Integrating Proxy Theories and Numeric Model Lifting for Floating-Point Arithmetic

Jaideep Ramachandran*, Thomas Wahl
Northeastern University

Abstract—Precise reasoning for floating-point arithmetic (FPA) is as critical for accurate software analysis as it is hard to achieve. Several recent approaches reduce solving an FPA formula \( f \) to reasoning over a related but easier-to-solve proxy theory. The rationale is that a satisfying proxy assignment may directly correspond to a model for \( f \). But what if it doesn’t? Prior work deals with this case somewhat crudely, or discards the proxy assignment altogether. In this paper we present an FPA decision framework, parameterized by the choice of proxy theory \( T \), that attempts to lift an encountered \( T \)-model to a numerically close FPA model. Other than assuming some “proximity” of \( T \) to FPA, our lifting procedure is \( T \)-agnostic; it is in fact designed to work independently of how the proxy assignment was obtained. Should the lifting fail, our procedure gradually reduces the gap between the FPA and the proxy interpretations of \( f \). We have instantiated the framework using real arithmetic and reduced-precision FPA as proxy theories, and demonstrate that we can, in many cases, decide \( f \) more efficiently than earlier work.

I. INTRODUCTION

Floating point arithmetic, the real-arithmetic (RA) approximation used on most general-purpose computers today, continues to surprise programmers. While computer scientists and mathematicians are aware of the loss of precision such approximation necessarily incurs, many are oblivious to the consequences this can have in programs beyond small inaccuracies in final results. To debug numeric programs effectively before the software is deployed, counterexample-producing analysis tools sensitive to FPA semantics are vital.

The last 5–10 years have seen increased efforts in building floating-point decision procedures. The first such were based on bit-precise encodings: floating-point expressions are encoded as propositional \([5, 11]\) or bitvector logic \([6]\) formulas that formalize the prescription of the IEEE 754 floating-point standard \([10]\). These approaches, while successful, tend to suffer from the size of the encoding that “bit-blasting” entails. Moreover, such very low-level encodings lose the intended numeric proximity of FPA to the real numbers and thus obscure even the simplest identities like \( a + b = b + a \).

The other, and more recent, approach to encoding floating-point formulas is to exploit said numeric proximity. The IEEE standard stipulates that a floating-point computation “shall be performed as if it first produced an intermediate result correct to infinite precision . . . , and then rounded that intermediate result . . . to the destination’s format” \([10]\). Experience has shown that encoding rounding precisely as a mathematical (floating point-free) operation is feasible but expensive \([11]\).

An alternative philosophy is to ignore the rounding altogether, solve the formula interpreted over the proxy theory of real arithmetic using off-the-shelf RA solvers, and check any obtained models for satisfaction of the formula under FPA semantics. This idea has been used successfully to detect floating-point exceptions in C programs \([2]\).

In line with recent work on floating-point model construction \([12]\), we present in this paper a method that extends the paradigm of reasoning about FPA via some proxy theory \( T \) to a (complete, in principle) decision method. Our method begins by abstracting the given floating-point formula \( f \) into the proxy theory \( T \) to obtain a formula \( f_T \) of the same propositional structure but with \( T \) constraints that are assumed to be easier to decide. It then tries to find a \( T \)-model \( \sigma_T \) of \( f_T \). If successful, we cast \( \sigma_T \) to a “nearby” FPA assignment \( \sigma \) (how exactly this is done depends on \( T \)). We now determine whether \( \sigma \models f \) according to FPA semantics; if yes, a model for \( f \) has been found. If \( \sigma \not\models f \), previous methods disagree widely on how to proceed. In \([2]\), where \( T = \text{RA} \), this case is treated as a failure. In \([12]\), where \( T \) is reduced-precision FPA, the authors attempt to reconstruct full-precision FPA models simply by initializing the unused bit positions with zeros.

In this paper we propose a numeric model lifting procedure that exerts much more fine-grained control over how a potential model for \( f \) is obtained. We aim to find this model in the close vicinity of (the non-satisfying) \( \sigma \). To this end, our procedure heuristically determines a subset \( O \) of \( f \)'s variables such that modifying the assignment to these variables slightly has a chance to make \( \sigma \) satisfy. We now partially instantiate \( f \), namely by the assignment \( \sigma \) restricted to the variables outside of \( O \), to obtain a formula \( f' \). This reduces the original decision problem for \( f \) to a decision problem for \( f' \) over only \(|O|\) variables.

Our method now goes a significant step further: instead of solving \( f' \) from scratch, we use a strategy reminiscent of lazy SMT solving: assignment \( \sigma_T \), satisfying the \( T \) abstraction \( f_T \) of \( f \), gives rise to a Boolean model for the propositional skeleton of \( f_T \). Since \( f_T \) is designed to have the same propositional structure as \( f \) (and hence as \( f' \)), we can reuse this skeleton assignment, and simply solve a conjunction \( \Delta \) of FPA constraints: for each constraint in \( f' \), we require its truth value to be the same as that \( \sigma_T \) has assigned to the corresponding \( T \) constraint in \( f_T \).

We summarize the point of constructing \( \Delta \). First, if \( \Delta \) is satisfiable, via some assignment \( \varepsilon \), then so is \( f' \); a satisfying assignment is given by updating \( \sigma \)'s assignment to \( O \)-variables using \( \varepsilon \). Second, \( \Delta \) is a conjunction of FPA constraints (no

*Email: jaideep@cs.neu.edu. Supported by NSF grant CCF-1218075.
propositional structure), and contains only \(|O|\) variables. In our experiments, we found that choosing a single variable in \(O\) often suffices. In that case we have reduced \(f\) drastically, to a univariate conjunction of constraints. This reduced problem can now be given to an FPA solver such as MATHSAT [6], with largely increased prospects for a speedy decision.

If the lifting step does not succeed, or \(f_T\) is unsatisfiable to begin with, our procedure refines \(f_T\), in a manner that depends on the choice of \(T\). The step-wise refinement often turns unsatisfiable abstractions \(f_T\) into satisfiable ones.

A classical example are formulas debunking “false identitities", like \((x + y) + z > x + (y + z)\), which is unsatisfiable in RA, but becomes satisfiable after a one-step refinement; Sect. [11] illustrates this in detail. If, for each intermediate abstraction \(f_T\), a \(T\) model cannot be found or the subsequent lifting fails, the iterative process eventually refines \(f_T\) to \(f\); the search for models in the proxy theory was in vain. In the spirit of [12], our method is intended for fast model construction.

We have experimented with two proxy theories in this paper: real arithmetic and reduced-precision floating-point (Sect. V). Both are often easier to solve than FPA [2], [12].

We finally note that special floating-point values like infinities and NaNs will occur in the assignment \(\sigma_T\) only if the proxy theory \(T\) is “aware” of such values (e.g. RA is not). Since the model lifting process presented in this paper is designed to be \(T\)-agnostic, we mostly avoid discussing special values. Our implementation currently enforces their absence in \(\sigma_T\) for proxy theories that have them. Incorporating special values fully into our framework is left for future work.

II. A MOTIVATING EXAMPLE

Our approach deals with floating-point formulas of propositional structure in a way that is reminiscent of lazy SMT solving; we present the details of this in Sect. II. In the present section we focus on the theory-specific (numeric) aspects. Consider therefore the atomic floating-point formula

\[
f :: (a_1 + a_2) + a_3 > a_1 + (a_2 + a_3)
\]  

where \(\oplus\) denotes floating-point addition. To keep the presentation succinct in this section, we assume single-precision and round-to-negative as rounding mode. We will demonstrate how our proposed framework processes this formula using real arithmetic as proxy theory (\(T = RA\)).

Motivated by the success of earlier work in finding floating-point models by searching in the reals instead [2], we express this formula in the logic of real arithmetic to obtain \(f_T :: (a_1 + a_2) + a_3 > a_1 + (a_2 + a_3)\), and give it to an SMT solver. The solver responds that \(f_T\) is unsatisfiable.

With the determination to construct a model in mind, our technique mistrusts the UNSAT result and proceeds by increasing the precision of the abstraction. Fortunately, we can perform this refinement in a lazy manner, by interpreting parts of \(f\) in floating-point, others in real arithmetic. Suppose we decide that the top-level \(+\) of the right-hand side expression in \(f_T\) is to be interpreted in (refined to) FPA. This turns \(f_T\) into the formula

\[
f_T' :: (a_1 + a_2) + a_3 > a_1 \oplus (a_2 + a_3)
\]  

The domain of all variables remains the real numbers. This is a formula in Mixed Real-FPA (MRFPA); details of its semantics are given in Sect. V.

Why is this refinement useful? The answer is that chances of finding a model for \(f\) by examining \(f_T'\) are higher than doing so by examining \(f_T\), since the semantics of MRFPA will ensure that the \(\oplus\) in \(f_T'\) implements floating-point addition (although its operands are reals; details in Sect. V). In addition, the cost of examining \(f_T'\) is only moderately higher than that of examining \(f_T\), and hopefully lower than that for \(f\). To analyze \(f_T'\) we need solver support for MRFPA, which is given by (an extension of) the tool REALIZER [11] details in Sect. V.

Giving \(f_T'\) to the extension of REALIZER, we obtain—for the first time—a satisfying assignment \(\sigma_T\), namely

\[
\sigma_T :: a_1 = a_2 \approx 1.1755 \cdot 10^{-38}, \quad a_3 \approx 1.9722 \cdot 10^{-31}.
\]

The left hand side term of \(f_T'\) evaluates slightly larger than the right hand side. We now project these real numbers to single-precision floating-point, which is done simply by rounding. We then apply the resulting assignment, call it \(\sigma\), to the floating-point formula \(f\). Unfortunately, \(\sigma\) does not satisfy \(f\): the left-hand and right-hand side sums turn out to be the same.

Instead of immediately refining \(f_T'\) further, our method does not give up the hope that a model for \(f\) can be found in a neighborhood of \(\sigma\). We therefore now try to “nudge” this assignment so that it satisfies \(f\). The plan is simple: we pick one of the \(a_i\) variables to modify—say our choice is \(a_3\)—while leaving all others constant. We then build a new, univariate formula \(\Delta\) as follows:

\[
\Delta :: (\sigma_1 \oplus \sigma_2) + a_3 > \sigma_1 \oplus (\sigma_2 + a_3),
\]

where, for \(i = 1, 2, \sigma_i := \sigma(a_i)\). By design of our method, if \(\Delta\) is satisfiable, say via \(\varepsilon\), then so is \(f\), and we obtain a satisfying assignment for \(f\) from \(\sigma\) by changing the value assigned to \(a_3\) using \(\varepsilon\). The key is that \(\Delta\) is simpler than \(f\): it contains only one free variable \((a_3)\). We have reduced the original floating-point decision problem to a much simpler one such that any model for the simpler problem gives rise to a satisfying assignment for \(f\).

Finishing up our example: applying the solver MATHSAT [6] to \(\Delta\) we learn that increasing \(a_3\) by \(1.1755 \cdot 10^{-38}\) leads to a satisfying assignment for \(f\): the left sum is now larger.

Recent work uses reduced-precision FPA as proxy theory [12] and attempts to “patch” a proxy assignment (like \(\sigma_T\)) to a satisfying floating-point one using syntactic means: by padding the lower-precision bitvector assignment with 0s or 1s. This initially fails and requires more refinement iterations, ultimately entailing higher cost, as our experiments will show. In contrast, our method takes the numeric circumstances into account, as reflected in formula \(\Delta\). As a result, a satisfying assignment for \(\Delta\) guarantees the existence of a model for \(f\).
III. DECIDING FPA USING A PROXY THEORY AND MODEL LIFTING

We describe in this section our procedure for deciding a floating-point formula \( f \); see Fig. 1. In addition to \( f \), the (implicit) input to the procedure includes floating-point specifics like the format parameters for range and precision, as well as settings like the rounding mode, which we assume to apply across the entire formula. If \( f \) is determined to be satisfiable, the algorithm returns a satisfying assignment \( \sigma \).

![Diagram](image)

Fig. 1: Deciding FPA formula \( f \) via proxy theory \( T \)

The procedure begins by mapping \( f \) to a formula over the proxy theory \( T \); we can view the resulting \( f_T \) as an abstraction of \( f \). (In general, though, it is neither an over- nor an underapproximation of \( f \); this happens to be immaterial for our procedure.) How this map is defined is clearly \( T \)-specific. However, we require that it maintains the propositional structure (skeleton) of \( f \) and applies only to its (atomic) theory constraints. For example, in the case of reduced-precision FPA as the proxy theory, the mapping simply changes the floating-point format parameters that come with \( f \). In the case \( T = \text{RA} \) (real arithmetic), the mapping causes all arithmetic function symbols and constants to be interpreted over the reals. We give more details on these specific proxy theories in Sect. V.

Given formula \( f_T \) in theory \( T \), the procedure now repeatedly tries to find a model for \( f_T \) and to “lift” that to a model for \( f \). If no model for \( f_T \) can be found, or the lifting fails, it refines \( f_T \) so as to narrow the semantic gap to the input formula \( f \). This is reflected in Fig. 1 as follows. The procedure first performs the satisfiability check \( \exists \sigma_T. \sigma_T |\Rightarrow_T f_T \) (where \( \Rightarrow_T \) is the satisfaction relation in \( T \)). If the check fails, \( f_T \) is refined; more on that below. If a satisfying assignment \( \sigma_T \) is found, we call a procedure \( \text{toFloat} \) that casts the \( T \)-assignment \( \sigma_T \) to a “nearby” floating-point assignment \( \sigma \). This step is \( T \)-dependent and may for instance involve rounding (when \( T \) is “more precise” than standard FPA) or fresh bit initialization (when \( T \) is “less precise” than standard FPA).

We now ask whether \( \sigma \) is indeed a satisfying floating-point assignment for \( f \). This amounts to plugging in the values given by \( \sigma \), and evaluating the grounded formula \( f \). Unless the satisfiability of \( f \) depends on floating-point peculiarities such as the lack of associativity (example in Sect. III), the query \( \sigma \models f \) may well succeed (\( \models \) is the satisfaction relation in FPA); this was observed in [2] for a large fraction of their (floating-point exception) benchmarks. In that case the procedure terminates, returning \( \sigma \).

A negative result to the query \( \sigma \models f \) is interpreted by the procedure to mean that the floating-point solution to \( f \) that we are suspecting in the vicinity of \( \sigma_T \) cannot be obtained simply by rounding or syntactic initialization. We therefore launch a more aggressive subroutine \( \text{Lift} \) that tries to modify the values assigned by \( \sigma \) to certain variables of \( f \) to force \( \sigma \) to be satisfying. This routine is described in Sect. [V]. Note that, while \( \text{toFloat} \) casts a \( T \)-assignment to floating point, \( \text{Lift} \) maps one floating-point assignment to another.

The \( \text{Lift} \) procedure is designed such that, if there exists a satisfying floating-point assignment in the vicinity of \( \sigma \), \( \text{Lift} \) will eventually find it, given enough time. In this case, \( \text{Lift} \) in Fig. 1 returns the new \( \sigma \); the procedure terminates. Lifting can fail because \( f \) is unsatisfiable, or because the lifting timed out. The latter indicates that assignment \( \sigma \) is not a good starting point for finding a model for \( f \). We increase the precision of the abstraction, by refining \( f_T \).

The \( \text{Refine} \) step fails when, upon invocation, \( f_T \) is “equivalent” to \( f \) in a sense that depends on the abstraction map. For the case of reduced-precision FPA as proxy theory, this simply means the floating-point format of \( f_T \) equals that of \( f \). In that case, the satisfiability check \( \exists \sigma_T. \sigma_T |\Rightarrow_T f_T \) in the previous iteration was actually a satisfiability check for \( f \). Since that did not succeed, \( f \) is unsatisfiable.

**Correctness.** Termination of the framework can be enforced with some “cooperation” from \( T \); we assume \( T \) to be chosen such that mapping \( f \to T \), and the calls \( \text{toFloat}(\sigma_T) \) and \( \text{Refine}(f_T) \) are straightforward and “fast”. The test \( \sigma \models f \) is trivial. The decision problem \( \exists \sigma_T. \sigma_T |\Rightarrow_T f_T \) may not terminate, e.g., due to undecidability of \( T \). We solve this problem by enforcing a timeout for this step. The call \( \text{Lift}(\sigma, \sigma_T, f_T) \) is discussed in detail in Sect. [V]. As we shall see, it introduces no potential for nontermination.

Finally, the framework itself (Fig. 1) contains a loop. We require of the proxy theory that it permits gradual refinement of \( T \) formulas to floating-point formulas. How this is done exactly depends on \( T \) and is discussed, for two instances, in Sect. [V]. With these provisions, instances of the framework in Fig. 1 are terminating. The framework is also easily seen to be sound for SAT and UNSAT outcomes; we omit the details.
We begin by selecting all (The algorithm also has access to the [unchanged] formula $\phi$ and an assignment $P = p$ produced during the main procedure in Fig. 1 all maintain the or $f$ FPA theory constraints of $P$. The goal of the model refinement procedure Lift is to reconcile this difference, thereby lifting assignment $\sigma$ to a proper model for $f$.

The basic idea is as follows. Given that $\sigma_T \models_T f_T$, there exists an assignment to the variables in the Boolean skeleton of $f_T$ that makes this propositional skeleton formula true. Since, by construction, $f$ and $f_T$ have the same Boolean skeleton (this is required of the abstraction; see Sect. III), the goal is to modify the assignment to the floating-point variables in $f$ such that the corresponding Boolean skeleton assignment coincides with that induced by $\sigma_T$. If we succeed, $f$ is satisfied. This turns the original FPA formula $f$ into a structurally simple conjunction of FPA theory constraints, since the “target” Boolean value for these constraints is determined via $\sigma_T$.

We point out parallels of this reduction of formula structure to lazy SMT solving: there, a formula $f$ over a background theory $T$ is solved by first applying a $T$-oblivious propositional SAT solver to $f$’s Boolean skeleton. A solution gives rise to a conjunction of $T$ constraints a model for which is a model for $f$. In our work we do not use a SAT solver — it is too weak for our purposes: we seek a (proxy) theory assignment that makes the skeleton true and gives hope that a satisfying FPA assignment can be found nearby.

**Notation.** Let $V$ be the set of arithmetic variables in $f$, and let $P$ be the set of propositional variables in the Boolean skeleton of $f$. We can think of variables $p \in P$ as pointers to the FPA theory constraints of $f$. Recall that the abstractions $f_T$ produced during the main procedure in Fig. 1 all maintain the skeleton of $f$. Hence, there exists a function $\gamma$ that takes $p \in P$ and $f$ or $f_T$ as input and returns the theory constraint of $f$ or $f_T$ pointed to by $p$. Consider this example for $T = \mathbb{R}$:

$$f = x \oplus y > 10 \lor x \oplus y < 7$$

$$f_T = x + y > 10 \lor x - y < 7.$$  

Choosing $p_1 \lor p_2$ as the skeleton, we then have $V = \{x, y\}$, $P = \{p_1, p_2\}$, and

$$\gamma(p_1, f) = (x \oplus y > 10), \quad \gamma(p_2, f) = (x \oplus y < 7)$$

$$\gamma(p_1, f_T) = (x + y > 10), \quad \gamma(p_2, f_T) = (x - y < 7).$$

We finally use $Eval$ to denote a function that takes a formula $\varphi$ and an assignment $A$ to all variables in $\varphi$ and returns the Boolean value of $\varphi$ under $A$, respecting the semantics of $\varphi$.

Our lifting procedure is shown in Alg. 1. The algorithm scheme receives the FPA assignment $\sigma$ that fails to satisfy $f$, the $T$ assignment $\sigma_T$ that does satisfy $f_T$, and formula $f_T$. (The algorithm also has access to the [unchanged] formula $f$.) We begin by selecting all invertible constraints $\text{Inv}$ in $f$ (via pointers $p$): those for which assignments $\sigma_T$ and $\sigma$ disagree in the Boolean value assigned in $f_T$ and $f$, resp.

In Line 2 we select a set $O$ of offset variables: floating-point variables whose assignment we plan to modify to make $\sigma$ satisfying. An upper bound on $O$ is that each $v \in O$ must be contained in at least one invertible constraint. More details on the selection are given in Implementation below. In Line 3 we build the set $P_O$ of (pointers to) constraints that contain at least one $O$-variable; these are the constraints whose truth value may be affected when assignments to $O$-variables are modified.

Line 4 modifies $f$ to $f'$ by instantiating every non-offset variable $v \in V \setminus O$ by its literal floating-point assignment $\sigma(v)$. Finally, in Line 5 we construct a constraint $\Delta$ that realizes the above basic idea: for each theory constraint $\gamma(p, f')$ in $f'$ with at least one $O$-variable, enumerated via pointers $p \in P_O$, we require that it be assigned the truth value $Eval(\gamma(p, f_T), \sigma_T)$ (a constant) given by $\sigma_T$ to the corresponding theory constraint $\gamma(p, f_T)$ in $f_T$. The right conjunct of formula $\Delta$ restricts the assignment to $O$-variables $v$ to some interval around $\sigma(v)$; terms $v.l$ and $v.r$ are floating-point literals (see Interval constraints below).

Intuitively, constraint $\Delta$ is satisfiable whenever there is a satisfying assignment to $f$ in some small neighborhood of $\sigma = toFloat(\sigma_T)$. We are hopeful this is the case, since $\sigma_T$ satisfies $f_T$. Hence, if there exists a satisfying assignment $\varepsilon$ to $\Delta$, we modify $\sigma$ by updating, using $\varepsilon$, the values assigned to $O$-variables. If $\varepsilon$ does not exist, the lifting fails.

**Implementation.** Our lifting procedure has reduced the floating-point decision problem for $f$ to that for $\Delta$. Is the reduced problem simple enough that we can solve it using an off-the-shelf FPA decision procedure? Formula $\Delta$ is a conjunction of constraints — no propositional reasoning is required to decide it. The free variables in $\Delta$ are precisely the offset variables. The choice of set $O$ thus crucially influences the variable complexity of $\Delta$; we use heuristics to keep it small. If $v$ occurs in expensive constraints in $f$, such as in high-degree polynomials or other non-linear terms, the variable ranks low in our selection heuristics. For example, if $f$ contains the quadratic form $x \otimes x \otimes x \otimes y$, our heuristic chooses $O = \{y\}$; Line 4 in Alg. 1 turns the entire term into the univariate, linear floating-point term

$$\sigma(x) \otimes \sigma(x) \otimes \sigma(x) \otimes y.$$  

**Interval constraints.** Gradient analysis of $f$ in a neighborhood of assignment $\sigma$ may reveal that a variable $v$ needs to be increased, say. In this case, we use a lower bound $v.l = \sigma(v)$ in the range constraint $v.l \leq v \leq v.r$ in Line 5. For example, for $f = x \otimes y > 4.0 \land x \otimes y < 2.0$ with $\sigma = \{x = 3.0, y = 1.0\}$, gradient analysis reveals that $y$ needs to be increased; we set $y.l = 1.0$. In the absence of such information, we choose interval $[v.l, v.r]$ to be symmetric around $\sigma(v)$, of a width that is a small fraction of $|\sigma(v)|$.  

IV. MODEL REFINEMENT: FROM PROXY MODELS TO FPA MODELS

We revisit Fig. 1. Suppose formula $f_T$ (the current abstraction of floating-point formula $f$) is satisfiable and gives rise to a model $\sigma_T$. Suppose further that the cast operation $toFloat(\sigma_T)$ yields a non-satisfying assignment $\sigma$ for $f$. This means that $\sigma$ assigns to at least one FPA constraint in $f$ a different Boolean value than $\sigma_T$ does to the corresponding $T$ theory constraint in $f_T$. The goal of the model refinement procedure Lift is to reconcile this difference, thereby lifting assignment $\sigma$ to a proper model for $f$. 


Algorithm 1 (Scheme) Lifting \( \sigma \) to FPA model

Input: \( \sigma \): FPA assignment (falsifying \( f \)), \( \sigma_T \): \( T \) assignment (satisfying \( f_T \)), \( f_T \): abstract formula

1: \( Inv := \{ p \in P \mid Eval(\gamma(p,f_T),\sigma_T) \neq Eval(\gamma(p,f),\sigma) \} \) \( \triangleright \) invertible constraints
2: select subset \( O \) of \( \bigcup_{i \in Inv} \text{Vars}(\gamma(i,f)) \)
3: \( P_O := \{ p \in P \mid \text{Vars}(\gamma(p,f)) \cap O \neq \emptyset \} \) \( \triangleright \) offset variables
4: \( f'_T := f \mid_{v \mapsto \sigma(v)} \) \( \triangleright \) O-affected constraints
5: \( \Delta := \bigwedge_{p \in P_O} \gamma(p,f'_T) = \text{Eval}(\gamma(p,f_T),\sigma_T) \land \bigwedge_{v \in \gamma(O)} v.1 \leq v \leq v.r \) \( \triangleright \) partially instantiated formula
6: if \( \exists \varepsilon, \varepsilon = \Delta \) then
7: for each \( v \in O \)
8: \( \sigma(v) := \varepsilon(v) \)
9: return \( \sigma \)
10: else
11: return failure

V. Proxy Theories for Floating-Point Arithmetic

We have instantiated our framework with two proxy theories at opposite ends of the precision spectrum: reduced-precision FPA and real arithmetic (which can, somewhat awkwardly, be viewed as an infinite-precision “approximation” of FPA). The former is a fairly obvious candidate: an FPA formula is abstracted by interpreting it over a floating-point format with smaller precision and/or range. Reduced-precision FPA is almost invariably easier to solve. Models can be cast to original-precision FPA by initializing the fresh bits to 0. Stepwise refinement consists of gradually increasing the precision (in our work: across the entire formula; more sophisticated schemes are possible). \( T \) is therefore actually the family of FPA theories parameterized by precision/range. Such proxy theories have been used before [12] without numeric lifting]. We discuss instead a less obvious choice for \( T \) in this section.

Real Arithmetic as Proxy Theory

As suggested in Sect. [1] and reported earlier [2], real arithmetic (RA) is suitable for an approximate interpretation of a floating-point formula \( f \): many formulas are easier to decide over the reals, since the complexity of rounding is avoided. A satisfying real assignment can easily be cast to a floating-point assignment via rounding. To enable stepwise refinement of the RA-interpretation of \( f \) back to FPA, however, we need a proxy theory that can express combinations of real and floating-point terms, such as \( a_1 \oplus (a_2 + a_3) \) (Sect.[1]).

Our proxy theory therefore is actually not real arithmetic, but an extension that we call Mixed Real-Floating-Point Arithmetic (MRFPA) and define as follows. Let \( \mathbb{R} \) be the set of real numbers, and \( \mathbb{F} \) be the numbers in \( \mathbb{R} \) representable in floating-point over some fixed precision and range (these parameters are constant in this section). Let \( \text{rd}: \mathbb{R} \rightarrow \mathbb{F} \) be the function that implements the given rounding mode, and let \( \text{Var}_\mathbb{R} \) and \( \text{Var}_\mathbb{F} \) be a set of real and floating-point variables, resp.

The syntax of MRFPA formulas \( f \) is as follows.

\[
\begin{align*}
  f &::= t_\mathbb{R} \theta_\mathbb{R} t_\mathbb{F} | t_\mathbb{F} \theta_\mathbb{F} t_\mathbb{F} - f | f \lor f \\
  \theta_\mathbb{R} &::= < | = \\
  \theta_\mathbb{F} &::= <_\mathbb{F} | =_\mathbb{F} \\
  \alpha_\mathbb{R} &::= + | \times | / \\
  \alpha_\mathbb{F} &::= \oplus | \otimes | \ominus \\
  \alpha_\mathbb{M} &::= +_\mathbb{M} | \times_\mathbb{M} | /_\mathbb{M}
\end{align*}
\]

\[ (3) \]

Intuitively, MRFPA formulas are built over \( \mathbb{F} \) terms \( t_\mathbb{F} \), which evaluate to elements of \( \mathbb{F} \), and \( \mathbb{R} \) terms \( t_\mathbb{R} \), which more generally evaluate to elements of the superset \( \mathbb{R} \). \( \mathbb{R} \) terms are formed using real operators \( \alpha_\mathbb{R} \), \( \mathbb{F} \) terms are formed using floating-point operators \( \alpha_\mathbb{F} \) or mixed operators \( \alpha_\mathbb{M} \). Operators \( \alpha_\mathbb{M} \) can take operands that are floating-point representable, and those that are not. There are no mixed comparison operators \( <_\mathbb{M} | =_\mathbb{M} \), as they are identical to the real operators \( < | = \).

The semantics of MRFPA formulas is defined recursively via an overloaded evaluation function \( [\cdot] \) that maps \( \mathbb{R} \) terms to elements of \( \mathbb{R} \), \( \mathbb{F} \) terms to elements of \( \mathbb{F} \), and formulas to a Boolean value, as follows. Let \( A_\mathbb{R} : \text{Var}_\mathbb{R} \rightarrow \mathbb{R} \) be an \( \mathbb{R} \) assignment to variables in \( \text{Var}_\mathbb{R} \), and \( A_\mathbb{F} : \text{Var}_\mathbb{F} \rightarrow \mathbb{F} \) be an \( \mathbb{F} \) assignment to variables in \( \text{Var}_\mathbb{F} \). The semantics of terms is as follows:

\[
\begin{align*}
  [c] &::= c \text{ for constants } c \in \mathbb{R} \cup \mathbb{F}, [u] = A_\mathbb{R}(v) \text{ for } v \in \text{Var}_\mathbb{R} \text{ and } [v] = A_\mathbb{F}(v) \text{ for } v \in \text{Var}_\mathbb{F}, \\
  [t_1 \alpha \theta t_2] &::= [t_1] \alpha [t_2] \\
  [t_1 \alpha_\mathbb{F} t_2] &::= [t_1] \alpha_\mathbb{F} [t_2] \\
  [t_1 \alpha_\mathbb{M} t_2] &::= \text{rd}(\gamma([t_1][\alpha_\mathbb{M}])[t_2])
\end{align*}
\]

where \( +_\mathbb{M} = + \), \( \times_\mathbb{M} = \otimes \), etc. Operators \( \alpha_\mathbb{M} \) differ from the corresponding real operators \( \alpha_\mathbb{R} \) in that they round the result. They also differ from the corresponding floating-point operators \( \alpha_\mathbb{F} \); the latter take only \( \mathbb{F} \) terms as inputs.

The semantics of an MRFPA formula \( f \) is then as follows:

\[
\begin{align*}
  [t_1 \theta \theta t_2] &::= [t_1] \theta [t_2] - [f] \\
  [t_1 \theta_\mathbb{F} t_2] &::= [t_1] \theta_\mathbb{F} [t_2] \left( [f] \lor [f/2] \right) \left( f/2 \right)
\end{align*}
\]

Our definition of MRFPA ignores numeric anomalies such as infinities and NaNs; see discussion in Sect.[1].

The use of MRFPA as proxy theory requires specific solver support, such as obtained by extending the tool REALIZER [11]. The tool translates floating-point formulas into
numerically equivalent formulas over mixed real-integer arithmetic (RIA): it replaces \( x \oplus y \) by \( \text{rd}(x + y) \), where \( \text{rd} \) encodes rounding as a RIA operation involving floor and ceiling functions. Our (straightforward) extension permits MRFPA as input, not just floating-point formulas.

In practice, deciding real-integer arithmetic is costly and in fact undecidable in the non-linear case. We have therefore experimented with MRFPA as proxy theory only for linear formulas; the prospects for extending this to richer classes are discussed in Sect. VII

VI. EXPERIMENTAL EVALUATION

The techniques described in this paper have been implemented in our tool MOLLY (roughly, “Model Lifter”), both with reduced-precision FPA as proxy theory (called “RPFPA” in the sequel), and with real arithmetic as proxy.

Tool set-up. For our RPFPA experiments, we used MATHSAT [6 v5.3.8] to obtain proxy models and also to solve the constraint during the lifting of proxy models to FPA models. For lifting, MOLLY picks one variable at a time; currently in an arbitrary way. The formula refinement process increases the number of bits in the exponent by 1, and in the mantissa by 3.

We compare against MATHSAT and against the technique presented in [12] called “Approx” there and in Table I. We used MATHSAT with the options input=smt2-model,-theory.eq_propagation=false-theory.fp.bit_blast_mode=1 both when used inside our tool and also when used stand-alone for the comparison. For comparison with “Approx”, we used our own tool MOLLY but with model lifting turned off: our routine toFloat then exactly implements the “padding” used in [12]. Not using their implementation allows us to exactly assess the contribution of the lifting.

All evaluations were performed on a machine with Intel(R) Core (TM) i7-4770 3.40GHz CPU, having 8 GB RAM and running x86_64 Ubuntu 14.04 LTS. An overall timeout (TO) of 20 min was used for each benchmark for every tool.

Benchmarks. We evaluated our technique primarily on two benchmark sets. The first benchmark set, named “I. Non-linear benchmarks from [4]” in Table I, contains a mix of 213 formulas from prior published work [4]. Since we currently do not support casts, we ignored them and interpreted all operations as being for the same (single) precision. We also disallowed special floating-point values in the solution by adding the SMT-LIB assertion \( \text{fp.isNormal} \) for every variable.

The second set of benchmarks, named “II. False Identity benchmarks” in Table I were created by us and are available for download here. These are formulas of the form \( E \equiv \tilde{E} \geq \epsilon \) along with range constraints on the input variables; the expression \( \tilde{E} \) is obtained from \( E \) using a real-arithmetic rewrite rule, i.e. \( \tilde{E} \) is mathematically equivalent to \( E \). Some of the simpler polynomials, for instance, involve factors, e.g. comparing deviation of \( x^3 - y^3 \) from the product of its factors \((x - y) \) and \((x^2 + xy + y^2)\), for a specific ordering of operations. We have formulas for such comparisons for a variety of polynomials, ranging from Horner scheme evaluations to power series expansions for the \( \sin \) function. Such decision problems are relevant for optimizing compilers since a rewrite based on an equivalence in real arithmetic is often unsafe in FPA. These benchmarks are all satisfiable and values of \( \epsilon \) were chosen such that MATHSAT solves each of these in less than 5 min.

Results. Running MOLLY on the first set of non-linear benchmarks confirmed the results reported in [12]: solving a simpler reduced precision approximation, often with the initial reduced precision of 3 bits for each of mantissa and exponent, suffices to solve a significant number of the satisfiable constraints. There were only some opportunities for numeric model lifting; the results on all those 22 benchmarks are reported in the set “I. Non-linear benchmarks from [4]” in Table II (1–22). A majority of these benchmarks turned out to be satisfiable and for the rest the satisfiability status is still unknown. To evaluate effectiveness of model lifting, a liberal timeout of 12 min was set for the numeric model lifting step. From Table II MATHSAT solves one benchmark more than MOLLY, which in turn solves one more than “Approx”. The average solving time per solved benchmark for MOLLY (219s) is greater than that for “Approx” (127s) but lesser than that for MATHSAT (443s). For the set of “False Identity benchmarks” in Table I (23–37), we used a timeout of 3 min per iteration for the reduced precision solving and a timeout of 1 min for the model lifting stage. MATHSAT and MOLLY solve all the 15 benchmarks, with MOLLY taking the least average time per benchmark (86s), closely followed by “Approx” (89s), which timed out on two.

Real arithmetic as proxy theory

We also evaluated MOLLY on a set of constraints consisting of linear formulas that involve checking non-associativity of FPA operations. We assumed single-precision FPA, with round-to-nearest-even rounding mode for FPA operations. MOLLY uses our real arithmetic abstraction detailed in Sect. VII for solving MRFFPA formulas, we extended the tool REALIZER [11], which previously accepted pure FPA formulas as input, to also accept MRFFPA formulas that are generated in the first formula refinement. In this case, the refinement step marks real arithmetic operators in some parts of the formula as FPA operators.

In Table III #Vars indicates the size of the formula, e.g. for

\[ ((a_1 + a_2) + (a_3 + a_4) + a_5) > (((a_1 + a_2) + a_3) + a_4) + a_5). \]  

MOLLY outperforms MATHSAT and is also seen to scale well. In each case, after a few iterations, our model lifting technique succeeded in transforming a real arithmetic assignment into a satisfying floating-point assignment. Based on a simple analysis of the behavior of the expressions constituting the formula in the neighborhood of the approximate assignment, a single variable was chosen to invert the result of the comparison. As before, we used our tool with model lifting disabled to mimic the tool “Approx” from [12]. Here we also ran the actual tool from [12]: it performed many more iterations and eventually timed out on each instance.
Table I: Numeric model lifting on non-linear problems.

Table I, Table II and Table III indicate MOLLY is efficient on benchmarks that require staying close to the original precision to find satisfying assignments. Numeric model lifting then closes the gap between the abstract but imprecise (with respect to FPA) solutions and genuine floating-point arithmetic.

VII. RELATED WORK

The idea of using real arithmetic to solve floating-point constraints approximately has been implemented before [2]. The earlier approach uses this real arithmetic approximation only once for a formula and is hence incomplete, for instance, a formula that is unsatisfiable in the reals but satisfiable in floating-point can not be handled. In contrast, as shown in Sect. [VI] we can handle such an input formula by refining the formula iteratively when the answer obtained in an iteration is not a correct answer to the original formula.

The above mentioned earlier work aims to detect exceptions in floating-point programs, by encoding, in real arithmetic, path conditions of programs as well as exceptional conditions like underflow, overflow, division by zero and certain invalid operations involving NaN. This approximation, ignoring rounding entirely, was sufficient to detect several exceptions, primarily of the underflow and overflow types, in a publicly available library. However, we eventually encode rounding for every operation as per the IEEE 754 standard, as we intend our procedure to be used to uncover bugs due to rounding, for instance, in floating-point comparisons in control flow conditions.

A framework for using abstractions that are neither under approximations nor over approximations of the original formulas was proposed recently [12]. These approximations are refined iteratively as necessary. The authors instantiated this framework for floating-point arithmetic using lower precision floating-point numbers. We extend this idea using numeric model lifting techniques.

In the above work, the authors mention very simple heuristics, like padding the solutions with 0s, for lifting a satisfying assignment from a lower (s, e) to one for the actual problem, but these are unlikely to succeed for many cases, especially in the context of detecting anomalies due to floating-point peculiarities or when the approximate assignment contains

<table>
<thead>
<tr>
<th>Problem</th>
<th>I</th>
<th>Lifted?</th>
<th>Time (s)</th>
<th>I</th>
<th>Lifted?</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>✓</td>
<td>7.8</td>
<td>2</td>
<td>5.0</td>
<td>344.0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>✓</td>
<td>15.8</td>
<td>2</td>
<td>12.3</td>
<td>986.5</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>×</td>
<td>60.1</td>
<td>2</td>
<td>45.6</td>
<td>995.9</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>-</td>
<td>TO</td>
<td>-</td>
<td>TO</td>
<td>977.5</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>-</td>
<td>TO</td>
<td>-</td>
<td>TO</td>
<td>977.4</td>
</tr>
<tr>
<td>6</td>
<td>-</td>
<td>-</td>
<td>TO</td>
<td>-</td>
<td>TO</td>
<td>977.4</td>
</tr>
<tr>
<td>7</td>
<td>-</td>
<td>-</td>
<td>TO</td>
<td>-</td>
<td>TO</td>
<td>983.5</td>
</tr>
<tr>
<td>8</td>
<td>-</td>
<td>-</td>
<td>TO</td>
<td>-</td>
<td>TO</td>
<td>983.5</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>×</td>
<td>337.1</td>
<td>8</td>
<td>330.8</td>
<td>TO</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>✓</td>
<td>3.2</td>
<td>2</td>
<td>0.3</td>
<td>61.8</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>×</td>
<td>680.5</td>
<td>2</td>
<td>0.3</td>
<td>TO</td>
</tr>
<tr>
<td>12</td>
<td>7</td>
<td>✓</td>
<td>863.3</td>
<td>-</td>
<td>TO</td>
<td>TO</td>
</tr>
<tr>
<td>13</td>
<td>-</td>
<td>-</td>
<td>TO</td>
<td>-</td>
<td>TO</td>
<td>TO</td>
</tr>
<tr>
<td>14</td>
<td>-</td>
<td>-</td>
<td>TO</td>
<td>-</td>
<td>TO</td>
<td>TO</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>✓</td>
<td>0.9</td>
<td>8</td>
<td>137.2</td>
<td>108.2</td>
</tr>
<tr>
<td>16</td>
<td>8</td>
<td>×</td>
<td>484.7</td>
<td>8</td>
<td>116.6</td>
<td>46.7</td>
</tr>
<tr>
<td>17</td>
<td>8</td>
<td>×</td>
<td>350.3</td>
<td>8</td>
<td>322.2</td>
<td>47.0</td>
</tr>
<tr>
<td>18</td>
<td>2</td>
<td>✓</td>
<td>4.9</td>
<td>6</td>
<td>29.4</td>
<td>46.8</td>
</tr>
<tr>
<td>19</td>
<td>2</td>
<td>✓</td>
<td>22.1</td>
<td>3</td>
<td>32.5</td>
<td>47.2</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>✓</td>
<td>3.3</td>
<td>2</td>
<td>6.3</td>
<td>46.5</td>
</tr>
<tr>
<td>21</td>
<td>2</td>
<td>✓</td>
<td>253.4</td>
<td>3</td>
<td>599.9</td>
<td>46.8</td>
</tr>
<tr>
<td>22</td>
<td>3</td>
<td>✓</td>
<td>39.1</td>
<td>4</td>
<td>118.8</td>
<td>65.7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Problem</th>
<th>I</th>
<th>Lifted?</th>
<th>Time (s)</th>
<th>I</th>
<th>Lifted?</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>3</td>
<td>✓</td>
<td>148.6</td>
<td>8</td>
<td>163.7</td>
<td>60.5</td>
</tr>
<tr>
<td>24</td>
<td>2</td>
<td>✓</td>
<td>64.6</td>
<td>8</td>
<td>137.9</td>
<td>108.4</td>
</tr>
<tr>
<td>25</td>
<td>8</td>
<td>×</td>
<td>162.7</td>
<td>8</td>
<td>137.2</td>
<td>108.4</td>
</tr>
<tr>
<td>26</td>
<td>1</td>
<td>✓</td>
<td>0.9</td>
<td>8</td>
<td>137.2</td>
<td>108.2</td>
</tr>
<tr>
<td>27</td>
<td>8</td>
<td>×</td>
<td>278.2</td>
<td>8</td>
<td>162.8</td>
<td>47.7</td>
</tr>
<tr>
<td>28</td>
<td>1</td>
<td>✓</td>
<td>12.4</td>
<td>8</td>
<td>123.1</td>
<td>51.8</td>
</tr>
<tr>
<td>29</td>
<td>4</td>
<td>×</td>
<td>70.2</td>
<td>4</td>
<td>9.8</td>
<td>112.4</td>
</tr>
<tr>
<td>30</td>
<td>2</td>
<td>✓</td>
<td>62.6</td>
<td>8</td>
<td>108.5</td>
<td>108.7</td>
</tr>
<tr>
<td>31</td>
<td>3</td>
<td>✓</td>
<td>144.5</td>
<td>8</td>
<td>172.4</td>
<td>122.5</td>
</tr>
<tr>
<td>32</td>
<td>3</td>
<td>✓</td>
<td>157.2</td>
<td>-</td>
<td>TO</td>
<td>133.6</td>
</tr>
<tr>
<td>33</td>
<td>1</td>
<td>✓</td>
<td>1.1</td>
<td>4</td>
<td>0.6</td>
<td>133.6</td>
</tr>
<tr>
<td>34</td>
<td>4</td>
<td>✓</td>
<td>181.4</td>
<td>-</td>
<td>TO</td>
<td>605.4</td>
</tr>
<tr>
<td>35</td>
<td>1</td>
<td>✓</td>
<td>2.1</td>
<td>8</td>
<td>7.7</td>
<td>596.5</td>
</tr>
<tr>
<td>36</td>
<td>1</td>
<td>×</td>
<td>0.1</td>
<td>1</td>
<td>0.1</td>
<td>0.3</td>
</tr>
<tr>
<td>37</td>
<td>3</td>
<td>×</td>
<td>0.5</td>
<td>3</td>
<td>0.5</td>
<td>0.3</td>
</tr>
</tbody>
</table>
non-integral values.

In the decision procedure world, the tools Z3 and MATHSAT have support for floating-point arithmetic, primarily based on bit vector reasoning and bit-blasting. 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, or when the formula has non-linear arithmetic operations. An attempt was made to alleviate this problem by applying a combination of under- and over-approximations to the same formula [5].

Goubault and Putot [8] present abstract domains and methods to bound the difference between floating-point and real-arithmetic interpretations of the program, and these have been incorporated into FLUCTUAT [7], 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. This is reflected especially in the potential for spurious assignments.

Various formalizations and libraries for FPA have been developed in the domain of theorem proving [9]. More recently, these provers have been used to certify programs [3]. The use of such tools requires expert skills to provide hints to steer the theorem prover towards the goal. In contrast, model exploration approaches such as ours aim at principally push-button techniques.

### VIII. CONCLUSIONS

We have presented a framework for building solvers for floating-point decision problems, by reducing them to decisions in some proxy theory $T$. The assumptions are that (i) $T$ models are often close to FPA models, and (ii) $T$ formulas are on average easier to decide than FPA formulas. Examples of suitable proxy theories include reduced-precision FPA and real arithmetic. Previous work embeds such reductions into a CEGAR loop [12]. Our framework extends it by a numeric model refinement procedure, which tries to lift $T$ models to FPA models. The procedure determines, using a floating-point solver, how much certain variables need to be adjusted away from the $T$ model, to compensate for the difference between $T$ and FPA. We derive a new formula with a simpler structure and fewer free variables, and whose satisfiability immediately gives rise to an FPA model. Experimental results indicate our technique can find satisfying assignments efficiently.

### Future work

We plan to extend our work in two main directions. One is the use of approximate numeric techniques, rather than (exact) decision procedures, to solve formulas in the proxy theory $T$: thanks to model lifting, a precise solution in $T$ is not required for the first green box in Fig. 1. This relaxation opens up a host of other and potentially very scalable techniques especially for complex non-linear input constraints, including for the case of real arithmetic as proxy theory, for which we currently have limited support for non-linear formulas. The other direction is to improve our strategy for dealing with unsatisfiable formulas, rather than just “waiting” for the refinement to revert $T$ back to $f$; the latter causes all model finding efforts to be wasted.

### REFERENCES