode. Similarly, for operator fma over predicates $\{\mathsf{isInf}, \mathsf{isNaN}, \mathsf{isNeg}, \mathsf{isPos}\}$, the invertibility conditions contain only floating-point classifiers. All of these conditions except for $\mathsf{isNeg}(\mathsf{fma}(x, s, t))$ and $\mathsf{isPos}(\mathsf{fma}(x, s, t))$ are also independent from the rounding mode.

For all floating-point operators with predicate isNaN, the invertibility condition is $\top$, i.e., an inverse value for x always exists. This is due to the fact that every floating-point operator returns NaN if one of its operands is NaN, hence NaN can be picked as an inverse value of x. Conversely, we identified four cases for which the invertibility condition is $\bot$, i.e., an inverse value for x never exists. These four cases are $\mathsf{isNeg}(|x|)$, $\mathsf{isInf}(x \,\mathsf{rem}\, s)$, $\mathsf{isInf}(s \,\mathsf{rem}\, x)$, and $\mathsf{isSub}(\mathsf{rti}(x))$. For the first three cases, it is obvious why no inverse value exists. The intuition for $\mathsf{isSub}(\mathsf{rti}(x))$ is that integers are not subnormal, and as a result if x is rounded to an integer it can never be a subnormal number. All of these cases can be easily implemented as rewrite rules in an SMT solver.

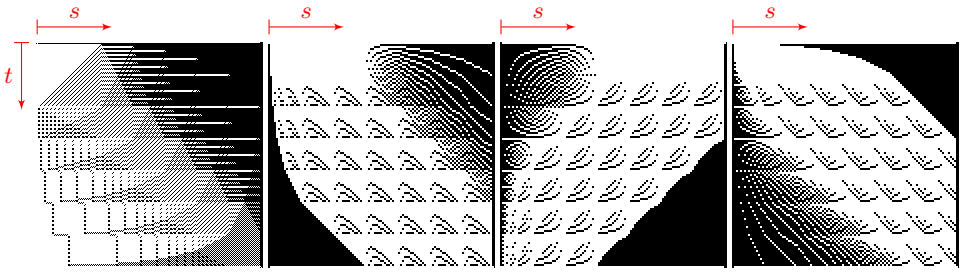
For operator fma, the invertibility conditions over predicates $\{\mathsf{isInf}, \mathsf{isNaN}, \mathsf{isNeg}, \mathsf{isPos}\}$ contain floating-point classifiers only. For predicate isZero, the invertibility conditions are more involved. Equations (2) and (3) show the invertibility conditions for $\mathsf{isZero}(\mathsf{fma}(x, s, t))$ and $\mathsf{isZero}(\mathsf{fma}(s, t, x))$ for all rounding modes R.

$$\overset{R}{\mathsf{fma}}(-(t \overset{\mathsf{RTP}}{\div} s), s, t) \approx \pm 0 \vee \overset{R}{\mathsf{fma}}(-(t \overset{\mathsf{RTN}}{\div} s), s, t) \approx \pm 0 \vee (s \approx \pm 0 \wedge t \approx \pm 0) \tag{2}$$

$$\overset{R}{\mathsf{fma}}(s, t, -(s \overset{\mathsf{RTP}}{\cdot} t)) \approx \pm 0 \vee \overset{R}{\mathsf{fma}}(s, t, -(s \overset{\mathsf{RTN}}{\cdot} t)) \approx \pm 0 \tag{3}$$

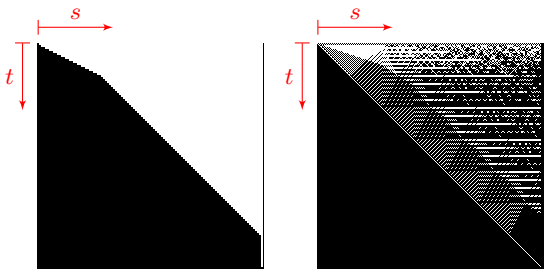
These two invertibility conditions contain case splits similar to those in Table 2 and indicate that, e.g., $-t \overset{\mathsf{RTP}}{\div} s$ is an inverse value for x when $\overset{\mathsf{R}}{\mathsf{fma}}(-(t \overset{\mathsf{RTP}}{\div} s), s, t) \approx \pm 0$ holds.

As we will describe in Section 4, an important aspect of synthesizing these invertibility conditions was considering their visualizations. This helped us determine which invertibility conditions were relatively simple and which exhibited complex behavior.



(a) $x + s \approx t$ (b) $x \cdot s \approx t$ (c) $x \div s \approx t$ (d) $s \div x \approx t$

Fig. 1: Invertibility conditions for $\{+, \cdot, \div\}$ over $\approx$ for $\mathbb{F}_{3,5}$ and rounding mode RNE.



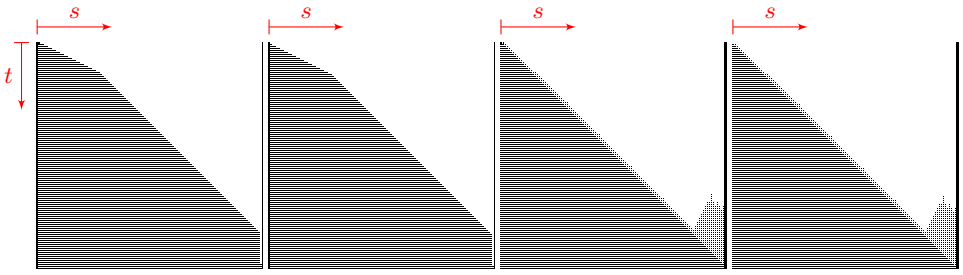
(a) x rem $s \approx t$ (b) s rem $x \approx t$

Fig. 2: Invertibility conditions for rem over $\approx$ for $\mathbb{F}_{3,5}$.

Figure 1 shows the visualizations of the invertibility conditions for operators $\{+, \cdot, \div\}$ over $\approx$ from Table 2 for sort $\mathbb{F}_{3,5}$ with rounding mode RNE (each of the literals is two-dimensional). We use 227×227 pixel maps over all possible values of s and t, where the pixel at point (s, t) is white if the invertibility condition is true, and black if it is false.[5] The values of s are plotted on the horizontal axis and the values of t are plotted on the vertical axis. The leftmost two columns (resp. topmost two rows) give the value of the invertibility condition for $s = \pm 0$ (resp. $t = \pm 0$); the rightmost column (resp. bottom row) gives its value for NaN; the next two columns left of (resp. next two rows on top of) NaN give its value for $\pm\infty$; the remainder plots the values of the subnormal and normal values of s and t, left-to-right (resp. top-to-bottom) in increasing order of their absolute value, alternating between positive and negative values. These visualizations give an intuition of the complexity of the behavior of invertibility conditions, which is a consequence of the complex semantics of floating-point operations.

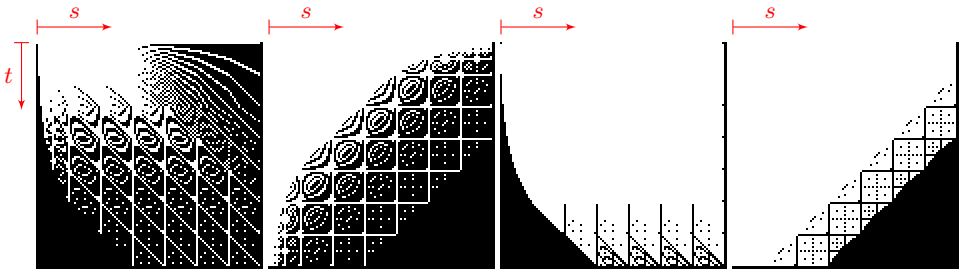
Figure 2 gives the invertibility condition visualizations for remainder over $\approx$ with sort $\mathbb{F}_{3,5}$ and rounding mode RNE. The visualization on the left hand shows that solving for x as the first argument is relatively easy. It suggests that an invertibility condition for this case involves a linear inequality relating the absolute values of s and t, which we were able to derive in Equation (1). Solving for x as the second argument, on the other hand, is much more difficult, as indicated by the right picture, which has a significantly more complex structure. We conjecture that no simple solution exists for the latter prob-

[5] Notice that we consider all possible $(2^{\sigma-1} - 1) * 2$ NaN values of T_{FP} as one single NaN value. Thus, for sort $\mathbb{F}_{3,5}$ we have 227 floating-point values (instead of $2^8 = 256$).



(a) $x \,\mathsf{rem}\, s > t$ (b) $x \,\mathsf{rem}\, s \geq t$ (c) $s \,\mathsf{rem}\, x > t$ (d) $s \,\mathsf{rem}\, x \geq t$

Fig. 3: Invertibility conditions for rem over inequalities for $\mathbb{F}_{3,5}$.



(a) $\mathsf{fma}(x,s,t) \approx \pm 0$ (b) $\mathsf{fma}(s,t,x) \approx \pm 0$ (c) $\mathsf{isSub}(\mathsf{fma}(x,s,t))$ (d) $\mathsf{isSub}(\mathsf{fma}(s,t,x))$

Fig. 4: Invertibility conditions for fma over $\{\mathsf{isZero}, \mathsf{isSub}\}$ for $\mathbb{F}_{3,5}$ and rnd. mode RNE.

lem. The visualization of the invertibility condition gives some of the intuition for this: the diagonal divide is caused by the fact that output t will always have a smaller absolute value than the input s. The top-left corner represents subnormal/subnormal computation, this acts as fixed-point and behaves differently from the rest of the function. The stepped blocks along the diagonal occur when s and t have the same exponent and thus the pattern is similar to the invertibility condition for $+$ shown in Figure 1. Portions right of the main diagonal appear to exhibit random behavior. We believe this is the result of repeated cancellations in the computation of the remainder for those values, which suggests a behavior that we believe is similar to the Blum-Blum-Shub random number generator [9].

For remainder with inequalities, we succeeded in determining invertibility conditions for $\leq$ and $\geq$ if x is the first argument. However, for $x \,\mathsf{rem}\, s$ over $\{<, >\}$, and $s \,\mathsf{rem}\, x$ over $\{\geq, \leq, <, >\}$ we did not. This is particularly surprising considering that the invertibility conditions for non-strict and strict inequalities are nearly identical (varying only by a handful of pixels), as shown in Figure 3. Note that for x as the first argument, all variations of the concise invertibility conditions for non-strict inequality we considered failed as solutions for the strict inequality. This behavior is representative of the many subtle corner cases we encountered while synthesizing these conditions.

Figure 4 shows visualizations for invertibility conditions involving fma. The left two images are visualizations for the invertibility conditions for isZero. The corresponding invertibility conditions are given in Equations (2) and (3) above. We were not able to determine invertibility conditions for operator fma over predicate isSub, which are visualized in the rightmost two pictures in Figure 4. Finally, we did not succeed in finding

invertibility conditions for fma with binary predicates, which are particularly challenging since they are three-dimensional. Finding solutions for these cases is ongoing work (see Section 4 for a more in-depth discussion).

4 Synthesis of Floating-Point Invertibility Conditions

Deriving invertibility conditions in T_{FP} is a highly challenging task. We were unable to derive these conditions manually despite our substantial background knowledge of floating-point numbers. As a consequence, we developed a custom extension of the syntax-guided synthesis (SyGuS) paradigm [1] with the goal of finding invertibility conditions automatically, which resulted in the conditions from Section 3. While the extension was optimized for this task, we stress that our techniques are theory-agnostic and can be used for synthesis problems over any finite domain. Our approach builds upon the SyGuS capabilities of the SMT solver CVC4 [29,5], which has recently been extended to support reasoning about the theory of floating-points [11]. We use the invertibility condition for floating-point addition with equality here as a running example.

Establishing an invertibility condition requires solving a synthesis problem with *three* levels of quantifier alternation. In particular, for floating-point addition with equality, we are interested in finding a solution for predicate IC that satisfies the conjecture:

$$\exists\, \mathsf{IC}.\, \forall s, t.\, (\mathsf{IC}(s,t) \Leftrightarrow (\exists x.\, x \overset{\mathsf{R}}{+} s \approx t)) \tag{4}$$

for some rounding mode R. In other words, this conjecture states that $\mathsf{IC}(s,t)$ holds exactly when there exists an x that, when rounding the result of adding x to s according to mode R, yields t. Furthermore, we are interested in finding a solution for IC that holds *independently of the format* of x, s, t. Note that SMT solvers are not capable of reasoning about constraints that are parametric in the floating-point format. To address this challenge, following the methodology from previous work [26], our strategy for establishing (general) invertibility conditions first solves the synthesis conjecture for a fixed format $\mathbb{F}_{\varepsilon,\sigma}$, and subsequently checks whether that solution also holds for other formats. The choice of the number of exponent bits ε and significand bits σ in $\mathbb{F}_{\varepsilon,\sigma}$ balances two criteria:

1. ε, σ should be large enough to exercise many (or all) of the behaviors of the operators and relations in our synthesis conjecture,
2. ε, σ should be small enough for the synthesis problem to be tractable.

In our experience, the best choices for (ε, σ) depended on the particular invertibility condition we were solving. The most common choices for (ε, σ) were $(3,5)$, $(4,5)$ and $(4,6)$. For most two-dimensional invertibility conditions (those that involve two variables s and t), we used $(3,5)$, since the required synthesis procedures mentioned below were roughly eight times faster than for $(4,5)$. For one-dimensional invertibility conditions, we often used higher precision formats. Since floating-point operators like addition take as additional argument a rounding mode R, we assumed a fixed rounding mode when solving, and then cross-checked our solution for multiple rounding modes.

Assume we have chosen to synthesize the invertibility condition for conjecture (4) for format $\mathbb{F}_{3,5}$ and rounding mode RNE. Notice that current SyGuS solvers [29,2]

support only two levels of quantifier alternation. However, we can expand the innermost quantifier in this conjecture to obtain the conjecture:

$$\exists \mathsf{IC}.\, \forall st.\, (\mathsf{IC}(s,t) \Leftrightarrow (\bigvee_{i=0}^{226} i \overset{\mathsf{RNE}}{+} s \approx t)) \tag{5}$$

where for simplicity of notation we use $i = 0, \ldots, 226$ to denote the values of $\mathbb{F}_{3,5}$. This methodology was also used in Niemetz et al. [26], where invertibility conditions for bit-vector operators were synthesized for bit-width 4 by giving the conjecture of the above form to an off-the-shelf SyGuS solver. In contrast to that work, we found that the synthesis conjecture above is too challenging to be solved efficiently by current state-of-the-art enumerative SyGuS solvers. The reason for this is twofold. First, the smallest viable floating-point format is $3 + 5 = 8$ bits, which requires the body of (5) to have a significantly large number of disjuncts (227), which is more than ten times larger than the 16 disjuncts required when synthesizing 4-bit invertibility conditions for bit-vectors. Second, floating-point formulas are much harder to solve than bit-vector formulas, due to the complexity of their bit-blasted encodings. Thus, a significantly challenging satisfiability query must be solved *for each* candidate considered within the SyGuS solver.

To address the above challenges, we perform a more extreme preprocessing step on our synthesis conjecture, which computes the input/output behavior of the invertibility condition on all points in the domain of s and t. In other words, we rephrase our synthesis conjecture as:

$$\exists \mathsf{IC}.\, \bigwedge_{i=0}^{226} \bigwedge_{j=0}^{226} (\mathsf{IC}(i,j) \Leftrightarrow c_{i,j}) \tag{6}$$

where each $c_{i,j}$ is a Boolean constant (either $\top$ or $\bot$) determined by a quantifier-free satisfiability query. In particular, for each pair of floating-point values (i, j), constant $c_{i,j}$ is $\top$ if $x+i \approx j$ is satisfiable, and $\bot$ if it is unsatisfiable. In practice, we represent the above conjecture as a 227×227 table, which we call the *full I/O specification* of invertibility condition IC. In our experiments, computing this table for most two-dimensional invertibility conditions of sort $\mathbb{F}_{3,5}$ required 15 minutes (for $227 * 227 = 51,529$ quantifier-free queries), and 2 hours for sort $\mathbb{F}_{4,5}$ (requiring $483 * 483 = 233,289$ queries). This process was accelerated by first applying random sampling over possible values of x to quickly test if a query was satisfiable. For some operators, notably remainder, this required significantly more time than for others (up to a factor of 2). Due to the high cost of this preprocessing step, we generated a database with the full I/O specifications for *all* invertibility conditions from Section 3 using a cluster of 50 nodes with Intel Xeon E5-2637 with 3.5GHz and 32GB memory, and then shared this database among multiple developers. Computing the full I/O specifications for $\mathbb{F}_{3,5}$, $\mathbb{F}_{4,5}$, and $\mathbb{F}_{4,6}$ required a total of 459 days of CPU time (6.1 for $\mathbb{F}_{3,5}$, 54.7 for $\mathbb{F}_{4,5}$, and 398.5 for $\mathbb{F}_{4,6}$). Despite the heavy cost of this step, it was crucial for accelerating our framework for synthesizing invertibility conditions, described next.

Figure 5 summarizes our architecture for solving synthesis conjectures of the above form. The user first selects an invertibility condition problem to solve, where we assume the full I/O specification has been computed using the aforementioned techniques. At a

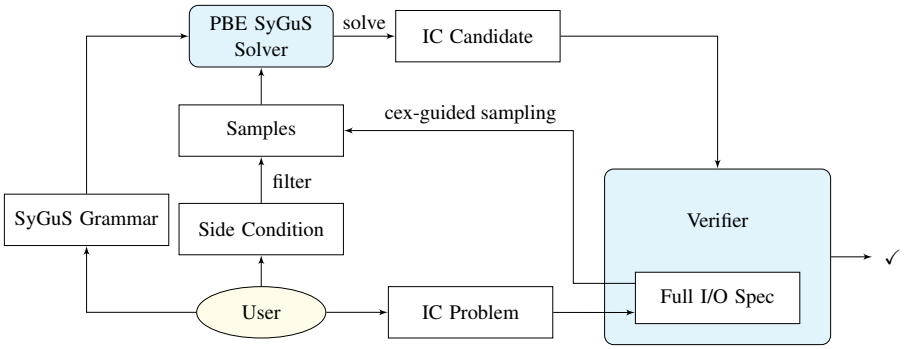


Fig. 5: Architecture for synthesizing invertibility conditions for floating point formulas.

high level, our architecture can be seen as an *interactive synthesis environment*, where the user manages the interaction between two subprocedures:

1. a SyGuS solver with support for decision tree learning, and
2. a solution verifier storing the full I/O specification of the invertibility condition.

We use a counterexample-guided loop, where the SyGuS solver provides the solution verifier with candidate solutions, and the solution verifier provides the SyGuS solver with an evolving subset of sample points taken from the full I/O specification. These points correspond to counterexamples to failed candidate solutions, and are sampled in a uniformly random manner over the domain of our specification. To accelerate the speed at which our framework converges on a solution, we configure the solution verifier to generate multiple counterexample points (typically 10) for each iteration of the loop. The process terminates when the SyGuS solver generates a candidate solution that is correct for all points according to its full I/O specification.

We give the user control over both the solutions and counterexample points generated in this loop. First, as is commonly done in syntax-guided synthesis applications, the user in our workflow provides an input grammar to the SyGuS solver. This is a context-free grammar in a standard format [28], which contains a guess of the operators and patterns that may be involved in the invertibility condition we are synthesizing. Second, note that the domain of floating-point numbers can be subdivided into a number of subdomains and special cases (e.g. normal, subnormal, not-a-number, infinity), as well as split into different classifications (e.g. positive and negative). Our workflow allows the user to provide a *side condition*, whose purpose is to focus on finding an invertibility condition that is correct for one of these subdomains. The side condition acts as a filtering mechanism on the counterexample points generated by the solution verifier. For example, given the side condition $\mathsf{isNorm}(s) \wedge \mathsf{isNorm}(t)$, the solution verifier checks candidate solutions generated by the SyGuS solver only against points (s, t) where both arguments are normal, and consequently only communicates counterexamples of this form to the SyGuS solver. The solution verifier may also be configured to establish that the current candidate solution generated by the SyGuS solver is *conditionally* correct, that is, it is true on all points in the domain that satisfy the side condition.

There are several advantages to the form of the synthesis conjecture in (6) that we exploit in our workflow. First, its structure makes it easy to divide the problem into sub-

cases: our synthesis workflow at all times sends only a subset of the conjuncts of (6) for some (i, j) pairs. As a result, we do not burden the underlying SyGuS solver with the entire conjecture at once, which would not scale in practice. A second advantage is that it is in *programming-by-examples* (PBE) form, since it consists of a conjunction of concrete input-output pairs. As a consequence, specialized algorithms can be used by the SyGuS solver to generate solutions for (approximations of) our conjecture in a way that is highly scalable in practice. These techniques are broadly referred to as decision tree learning or unification algorithms. As a brief review (see Alur et al. [2] for a recent SyGuS-based approach), a decision tree learning algorithm is given as input a set of good examples $c_1 \mapsto \top, \ldots, c_n \mapsto \top$ and a set of bad examples $d_1 \mapsto \bot, \ldots, d_m \mapsto \bot$. The goal of a decision tree algorithm is to find a predicate, or *classifier*, that evaluates to true on all the good examples, and false on all the bad examples. In our context, a classifier is expressed as an if-then-else tree of Boolean sort. Sampling the space of conjecture (6) provides the decision tree algorithm with good and bad examples and the returned classifier is a candidate solution that we give to the solution verifier. The SyGuS solver of CVC4 uses a decision-tree learning algorithm, which we rely on in our workflow. Due to the scalability of this algorithm and the fact that only a small subset of our conjecture is considered at any given time, candidate solutions are typically generated by the SyGuS solver in our framework in a matter of seconds.

Another important aspect of the SyGuS solver in Figure 5 is that it is configured to generate *multiple* solutions for the current set of sample points. Due to the way the SyGuS-based decision-tree learning algorithm works, these solutions tend to become *more general* over the runtime of the solver. As a simple example (assuming exact integer arithmetic), say the solver is given input points $(1, 1) \mapsto \top$, $(2, 0) \mapsto \top$, $(1, 0) \mapsto \bot$ and $(0, 1) \mapsto \bot$ for (s, t). It enumerates predicates over s and t, starting with simplest predicates first, say $s \approx 0$, $t \approx 0$, $s \approx 1$, $y \approx 1$, $s + t > 1$, and so on. After generating the first four predicates, it constructs the solution $\mathsf{ite}(s \approx 1, t \approx 1, t \approx 0)$, which is a correct classifier for the given set of points. However, after generating the fifth predicate in this list, it returns $s + t > 1$ itself as a solution; this can be seen as a generalization of the previous solution since it requires no case splitting.

Since more general candidate solutions have a higher likelihood of being actual solutions in our experience, our workflow critically relies on the ability of users to manually terminate the synthesis procedure when they are satisfied with the last generated candidate. Our synthesis procedure logs a list of candidate solutions that satisfy the conjecture on the current set of sample points. When the user terminates the synthesis process, the solution verifier will check the last solution generated in this list. Users have the option to rearrange the elements of this list by hand, if they have an intuition that a specific candidate is more likely to be correct — and so should be tested first.

Experience The first challenging invertibility condition we solved with our framework was addition with equality for rounding mode RNE. Initially, we used a generic grammar that contained the entire floating-point signature. As a first key step towards solving this problem, the synthesis procedure suggested the single literal $t \approx s \overset{\mathsf{RNE}}{+} (t \overset{\mathsf{RNE}}{-} s)$ as candidate solution. Although counterexamples were found for this candidate, we noticed that it satisfied over 98% of the specification, and a visualization of its I/O behavior

showed similar patterns to the invertibility condition we were solving for. Based on these observations, we focused our grammar towards literals of this form. In particular, we used a function that takes two floating-points x, y and two rounding modes R_1, R_2 as arguments and returns $x \overset{R_1}{+} (y \overset{R_2}{-} x)$ as a builtin symbol of our grammar. We refer to such a function as a *residual* computation of y, noting that its value is often approximately y. By including various functions for residual computations, we focused the effort of the synthesizer on more interesting predicates. The end solution involved multiple residual computations, as shown in Table 2. Our initial solution was specific to the rounding mode RNE. After solving for several other rounding modes, we were able to construct a parametric solution that was correct for all rounding modes. In total, it took roughly three days of developer time to discover the generalized invertibility condition for addition with equality. Many of the subsequent invertibility conditions took a matter of hours, since by then we had a good intuition for the residual computations that were relevant for each case.

Invertibility conditions involving rem, fma, isNorm, and isSub were challenging and required further customizations to the grammar, for instance to include constants that corresponded to the minimum and maximum normal and subnormal values. Three-dimensional invertibility conditions (which in this work is limited to cases of fma with binary predicates) were especially challenging since the domain of their conjecture is a factor of 227 larger for $\mathbb{F}_{3,5}$ than the others. Following our strategy for solving the invertibility conditions for specific formats and rounding modes, in ongoing work we are investigating solving these cases by first solving the invertibility condition for a fixed value c for one of its free variables u. Solving a two-dimensional problem of this form with a solution φ may suggest a generalization that works for all values of u where all occurrences of c in φ are replaced by u.

We found the side condition feature of our workflow important for narrowing down which subdomain was the most challenging for the conjecture in question. For instance, for some cases it was very easy to find invertibility conditions that held when both s and t were normal (resp., subnormal), but very difficult when s was normal and t was subnormal or vice versa.

We also implemented a fully automated mode for the synthesis loop in Figure 5. However, in practice, it was more effective to tweak the generated solutions manually. The amount of user interaction was not prohibitively high in our experience.

Finally, we found that it was often helpful to visualize the input/output behavior of candidate solutions. In many cases, the difference between a candidate solution and the desired behavior of the invertibility condition would reveal a required modification to the grammar or would suggest which parts of the domain of the conjecture to focus on.

4.1 Verifying Conditions for Multiple Formats and Rounding Modes

We verified the correctness of all 167 invertibility conditions by checking them against their corresponding full I/O specification for floating-point formats $\mathbb{F}_{3,5}$, $\mathbb{F}_{4,5}$, and $\mathbb{F}_{4,6}$ and all rounding modes, which required 1.6 days of CPU time. This is relatively cheap compared to computing the specifications, since checking is essentially constant evaluation of invertibility conditions for all possible input values. However, this quickly

$\mathsf{QE}_{\mathsf{FP}}(\exists x.\, P(t_1, \ldots, t_j[x], \ldots, t_n))$, where $x \notin FV(t_i)$ for $i \neq j$:

If $t_j[x] = x$, return $\mathsf{getIC}(x, P)$.
Otherwise, $t_j[x] = \diamond(s_1, \ldots, s_k[x], \ldots s_m)$ where $m > 0$, $x \notin FV(s_i)$ for $i \neq k$.
Let $Q[y] = P(t_1, \ldots, t_{j-1}, \diamond(s_1, \ldots, s_{k-1}, y, s_{k+1}, \ldots, s_m), t_{j+1}, \ldots, t_n)$ where y is a fresh variable.
Return $\mathsf{getIC}(y, Q[y]) \wedge \mathsf{QE}_{\mathsf{FP}}(\exists x.\, s_k[x] \approx y)$.

Fig. 6: Recursive procedure $\mathsf{QE}_{\mathsf{FP}}$ for computing quantifier elimination for x in the unit linear formula $\exists x.\, P(t_1, \ldots, t_j[x], \ldots, t_n)$. The free variables in this formula and the fresh variable y are implicitly universally quantified. Placeholder $\diamond$ denotes a floating-point operator from Table 1.

becomes infeasible with increasing precision, since the time required for computing the I/O specification roughly increases by a factor of 8 for each bit.

As a consequence, we generated quantified floating-point problems to verify the 167 invertibility conditions for formats $\mathbb{F}_{3,5}$, $\mathbb{F}_{4,5}$, $\mathbb{F}_{4,6}$, $\mathbb{F}_{5,11}$ (Float16), $\mathbb{F}_{8,24}$ (Float32), and $\mathbb{F}_{11,53}$ (Float64) and all rounding modes. Each problem checks the T_{FP}-unsatisfiability of formula $\neg(\phi_c \Leftrightarrow \exists x.\, l[x])$, where $l[x]$ corresponds to the floating-point literal, and ϕ_c to its invertibility condition. In total, we generated 3786 problems ($116 * 5 + 51$[6] for each floating-point format) and checked them using CVC4 [5] (master 546bf686) and Z3 [16] (version 4.8.4).

We consider an invertibility condition to be verified for a floating-point format and rounding mode if at least one solver reports unsatisfiable. Given a CPU time limit of one hour and a memory limit of 8GB for each solver/benchmark pair, we were able to verify 3577 (94.5%) invertibility conditions overall, with 99.2% of $\mathbb{F}_{3,5}$, 99.7% of $\mathbb{F}_{4,5}$, 100% of $\mathbb{F}_{4,6}$, 93.8% of $\mathbb{F}_{5,11}$, 90.2% of $\mathbb{F}_{8,24}$, and 84% of $\mathbb{F}_{11,53}$. This verification with CVC4 and Z3 required a total of 32 days of CPU time. All verification jobs were run on cluster nodes with Intel Xeon E5-2637 3.5GHz and 32GB memory.

5 Quantifier Elimination for Unit Linear Floating-Point Formulas

Based on the invertibility conditions presented in Section 3, we can define a quantifier elimination procedure for a restricted fragment of floating-point formulas. The procedure applies to *unit linear* formulas, that is, formulas of the form $\exists x.\, P[x]$ where P is a Σ_{FP}-literal containing exactly one occurrence of x.

Figure 6 gives a quantifier elimination procedure $\mathsf{QE}_{\mathsf{FP}}$ for unit linear floating-point formulas $\exists x.\, P[x]$. We write $\mathsf{getIC}(y, Q[y])$ to indicate the invertibility condition for y in $Q[y]$, which amounts to a table lookup for the appropriate condition as given in Section 3. Note that our procedure is currently a partial function because we do not have yet invertibility conditions for some unit linear formulas. The recursive procedure returns a conjunction of conditions based on the path on which x occurs in P. If x occurs beneath multiple nested function applications, a fresh variable y is introduced

[6] 116 invertibility conditions from rounding mode dependent operators and 51 invertibility conditions where the operator is rounding mode independent (e.g., rem).

and used for referencing the intermediate result of the subterm we are currently solving for. We demonstrate this in the following example.

Example 2. Consider the unit linear formula $\exists x.\ (x \overset{\mathsf{R}}{\cdot} u) \overset{\mathsf{R}}{+} s \geq t$. Invoking the procedure $\mathsf{QE_{FP}}$ on this input yields, after two recursive calls, the conjunction

$$\mathsf{getIC}(y_1, y_1 \overset{\mathsf{R}}{+} s \geq t) \wedge \mathsf{getIC}(y_2, y_2 \overset{\mathsf{R}}{\cdot} u \approx y_1) \wedge \mathsf{getIC}(x, x \approx y_2)$$

where y_1 and y_2 are fresh variables. The third conjunct is trivially equivalent to $\top$. This formula is quantifier-free and has the properties specified by the following theorem.

Theorem 1 *Let $\exists x.\ P$ be a unit linear formula and let $\mathcal{I}$ be a model of T_{FP}. Then, $\mathcal{I}$ satifies $\neg\exists x.\ P$ if and only if there exists a model $\mathcal{J}$ of T_{FP} (constructible from $\mathcal{I}$) that satisfies $\neg\mathsf{QE_{FP}}(\exists x.\ P)$.*

Niemetz et al. [26] present a similar algorithm for solving unit linear bit-vector literals. In that work, a counterexample-guided loop was devised that made use of Hilbert-choice expressions for representing quantifier instantiations. In contrast to that work, we provide here only a quantifier elimination procedure. Extending our techniques to a general quantifier instantiation strategy is the subject of ongoing work. We discuss our preliminary work in this direction in the next section.

6 Solving Quantified Floating-Point Formulas

We implemented a prototype extension of the SMT solver CVC4 that leverages the results of the previous section to determine the satisfiability of quantified floating-point formulas. To handle quantified formulas, CVC4 uses a basic model-based instantiation loop (see, e.g., [32,30] for instantiation approaches for other theories). This technique maintains a quantifier-free set of constraints F corresponding to instantiations of universally quantified formulas. It terminates with the response "unsatisfiable" if F is unsatisfiable, and terminates with "satisfiable" if it can show that the given quantified formulas are satisfied by a model of T_{FP} that satisfies F. For T_{FP}, the instantiations are substitutions of universally quantified variables to concrete floating-point values, e.g. $\forall x.\, P(x) \Rightarrow P(0)$, which can be highly inefficient in the worst case for higher precision.

We extend this basic loop with a preprocessing pass that generates theory lemmas based on the invertibility conditions corresponding to literals of quantified formulas $\forall x.P$ with exactly one occurrence of x, as explained in the example below.

Example 3. Suppose the current set S of formulas contains a formula φ of the form $\forall x.\, \neg((x \cdot u) + s \geq t \wedge Q(x))$ where u, s and t are ground terms; then we add the following formula to S where y_1 and y_2 are fresh (free) variables:

$$(\mathsf{getIC}(y_1, y_1 + s \geq t) \Rightarrow y_1 + s \geq t) \wedge (\mathsf{getIC}(y_2, y_2 \cdot u \approx y_1) \Rightarrow y_2 \cdot u \approx y_1)$$

The addition of this lemma is satisfiability preserving because, if the invertibility condition holds for $y_1 + s \geq t$ (resp., $y_2 \cdot u \approx y_1$), then y_1 (resp., y_2) a solution for that literal.

We then add the instantiation lemma $\varphi \Rightarrow \neg((y_2 \cdot u) + s \geq t \wedge Q(y_2))$. Although x is not necessarily linear in the body of φ, if both invertibility conditions hold, then the combination of the above lemmas implies $(y_2 \cdot u) + s \geq t$, which together with the instantiation lemma allows the solver to infer that the remaining portion of the quantified formula Q cannot hold for y_2. An inference of this form may be more productive than enumerating the possible values of x in instantiations.

Evaluation. We considered all 61 benchmarks from SMT-LIB [6] that contained quantified formulas over floating-points (logic FP), which correspond to verification conditions from the software verification competition that use a floating-point encoding [19]. The invertibility conditions required for solving their literals include floating-point addition, multiplication and division (both arguments) with equality and inequality. We implemented all cases of invertibility conditions for solving these cases. We extended our SMT solver CVC4 (GitHub master 5d248c36) with the above preprocessing pass (GitHub cav19fp 9b5acd74), and compared its performance with (configuration CVC4-ext) and without (configuration CVC4-base) the above preprocessing pass enabled to the SMT solver Z3 (version 4.8.4). All experiments were run on the same cluster mentioned earlier, with a memory limit of 8GB and a 1800 second time limit. Overall, CVC4-base solved 35 benchmarks within the time limit (with no benchmarks uniquely solved compared to CVC4-ext), CVC4-ext solved 42 benchmarks (7 of these uniquely solved compared to the base version), and Z3 solved 56 benchmarks. While CVC4-ext solves significantly fewer benchmarks than Z3, we believe that the improvement over CVC4-base is indicative that our approach for invertibility conditions shows potential for solving quantified floating-point constraints in SMT solvers. A more comprehensive evaluation and implementation is left as future work.

7 Conclusion

We have presented invertibility conditions for a large subset of combinations of floating-point operators over floating-point predicates supported by SMT solvers. These conditions were found by a framework that utilizes syntax-guided synthesis solving, customized for our problem and developed over the course of this work. We have shown that invertibility conditions imply that a simple fragment of quantified floating-points admits compact quantifier elimination, and have given preliminary evidence that an SMT solver that partially leverages this technique can have a higher success rate on floating-point problems coming from a software verification application.

For future work, we plan to extend techniques for quantified and quantifier-free floating-point formulas to incorporate our findings, in particular to lift previous quantifier instantiation approaches (e.g., [26]) and local search procedures (e.g., [25]) for bit-vectors to floating-points. We also plan to extend and use our synthesis framework for related challenging synthesis tasks, such as finding conditions under which more complex constraints have solutions, including those having multiple occurrences of a variable to solve for. Our synthesis framework is agnostic to theories and can be used for any sort with a small finite domain. It can thus be leveraged also for solutions to quantified bit-vector constraints. Finally, we would like to establish formal proofs of correctness of our invertibility conditions that are independent of floating-point formats.

References

1. Alur, R., Bodík, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013. pp. 1–8. IEEE (2013), http://ieeexplore.ieee.org/document/6679385/
2. Alur, R., Radhakrishna, A., Udupa, A.: Scaling enumerative program synthesis via divide and conquer. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 319–336 (2017)
3. Association, I.S.: 754-2008 - IEEE Standard for Floating-Point Arithmetic. https://ieeexplore.ieee.org/servlet/opac?punumber=4610933 (2008)
4. Barr, E.T., Vo, T., Le, V., Su, Z.: Automatic detection of floating-point exceptions. SIGPLAN Not. **48**(1), 549–560 (Jan 2013)
5. 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. pp. 171–177. CAV'11, Springer-Verlag (2011)
6. Barrett, C., Stump, A., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org (2010)
7. Ben Khadra, M.A., Stoffel, D., Kunz, W.: goSAT: Floating-point satisfiability as global optimization. In: FMCAD. pp. 11–14. IEEE (2017)
8. Bjørner, N., Janota, M.: Playing with quantified satisfaction. In: 20th International Conferences on Logic for Programming, Artificial Intelligence and Reasoning - Short Presentations, LPAR 2015, Suva, Fiji, November 24-28, 2015. pp. 15–27 (2015)
9. Blum, L., Blum, M., Shub, M.: A simple unpredictable pseudo-random number generator. SIAM J. Comput. **15**(2), 364–383 (1986)
10. Brain, M., D'silva, V., Griggio, A., Haller, L., Kroening, D.: Deciding floating-point logic with abstract conflict driven clause learning. Formal Methods in System Design **45**(2), 213–245 (2014)
11. Brain, M., Schanda, F., Sun, Y.: Building better bit-blasting for floating-point problems. In: Vojnar, T., Zhang, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part I. Lecture Notes in Computer Science, vol. 11427, pp. 79–98. Springer (2019)
12. Brain, M., Tinelli, C., Rümmer, P., Wahl, T.: An automatable formal semantics for IEEE-754 floating-point arithmetic. In: 22nd IEEE Symposium on Computer Arithmetic, ARITH 2015, Lyon, France, June 22-24, 2015. pp. 160–167. IEEE (2015)
13. Brillout, A., Kroening, D., Wahl, T.: Mixed abstractions for floating-point arithmetic. In: FMCAD. pp. 69–76. IEEE (2009)
14. Conchon, S., Iguernlala, M., Ji, K., Melquiond, G., Fumex, C.: A three-tier strategy for reasoning about floating-point numbers in SMT. In: CAV. pp. 419–435. Springer (2017)
15. Daumas, M., Melquiond, G.: Certification of bounds on expressions involving rounded operators. ACM Transactions on Mathematical Software **37**(1), 1–20 (2010)
16. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. TACAS'08/ETAPS'08, Springer-Verlag (2008)
17. Dutertre, B.: Solving exists/forall problems in yices. Workshop on Satisfiability Modulo Theories (2015)

18. Fu, Z., Su, Z.: XSat: A fast floating-point satisfiability solver. In: CAV. pp. 187–209. Springer (2016)
19. Heizmann, M., Chen, Y., Dietsch, D., Greitschus, M., Nutz, A., Musa, B., Schätzle, C., Schilling, C., Schüssele, F., Podelski, A.: Ultimate automizer with an on-demand construction of floyd-hoare automata - (competition contribution). In: Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II. pp. 394–398 (2017)
20. Lapschies, F.: SONOLAR, the solver for non-linear arithmetic. http://www.informatik.uni-bremen.de/agbs/florian/sonolar (2014)
21. Liew, D.: JFS: JIT fuzzing solver. https://github.com/delcypher/jfs
22. Marre, B., Bobot, F., Chihani, Z.: Real behavior of floating point numbers. In: SMT Workshop (2017)
23. Michel, C., Rueher, M., Lebbah, Y.: Solving constraints over floating-point numbers. In: International Conference on Principles and Practice of Constraint Programming. pp. 524–538. Springer (2001)
24. de Moura, L.M., Bjørner, N.: Efficient e-matching for SMT solvers. In: Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings. pp. 183–198 (2007)
25. Niemetz, A., Preiner, M., Biere, A.: Precise and complete propagation based local search for satisfiability modulo theories. In: Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I. pp. 199–217 (2016)
26. Niemetz, A., Preiner, M., Reynolds, A., Barrett, C., Tinelli, C.: Solving quantified bit-vectors using invertibility conditions. In: Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II. pp. 236–255 (2018)
27. Preiner, M., Niemetz, A., Biere, A.: Counterexample-guided model synthesis. In: Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I. pp. 264–280 (2017)
28. Raghothaman, M., Udupa, A.: Language to specify syntax-guided synthesis problems (05 2014)
29. Reynolds, A., Deters, M., Kuncak, V., Tinelli, C., Barrett, C.W.: Counterexample-guided quantifier instantiation for synthesis in SMT. In: Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II. pp. 198–216 (2015)
30. Reynolds, A., King, T., Kuncak, V.: Solving quantified linear arithmetic by counterexample-guided instantiation. Formal Methods in System Design **51**(3), 500–532 (2017)
31. Scheibler, K., Kupferschmid, S., Becker, B.: Recent improvements in the SMT solver iSAT. MBMV **13**, 231–241 (2013)
32. Wintersteiger, C.M., Hamadi, Y., de Moura, L.M.: Efficiently solving quantified bit-vector formulas. Formal Methods in System Design **42**(1), 3–23 (2013)
33. Zeljić, A., Wintersteiger, C.M., Rümmer, P.: Approximations for model construction. In: IJCAR. pp. 344–359. Springer (2014)