HAL
archives-ouvertes.fr

Built-in Treatment of an Axiomatic Floating-Point Theory for SMT Solvers

Sylvain Conchon, Guillaume Melquiond, Cody Roux, Mohamed Iguernelala

▶ **To cite this version:**

Sylvain Conchon, Guillaume Melquiond, Cody Roux, Mohamed Iguernelala. Built-in Treatment of an Axiomatic Floating-Point Theory for SMT Solvers. 10th International Workshop on Satisfiability Modulo Theories, Jun 2012, Manchester, United Kingdom. pp.12-21. hal-01785166

HAL Id: hal-01785166
https://hal.inria.fr/hal-01785166

Submitted on 4 May 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Built-in Treatment of an Axiomatic Floating-Point Theory for SMT Solvers*

Sylvain Conchon
LRI, Université Paris Sud

Guillaume Melquiond
INRIA Saclay–Île-de-France

Cody Roux
LRI, Université Paris Sud

Mohamed Iguernelala
LRI, Université Paris Sud

Abstract

The treatment of the axiomatic theory of floating-point numbers is out of reach of current SMT solvers, especially when it comes to automatic reasoning on approximation errors. In this paper, we describe a dedicated procedure for such a theory, which provides an interface akin to the instantiation mechanism of an SMT solver. This procedure is based on the approach of the Gappa tool: it performs saturation of consequences of the axioms, in order to refine bounds on expressions. In addition to the original approach, bounds are further refined by a constraint solver for linear arithmetic. Combined with the natural support for equalities provided by SMT solvers, our approach improves the treatment of goals coming from deductive verification of numerical programs. We have implemented it in the Alt-Ergo SMT solver.

1 Introduction

On modern computers, floating-point arithmetic is by far the most common way of approximating real arithmetic and hence performing numerical computations. This comes from its inherent mathematical simplicity (very large range and guaranteed precision) combined with a strong will to ensure portability of this arithmetic across all architectures without forsaking performance [11]. These properties usually make this arithmetic an obvious choice for building numerical software. In many cases, for instance in avionics, critical systems make heavy use of floating-point computations.

But despite their simplicity, floating-point computations do not in general return the same result as ideal real computations. Indeed, while precision is guaranteed, accuracy is not: the negligible rounding errors caused by every operation might add up and cause the computed values to arbitrarily diverge from the expected values. In particular, floating-point arithmetic has some counter-intuitive properties when it comes to accuracy: the sum of two numbers that causes the biggest inaccuracy is the one that is the most precise (exact result). For example, take a and b to be the floating-point representations of 1.1 and 1, respectively. Then the floating-point number that is the result of $a - b$ can be computed with perfect accuracy: there is no loss of digits. However it is *not* the floating-point number that is closest to the "desired" result 0.1. This discrepancy between computed and expected results has been the source of numerous software and hardware failures [13].

The Gappa tool replicates the kind of reasoning one would use to prove correctness of state-of-the-art floating-point hardware and software. It automatizes the proofs in a highly efficient way, as long as the verification conditions only deal with arithmetic constructs [6]. Unfortunately, some program constructs tend to leak into the verification conditions and obfuscate the arithmetic constructs Gappa relies on. For instance, programs might be using arrays, have lots

*Work supported by the French ANR projects ANR-08-005 DeCert and ANR-11-INSE-03 Verasco.

of different execution paths; as a side-effect of their generation process, verification conditions might be littered with useless lemmas and equalities. All of these issues are the cause for extra work from the user and hence partly defeat the point of automatizing the process.

On the other hand, SMT solvers, which are ubiquitous in deductive program verification, are especially designed to handle these issues. They have built-in theories of arrays and congruences and specialized algorithms for instantiating lemmas, they depend on SAT solvers for propositional logic, and so on. Floating-point numbers are not presently handled by most SMT solvers, although there is a proposal to add them to the SMT-LIB 2 standard [14]. This would allow a complete description of the bit-level representation of IEEE-754 floating-point numbers and operations, including exceptional values. However error analysis is out of the scope of such a description, as there is no way to embed floating-point values into real numbers.

In this paper, we present a way to implement the proof mechanism of Gappa inside an SMT solver to allow reasoning on approximation errors in a theory of reals with floating-point operators. Section 2 describes the proof mechanism of the Gappa tool. Section 3 explains the limits of a purely axiomatic approach to integrating floating-point reasoning in an SMT solver. Section 4 presents a description of the integration of the procedure as a built-in instantiation based algorithm. Section 5 gives an example that uses the most salient features of the algorithm.

2 The Gappa Tool

Gappa is a tool dedicated to proving the logical formulas that usually occur when verifying the numerical correctness of a program [5]. These formulas are described by the following simple grammar:

$$\begin{array}{rcl} \mathit{prop} & ::= & \neg \mathit{prop} \mid \mathit{prop} \wedge \mathit{prop} \mid \mathit{prop} \vee \mathit{prop} \mid \mathit{prop} \Rightarrow \mathit{prop} \mid \mathit{atom} \\ \mathit{atom} & ::= & \mathit{expr} \leq \mathit{number} \mid \mathit{expr} \geq \mathit{number} \mid \mathit{expr} = \mathit{expr} \\ \mathit{expr} & ::= & \mathit{ident} \mid \mathit{number} \mid -\mathit{expr} \mid \mathrm{abs}(\mathit{expr}) \mid \mathrm{sqrt}(\mathit{expr}) \\ & & \mid \mathit{expr} \diamond \mathit{expr} \mid \mathrm{rnd}_{\dots}(\mathit{expr}) \end{array}$$

with $\diamond \in \{+, -, \times, /\}$. The semantics of these formulas is, for the most part, straightforward: identifiers are mapped to universally-quantified variables of real type, arithmetic operators and relations are mapped to the corresponding symbols of real arithmetic, and logical connectives are given their usual propositional meaning.

2.1 Rounding operators

Rounding functions are sufficient to express computations that occur inside a numerical program, thanks to the philosophy that underlies most computer arithmetic. For instance, consider the IEEE-754 standard for floating-point arithmetic [11] that describes the behavior of most floating-point units you will find in modern processors. It states: "a floating-point operator shall behave as if it was first computing the *infinitely-precise* value and then *rounding* it so that it fits in the destination floating-point format", assuming the inputs are not exceptional values. As a consequence, a floating-point addition between two floating-point numbers u and v should behave as if it was $\mathrm{rnd}_{\dots}(u + v)$ for some properly parametrized rounding operator.

For instance, if the destination format is *binary32* from IEEE-754 and if the rounding direction is the default one, then $\mathrm{rnd}(x)$ satisfies the following properties: it is a real number y such that there are two integers m and e with $y = m \cdot 2^e$, $|m| < 2^{24}$ and $e \geq -149$. Moreover, it

is the number that minimizes the distance $|y - x|$. Finally, if there are several such numbers,[1] then y is chosen so that it can be represented with a mantissa m that is an even integer while still preserving $e \geq -149$. Note that there is no *upper* bound on e as overflow is generally handled by a separate analysis. Floating-point arithmetic is standardized in such a way that it is always the case that there exists one and only one such number.

2.2 Proof Mechanism

When given a logical formula to prove, Gappa starts by performing some manipulations until it reaches an equivalent problem consisting of a disjunction of formulas of the following form:

$$t_1 \in I_1 \wedge \ldots \wedge t_n \in I_n \Rightarrow \bot.$$

Terms $t_1, \ldots, t_n$ are Gappa expressions, while $I_1, \ldots, I_n$ are intervals with numerical bounds. Actually, Gappa handles more predicates than just membership in an interval [6], but for the sake of simplicity, they will not be mentioned here.[2]

Gappa has a database of over 200 theorems which can be split into three categories. The first deals with real arithmetic, e.g. knowledge about square roots. The second category handles rounding operators and computer formats. The third category is composed of rewriting rules that simulate reasoning similar to that of forward error analysis.

Gappa applies these theorems on the hypotheses of the logical formula by a saturation mechanism, in order to deduce new facts. It keeps going until a contradiction is deduced or no new facts are found. Note that the tool is not guaranteed to terminate, and even if it does, an absence of contradictions does not mean that the property is true.

Theorems about real arithmetic and rounding operators have the following form:

$$\forall \vec{x},\ \forall I_1, \ldots, I_n, I, \quad f_1(\vec{x}) \in I_1 \wedge \ldots \wedge f_n(\vec{x}) \in I_n \wedge P(I_1, \ldots, I_n, I) \Rightarrow f(\vec{x}) \in I.$$

Expressions $f_1, \ldots, f_n, f$ are specific to a given theorem, and so is the relation P between intervals. Moreover, Gappa knows some partial function h such that $P(I_1, \ldots, I_n, h(I_1, \ldots, I_n))$. From the point of view of Gappa, even rewriting rules fit this model: $n = 1$, P is $\subseteq$, and h is the identity.

2.3 Theorem Instantiation

Gappa first performs symbolic backward reasoning for all the terms $t_1, \ldots, t_n$ that appear on the left-hand side of the original formula. The heuristic is that, if a contradiction is to be found, then it will be with one of the original bounds. It therefore searches which theorems have a conclusion that could be used to compute a bound for one of the original terms. The hypotheses of these theorems are enclosures too, so Gappa searches recursively on their terms too. The search proceeds in a manner similar to Prolog, as the theorems have the form of Horn clauses. The procedure stops when it finds that it needs to bound one of the terms of the original formula. In practice, this backward search terminates since theorems have been chosen so that the terms on the left-hand side are somehow smaller than the term on the right-hand side. This solves the issue of the set of terms being unbounded, but not the issue of finding instances for every $\vec{x}$, except that this time, the uninstantiated variables are on the left. Gappa relies on various heuristics to fill the holes.

[1] In other words, x is the midpoint between two consecutive floating-point numbers.

[2] The list of predicates and theorems that Gappa relies on are listed in its documentation: `http://gappa.gforge.inria.fr/doc/index.html`.

Once Gappa has performed this step of backward reasoning, it knows which terms and theorems are potentially useful. It can therefore perform the actual saturation, which is now just reduced to numerical computations on the bounds of the intervals.

3 Handling Gappa Axioms with SMT Solvers

SMT solvers are highly efficient tools for checking satisfiability of ground formulas from the combination of specific theories such as uninterpreted equality, linear arithmetic over rationals or integers, arrays and bit-vectors. SMT solvers may also handle quantified formulas[7, 2, 8, 4]. For that, they use heuristics to find good instances of universally-quantified lemmas present in the problem. These heuristics are usually guided by a set of instantiation patterns (also known as *triggers*) and a set of *known* terms, that is (ground) terms that appear in the facts assumed by the solver.

Knowing how SMT solvers handle quantified formulas, a tempting approach would be to describe the Gappa floating-point theory by a set of first-order formulas annotated with relevant triggers. For instance, axioms like

$$\forall x, y, z. \quad y \neq z \Rightarrow x \times (y - z) = x \times y - x \times z$$

can be given to an SMT solver with $x \times (y - z)$ as a trigger (if it allows defined functions to appear in triggers). Other axioms can also just be ignored because they are directly implemented by the decision procedures of SMT solvers. For instance, axioms like

$$\forall x, y. \quad x - y = (x - rnd(y)) + (rnd(y) - y)$$

are directly handled by the decision procedure for linear arithmetic. This category contains all the axioms for the free theory of equality, the linear theory of arithmetic, etc.

Unfortunately, this solution does not apply to all Gappa axioms. For example, in the following quantified formula

$$\forall i, p, e, x. \quad |x| \leq i \Rightarrow |\texttt{float}(p, e, \texttt{NearE}, x) - x| \leq 2^{\texttt{max}(e, (\texttt{ceil}(\texttt{log}_2(i)) - p))}$$

a relevant trigger would be the term $\texttt{float}(p, e, \texttt{NearE}, x)$. However, since it does not contain the bound variable i, the solver has to consider all reals as potential bounds for x, or it can restrict substitutions to *syntactic* bounds, *i.e.* bounds that occur in a syntactic ground predicate of the form $|\texttt{t}| \leq c$. The first solution is impracticable. The main problem with the second solution is that relevant bounds do not usually occur in terms, but come from some *deduction*. Moreover, an effective implementation of the floating-point theory of Gappa also requires from SMT solvers the ability to *compute* bounds like $2^{\texttt{max(b,(ceil(log}_2(i))-\texttt{a))}}$.

4 Gappa as a Matching Algorithm

The floating-point module we describe here is based on an interval analysis and a saturation approach nearly identical to that which is implemented in Gappa [5]. We interact with the SMT solver using the following interface. We take as input

- the set of terms considered by the solver that have as head symbol a symbol of arithmetic, such as $x + y$ or $\text{float}(t[i])$,

- the set of literals that involve arithmetic, such as $t \leq u$, or $t = u$ with t or u in the arithmetic theory,
- the module for matching modulo the equalities deduced by the solver.

The procedure returns in turn

- the deduced bounds on the considered terms,
- a set of equalities that are a consequence of these bounds,
- a set of contradictory literals, if a contradiction is deduced from its input.

We proceed as outlined in Section 2 and try to instantiate axioms of the database of axioms from the floating-point theory that are deemed relevant for finding interesting bounds on the considered terms. Note that the effectiveness of this approach is based in great part on the relevance of the axioms involved, which are based on the fine-tuned library of theorems found in Gappa. The lemmas generated by instantiation of the theorems are then applied to known bounds of terms until saturation, *i.e.* until there are no improvements on the bounds.

The interface is generic: it can be integrated into a framework with an existing procedure for arithmetic. This may in fact be quite useful, as the procedure is far from complete for arithmetic expressions involving non-linear constructions.

The floating-point module maintains internal information that is not directly expressible in most SMT solvers, such as the number of bits with which it is possible to represent a real number in a certain floating-point encoding. The set of lemmas will therefore be internal to the module, and computational *instances* are created when needed, by matching modulo equality on the considered terms. The procedure then checks which lemmas may actually be applied given the bound information on the terms, and applies these lemmas until no new consequences can be created.

A simplex-based algorithm [3] then finds the optimal bounds by linear combination of these. Improved bounds may then lead to another round of saturation by application of the instantiated axioms. In the case where no contradictions (empty bounds) are found, then equalities that were deduced by the simplex algorithm are sent on to the main loop of the SMT solver.

4.1 Description of the algorithm

The internal state of the module consists of the following data:

- the set of all instantiated theorems, which we call *lemmas*, with the set of terms on which they depend, as well as the terms in the conclusion,
- the set of all considered terms, along with the set of lemmas that allow a bound to be deduced for each term and the lemmas that depend on them,
- the bounds that have already been deduced by the theory.

The SAT solver sends literals to a theory solver that deduces initial bounds on the terms by linear combinations, and terms to the *lemma generation* which instantiates the axioms potentially necessary to infer bounds on terms and sub-terms that appear in the literals. The graph generated is in the form of that shown in Figure 2, where terms are linked with the lemmas that depend on them and lemmas are connected to the terms whose bounds they may

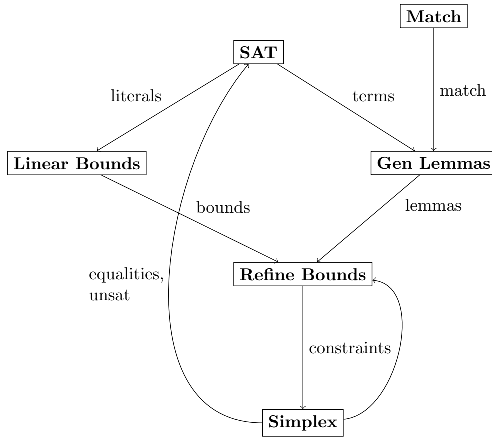

Figure 1: Description of the floating-point procedure.

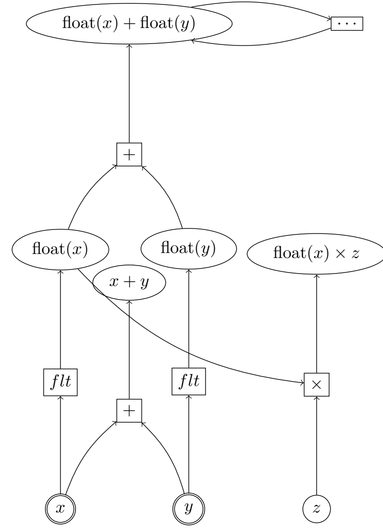

Figure 2: An example lemma dependency graph (with some cycle).

improve. The theorems are instantiated using the matching mechanism already present in the SMT solver. The description of the process is outlined in Figure 1.

Once we have computed the graph of lemmas that are potentially useful, we enter into the computational phase: we use the linear solver to infer initial bounds on some subset of the considered terms, and the lemmas are applied in succession, each new refined bound triggers the application of the lemmas that depend on it.

Once a fixed point is reached (no bounds are improved), we use the deduced bounds to build constraints which are sent to a simplex algorithm, which refines the bounds of every term. To limit the calls to the simplex algorithm, we only refine the bounds of terms on which some computational lemma applies, and on terms that are known to not be constants.

The alternation of calls between the simplex algorithm and the application of the lemmas continues until

- either a contradiction is found, in which case the incriminating literals are identified and sent to the SAT module, which then chooses the next literals to send to the theory,
- or no new bounds are computed yet no contradiction is found, and the equalities that can be deduced from the bounds are returned to the SAT solver.

The simplex algorithm is incremental. This increases the performance of the procedure in subsequent calls by a factor of 5 to 10. However backtracking is not possible in the current version of the algorithm, which forces the whole process to be restarted at each call to the global decision procedure.

At the end of the saturation process, an equality is deduced by examining the bounds which are trivial, *i.e.* $t \in [i, i]$ in which case we deduce $t = i$. The use of the simplex algorithm ensures that all such equalities that are linear consequences of the deduced constraints will be found. Since the module considers terms that may not be terms (or sub-terms) that appear in the original problem, it may choose not to propagate such equalities to the other theories.

Correctness of the algorithm depends on the correctness of each process. The crucial points are the validity of the theorems that are instantiated, the correctness of the simplex algorithm, and the fact that the literals identified in the unsatisfiability witnesses are a super-set of some minimal set of literals that lead to a contradiction.

We take pains to apply only lemmas that are true of both the ring of real numbers and that of integers, with the computed bounds being represented by rational numbers. Of course there is no floating-point operator on integers, but certain operators are overloaded, like the arithmetic operators and absolute values. This means that the floating-point "box" is generic: the bounds deduced by it are valid in the case where terms denote integers, and can be subsequently refined by integer-specific methods; typically by performing case analysis on the bounds.

Our system is not complete, as we are in a very expressive fragment of arithmetic, that includes integer arithmetic. Additionally, even the restriction of our theory to real arithmetic with a floating-point operator is likely to be either undecidable, or with a decision procedure of prohibitive complexity. We have tried to find the optimal trade-off between speed and expressiveness, with emphasis on proving error bounds.

Termination is also not guaranteed, as it is easy to build a set of terms that satisfy the following relations:

$$x \in [-1, 1]$$

$$x = f(x)$$

$$\forall y, f(y) \leq k \cdot y, \quad k \in]0, 1[$$

In which case each pass of the algorithm will scale the bounds for x by a factor k, without ever converging to the fixed point $x \in [0, 0]$.

We adopt the simple solution of stopping the iteration after a fixed number of cycles.

5 Case Study

We give an example application of our development by automatically deriving an error bound on a simple algorithm that computes the sum of the 10 elements of an array with values initialized to the floating-point approximation of 0.1. We give this code in Figure 3 using the Why3 specification and programming language [9], which allows us to write imperative code with Hoare-logic style annotations.

The Why3 framework generates a number of proof obligations in the Alt-Ergo format, split into safety properties and properties given by the assertions.

The `rnd` function performs the *binary32* nearest-ties-to-even rounding of a real number, if this operation does not result in an overflow. The `add` function performs the addition as it is defined on floating-point numbers *i.e.* as the rounding of the infinite-precision addition on the floats seen as real numbers. Again, this function takes as precondition that the addition does not result in an overflow.

The invariant specifies that the partial sums at step $i+1$ are at distance no more than $\frac{i}{1000}$ from $\frac{i}{10}$, which allows us to easily conclude that the sum is at distance less than $1/100$ from 1 at the end of the loop.

Our extension is able to prove the 10 generated properties, three of which are not proven by Alt-Ergo or Gappa as the former does not have knowledge of bounds on floating-point errors, and Gappa cannot perform reasoning on equality or general linear arithmetic. This stresses the necessity of combining the two solvers. The hardest goal generated by Why3 is given in

```
module Sum

let rnd x = { no_overflow NearestTiesToEven x }
  round NearestTiesToEven x
  {result = round NearestTiesToEven x}
let add x y =
  { no_overflow NearestTiesToEven (x +. y) }
    round NearestTiesToEven (x +. y)
  { result = round NearestTiesToEven (x +. y)}


let sum () =
  {}
  let a = make 10 (rnd 0.1) in
  let s = ref (rnd 0.) in
  for i = 0 to 9 do
    invariant
      {
        abs(!s -.  0.1 *. from_int i) <=. 0.001 *. from_int i
      }
      s := add !s a[i]
  done;
  !s
  {  abs (result -. 1.) <=. 0.01 }

end
```

Figure 3: A simple imperative program using floats.

```
goal WP_parameter_sum :
  abs(float(53,1074,ne,0.1)) <= 0x1.FFFFFFFFFFFFFp1023 -> 0 <= 10 ->
  abs(float(53,1074,ne,0.)) <= 0x1.FFFFFFFFFFFFFp1023 -> 0 <= 9 ->
  forall s:real. forall i:int. (0 <= i and i <= 9) -> i <= 10 and
  abs(s - (0.1 * real_of_int(i))) <= 0.001 * real_of_int(i)) ->
  (0 <= i and i <  10) and
  abs(float(53,1074,ne,s + const(float(53,1074,ne,0.1))[i]))
  <= 0x1.FFFFFFFFFFFFFp1023 and
  forall s1:real.
  s1 = float(53,1074,ne,s + const(float(53,1074,ne,0.1))[i]) ->
  i + 1 <= 10 and
  abs(s1 - (0.1 * real_of_int(i + 1))) <= 0.001 * real_of_int(i + 1)
```

Figure 4: Loop invariant preservation.

Figure 4; it specifies that the invariant is preserved by the operations performed in the loop, with additional non-overflow conditions.

The expression `float(p,e,m,x)` denotes the floating-point approximation to x with round-

ing mode m (`ne` is nearest float with even mantissa in case of tie), a mantissa of at most p bits, and an exponent greater than $-e$. The expression `real_of_int(x)` is simply a cast from the type of integers to that of real numbers.

Our extension of Alt-Ergo handles this goal in 1.4 seconds (Intel Core 2.66 GHz, 2 GB RAM) by application of the error-bound theorems and the linear arithmetic solver. First note that `real_of_int` is a congruence, which allows $\frac{1}{1000} \cdot \text{real_of_int}(i+1)$ to be treated as $\frac{\text{real_of_int}(i)}{1000} + \frac{1}{1000}$. We split on the sign of terms t such that $|t|$ appears in the goal. If $t \geq 0$, we add the equality $t = |t|$ to the set of literals and $t = -|t|$ otherwise. Then the equalities $\text{const}(t)[i] = i$ are instantiated for each appropriate term.

Finally linear arithmetic allows us to bound the terms t such that $\text{float}(p, e, d, t)$ appears in the goal: 0.1 is bounded by definition, and $s + \text{float}(53, 1074, ne, 0.1)$ can be derived from the bound on i and the inequality $s - (0.1 \times i) \leq 0.001 \times i$.

The theorem on absolute errors can then be instantiated by the floating-point module to generate a bound of the form:

$$\text{float}(53, 1074, ne, s + \text{float}(53, 1074, ne, 0.1)) - (s + \text{float}(53, 1074, ne, 0.1)) \in [-\varepsilon, \varepsilon]$$

A last step of linear arithmetic is required to conclude.

A more realistic example entails proving the safety properties of a C program involving floating-point computations which was originally part of an avionics software system. Safety properties are generated by the Frama-C/Why3 tool-chain, and involve properties about integer and floating-point overflow, correctness of memory access, etc. Here also the Alt-Ergo extension manages to prove all 36 goals, 10 of which involve floating-point properties (Alt-Ergo proves 2 of those goals). However the added expressiveness comes at the cost of speed: on average Alt-Ergo with the floating-point module is 5 times slower than Alt-Ergo alone, on goals involving real arithmetic.

Gappa is able to prove the overflow properties, but at the cost of manually performing theorem instantiation, which Alt-Ergo handles on its own.

6 Conclusion and perspectives

We have described an implementation of a theory of real numbers with floating-point operators by a dedicated matching procedure, which can be plugged into an SMT solver. Given terms and initial constraints, the procedure performs matching modulo ground equalities, saturation and calls to a linear solver to deduce bound information on terms.

This framework is standalone. It can be integrated as-is, notwithstanding slight changes to the usual interface of the matching module: literals assumed by the solver need to be sent instead of just terms. A tighter integration would re-use the procedure for deciding linear arithmetic already present in the SMT solver, provided that it could be queried for the best possible bounds of a term.

We have implemented this procedure in the Alt-Ergo SMT solver, replacing the part of the arithmetic module that deals with inequalities, as it became redundant with our work.

The use of the simplex algorithm, while powerful, is also the source of the greatest performance penalty: it may be called many thousands of times on large goals.

The theory of floats we describe here does not consider NaN or infinite values, unlike to the proposed addition to the SMT-LIB. However Ayad and Marché show [1] that it is possible to encode such floating-point data by a combination of algebraic data-types and cases on one hand, and an unbounded model of floating-point numbers as used in Gappa and in the present

work on the other hand. One can then observe that the "exceptional" cases in which NaN or $\pm\infty$ appear are handled quite easily by the traditional theories in the SMT solver, and our addition is designed to solve the other cases.

Future Work

Clearly, our first goal is to improve performance of the procedure on large problems, allowing us to treat a wider range of examples from industrial software verification. Moreover we have not integrated all the theorems from Gappa, just those which were required for most immediate use. In particular, we only treat *absolute error* and not yet *relative error*.

Similar work has been brought to our attention by the reviewers [12, 10]. In this work, the authors introduce a theory capable of handling general constraints on real numbers with floating-point constraints, as we are. The constraints are handled by their LRA+ICP algorithm (Linear Real Arithmetic and Interval Constraint Propagation) in a manner related to ours, though they do not have as general floating-point operators, nor do they have rewriting-based heuristics for transforming the non-linear constraints. They furthermore propose a mechanism based on successive interval refinements to avoid the cost of successive calls to a linear optimization algorithm. Further work is needed to understand whether such an approach is feasible in our framework.

References

[1] A. Ayad and C. Marché. Multi-prover verification of floating-point programs. In *IJCAR 5*, volume 6173 of *LNAI*, pages 127–141, Edinburgh, Scotland, July 2010.

[2] C. Barrett and C. Tinelli. CVC3. In *CAV*, pages 298–302, 2007.

[3] F. Bobot, S. Conchon, E. Contejean, M. Iguernelala, A. Mahboubi, A. Mebsout, and G. Melquiond. A simplex-based extension of Fourier-Motzkin for solving linear integer arithmetic. In *IJCAR 6*, volume 7364 of *LNAI*, Manchester, UK, 2012.

[4] S. Conchon and E. Contejean. Alt-Ergo. Available at `http://alt-ergo.lri.fr`, 2006.

[5] M. Daumas and G. Melquiond. Certification of bounds on expressions involving rounded operators. *Transactions on Mathematical Software*, 37(1):1–20, 2010.

[6] F. de Dinechin, C. Lauter, and G. Melquiond. Certifying the floating-point implementation of an elementary function using Gappa. *Transactions on Computers*, 60(2):242–253, 2011.

[7] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[8] B. Dutertre and L. de Moura. Yices. Available at `http://yices.csl.sri.com/tool-paper.pdf`, 2006.

[9] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *CAV 19*, volume 4590 of *LNCS*, pages 173–177, Berlin, Germany, 2007.

[10] S. Gao, M. K. Ganai, F. Ivancic, A. Gupta, S. Sankaranarayanan, and E. M. Clarke. Integrating ICP and LRA solvers for deciding nonlinear real arithmetic problems. In *FMCAD*, pages 81–89, Lugano, Switzerland, 2010.

[11] IEEE Computer Society. *IEEE Standard for Floating-Point Arithmetic*. 2008.

[12] F. Ivancic, M. K. Ganai, S. Sankaranarayanan, and A. Gupta. Numerical stability analysis of floating-point computations using software model checking. In *MEMOCODE*, pages 49–58, Grenoble, France, 2010.

[13] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser, 2010.

[14] P. Rümmer and T. Wahl. An SMT-LIB theory of binary floating-point arithmetic. In *SMT 8 at FLoC*, Edinburgh, Scotland, 2010.