Real Behavior of Floating Point Numbers *

Bruno Marre[1], François Bobot[1], and Zakaria Chihani[1]

CEA LIST, Software Security Lab, Gif-sur-Yvette, France
(first.last@cea.fr)

Abstract

We present an efficient constraint programming (CP) approach to the SMTLIB theory of quantifier-free floating-point arithmetic (QF_FP). We rely on dense interreduction between many domain representations to greatly reduce the search space. We compare our tool to current state-of-the-art SMT solvers and show that it is consistently better on large problems involving non-linear arithmetic operations (for which bit-blasting techniques tend to scale badly). Our results emphasize the importance of the conservation of the high-level structure of the original problems.

1 Introduction

Context. A few decades ago, formal program verification was born amidst vivid criticism and mitigated support. Even brilliant minds deemed it "bound to fail" [23], concluding their controversial paper saying that if, despite all their reasons, "verification still seems an avenue worth exploring, so be it". And so it was. Today, software verification is a well established field of research, and industrial adoption has been achieved in some key areas, such as safety-critical and embedded systems.

Software verification offers a means to ensure correctness of a program with respect to some formal specification by delegating proof obligations to solvers and theorem provers. The main challenge of this method is to give to the solvers the ability to reason about basic datatypes encountered in everyday programs such as strings, integers, bit-vectors, floating-point numbers, *etc.*

Floating-point arithmetic. Machine memory being finite, floating-point numbers tackle a difficult problem: representing real numbers. It is an old invention that predates computers (used in 1914), standardized in early 1980 as the IEEE-754 [18]. Despite their ubiquity, reasoning about floating-point numbers (FPNums) is tricky due to the regular approximations usually involved with floating-point operations (FPOp).

Consider, for example, that $1. +. 2^{100} = 2^{100}$, indeed in double precision format the floating-point representation (FPRep) of 2^{100} absorbs all values between -2^{46} and 2^{47}. Furthermore, we have the following counter-intuitive truth: $\overbrace{0.1 +. \ldots +. 0.1}^{10} \neq 0.1 \times. 10. = 1.$ because 0.1 is not representable as a binary FPNum. However the semantics of FPOp, given by IEEE-754, are clear: the result of a FPOp is the result of the rounding o of the real operator result, *i.e.,* $x +. y = \mathrm{o}(x + y)$ [24, Section 2.2.1]. Unless otherwise specified, all the results presented in this paper are valid for each individual rounding mode, therefore we omit them as a parameter of the o symbol. For concision, we also write the floating-points (FP) operators with their mathematical notation subscripted with f (*i.e.,* $<_f$ for `fp.leq`, $\geq_f$ for `fp.geq`, *etc.*).

*Work partially funded by ANR-14-CE28-0020 grant. The CP solver COLIBRI is generously sponsored by IRSN, the French Institute for Radioprotection and Nuclear Safety.

Methodology. Several approaches, discussed in the following related work section, are used to deal with floating-point numbers and arithmetic, ranging from satisfiability modulo theory (SMT) to constraint programming (CP) efforts, and include interactive theorem provers. This paper, through comparison with state-of-the-art solvers, will show the advantages of our approach, specifically in terms of scalability and theory combination.

Proposal and contributions. We tackle the problem of solving quantifier-free floating-point arithmetic problems, also called QF_FPA or QF_FP [8]. Henceforth, we use the latter. In particular, we now list the novelties of our approach before detailing them in the following sections:

- The CP approach allows *attributing* to a variable several *domain representations* (integer intervals, known bits, FPNums interval). The domain of a variable is an over-approximation of the set of values the variable could take. We use *union of intervals*, presented below, to maintain a more extensive disjunctive knowledge and additional ways to restrict the search for solution.
- A relational attribute is associated to our variables through difference logic [16] (also known as Difference-Bound Matrices, or DBM in the SMT community), where a *FP tailor-made distance* between variables is recorded.
- The monotonicity property of rounding allows to propagate information from one edge to another of then the difference logic global constraints.
- A link with a bit-vector domain (see our previous work on CP(BV) [10] allows us to treat type casts and reinterpretation between FP and BV. Furthermore we handle casts between FP and Real (cf. fig. 1a).
- Preserving the original high-level structure (no blasting of FP variable), we intertwine constraint propagation with simplifications and factorizations that rely on identified algebraic properties of FPNums. This is in contrast to most SMT solvers where these simplifications are limited to a preprocessing step.
- Finally, we implement all these novelties in the solver COLIBRI[1]. Note that the above reasonings are implemented mainly for nearest even rounding, the other roundings currently feature only domain propagations. Moreover only simple and double precision FPNums format are supported.

After looking at the related work and background, the paper presents the different domains used in COLIBRI to handle QF_FP, then the propagations applied on each of them, follows the communications between domains, and finally discusses some aspects of the search.

2 Related Work

Several approaches are explored when it comes to FP. In the SMT community, few solvers [15, 11] can handle QF_FP. A common approach in the SMT solvers is to use a flattening called bit-blasting [4]. This Boolean encoding allows to delegate to powerful SAT solvers but looses high-level structure of the original problems and scales extremely poorly for some common

[1] freely available for test, evaluation, educational purposes or internal research purposes only at http://soprano-project.fr/software.html

problems involving non-linear operators over 64-bits FP. Our experimentation section shows interesting results compared to bit-blasting approaches on QF_FP.

We also mention the interactive prover Gappa, which is able to verify preservation of numerical properties, relating FP and real numbers. The Gappa tool reasons on real numbers operations then explicitly applies rounding (*i.e.,* FP operators are interpreted as the rounding of real number operators). However, `NaN` and Infinities are not supported by Gappa. They are supported in our tool.

Alt-Ergo, a non-bit-blasting SMT solver, is based on Congruence Closure parametrized by a Shostak theory X. For FP it uses built-in support for rounding operators on constants and axioms with semantic triggers for the general case [13, 12]. This tool combines the force of the Gappa tool (saturation of consequences of the axioms) and a constraint solver for linear arithmetic. Like Gappa, this tool lacks built-in support for `NaN` and Infinities, which prevents it from treating most of the SMT competition benchmarks.

But perhaps the closest effort to our paper is the CP solver FPCS based on exact projection functions for FP constraints [22]. It performs powerful linearization techniques[3]. The FPCS solver is a strict subset of ours: it does not perform simplification and factorization, it has only one domain (FP) whereas our solver relies on interreduction between BV, Real, Int, congruence and global difference constraints.

The difference between propagation in the CP community and learning in the SMT community is known. Although our experimentations seem to cautiously indicate that, in the case of QF_FP, propagation and interreduction on high-level FP domain is more efficient than learning on low-level bit-blasted FP, the disparity, in our opinion, is due more to the high/low level opposition than to the propagation/learning opposition. Several CP related efforts are investigating learning possibilities, though not in FP, to our knowledge. We cite the encouraging efforts of Lazy Clause Generation [25] and Learning General Constraints in CSP [27]. We also believe that the abstract CDCL learning, on the SMT side[9], can spawn possible similarly inspired methods on the CP side.

3 Background

This section exhibits the ground on which we carried out the research we discuss in this paper, and introduces QF_FP.

COLIBRI Initially developed, in ECLiPSe Prolog[2], to assist CEA[3] verification tools [5, 28, 2, 14], the COLIBRI CP solver [20] supports bounded integers (both standard and modular arithmetic [17]), reals, global difference constraints [16], and more recently bit-vectors [10]. Since its inception, it enjoyed some measure of treatment for FP [21]. COLIBRI handles communications between all these theories: Real, Int, BV, FP. This paper focuses on the novel improvements, both theoretical and practical, that allowed this tool to surpass many state-of-the-art solvers.

QF_FP theory. We consider in this paper the semantics of floating-point arithmetic as defined by the IEEE754-2008 standard and as supported by the SMT-LIB2.5 standard [26, 8]. In addition to FPOps, the standard also defines type casts, rounding, and undefined behaviours. The set of FPNums without `NaN` is denoted by $\mathcal{F}$.

[2] http://eclipseclp.org/
[3] French Atomic Energy Commission

4 CP for FP arithmetic

4.1 Domain representation

Handling `NaNs`. One of the many definitions fixed by the IEEE 754 standard [18] is the constant `NaN`, indicating an unrepresentable value (a division of zero by zero, square roots of negative numbers, *etc.*). It is generally not desirable to generate these values, especially in the context of critical systems. A solver must, as COLIBRI does, not only propagate `NaNs` throughout any arithmetic or comparison operator but also manage the contexts where a `NaN` *could* be generated. In the particular case of `fp.max` and `fp.min` if an argument is known to be `NaN` the result is unified with the other argument. In particular, a `NaN` falsifies any comparison operator (a consequence is that FP equality is not reflexive, because `not (fp.eq NaN NaN)`). Conversely, if a comparison is known to hold, its operands cannot be `NaNs`.

A FP variable is associated with a Boolean flag that is set if the variable is considered `NaN`. Until `NaN` is an impossible value for all the arguments and the result of a constraint, no other propagation is done on it; it is blocked.

Handling partial functions The cast functions from FPNums to integers and reals have no meaning for $\{\texttt{NaN}, +\infty, -\infty\}$. Since the logic used only handles total functions, the SMT-LIB accepts any interpretation for these undefined terms that are considered as uninterpreted. This does not mean that each prover can choose a different value for `fp.to_ubv NaN`, but that the problem is satisfiable if it is true for some valuation of the variable and a choice of interpretation for these undefined terms. In COLIBRI the possibility of these undefined terms are handled by first waiting for the argument to be in the domain of definition through propagation, labelling or domain splitting of the argument before doing any propagation of the constraint. During this period, functions applied to the same arguments are factorized, *i.e.,* congruence closure is applied.

In all the remainder of the paper, we suppose that no `NaN` could appear since we discuss the propagation only on unblocked constraints.

Integral status. For concision let `rti` be a shorthand for `fp_roundToIntegral`. A FP number f is *integral* when f is finite and it satisfies $\texttt{rti}(RTN, f) = \texttt{rti}(RTP, f)$, *i.e.,* it can be interpreted as an integer. Let `isintegral` be a predicate that checks if a FPNum is integral. Constraints involving only FPNums with integral status can sometimes rely on all the integer-related machinery involved in the solving process of COLIBRI, and their result is also integral. This delegation is made possible by the open-box dense communication offered by the CP framework. A FP variable is thus associated with a flag that is set if the variable is known to be integral. If casts are not available, then it would still be useful for program specifications to let users specify that a FPNum is integral.

Union of intervals. The classical representation of the domain of FP variable f is through a pair of two FPNums a, b s.t. $a \leq f \leq b$. Since the FP domain is discrete, only closed intervals are needed. However, FP programs often use tests of the form `abs(f) >` ϵ instead of comparing f to 0.0. This is because a test that is true using ϵ greater than the computation error would also be true for the real comparison. Moreover many FP program specifications are expressed as a bound between the computation with FP and the computation with real.

Considering the above cases, one understands the importance of allowing gaps in the intervals, or equivalently, to use a disjoint union $\uplus$ of intervals. This domain can represent

$-\infty$,$+\infty$,$-0.$,$+0.$ (unless, of course, the `NaN` flag is set). The backward and forward propagations on simple intervals naturally extend to unions of intervals by propagating pairwise on the intervals. For example with $x \in [1.;4.] \uplus [5.;7.]$, $y \in [2.;3.] \uplus [10.;11.]$, $x + y = z$ the propagation computes that $z \in [3.;7.] \cup [7.;10.] \cup [11.;15.] \cup [15.;18.] = [3.;10.] \uplus [11.;18.]$. If one knows an order on the variable, for example $x \leq y$, some pairs of intervals can be filtered or refined. For example when $x \leq y$, it is impossible to have $x \in [5.;7.]$ and $y \in [2.;3.]$ simultaneously. The same goes for $x \in [3.^+;4.]$ and $y \in [2.;3.]$ ($3.^+$ is the successor of 3.). So $z \in [3.;6.] \cup [11.;15.] \cup [15.;18.] = [3.;6.] \uplus [11.;18.]$.

Difference constraints The difference constraints keep for each variable the distance with other variables using an interval. We call them *deltas* henceforth. The distance could be expressed as the distance in rationals but usual representation of rationals are not efficient (big integers for numerator and denominator, for example the FPNum 2^{1024} contains lot of zero words). Contrary to rationals there are a finite number of FPNums. In fact for positive FPNums they can be associated with integer using the reinterpretation (not cast) of FPNum to bitvectors. For negative FPNum it is not direct because they are not 2's complement and we identify -0 and $+0$ with a distance of 0.

Formally for $x \in \mathcal{F}$, $\mathrm{num}(x)$ is defined by

$$\mathrm{num}(x) = \begin{cases} -|\{z \in \mathcal{F} | x \leq_f z <_f 0\}| & \text{when } x < -0 \\ |\{z \in \mathcal{F} | 0 <_f z \leq_f x\}| & \text{when } +0 < x \\ 0 & \text{when } x \in \{-0, +0\} \end{cases}$$

where $|\cdot|$ is the cardinality function. The floating-point distance $d^{\mathcal{F}}$ is then defined simply by

$$d^{\mathcal{F}}(x, y) = \mathrm{num}(y) - \mathrm{num}(x).$$

This distance is a good representation of multiplication by two of a normal number without overflow, this is just an increment of the exponent. In that case the distance is exactly the size of a binade (a set of FPNums with the same exponent). Note the linear approximation used by COLIBRI (section 4.2.3) uses the other distance in rational, which is better for representing additions with constants. Thus the deltas and the linear approximation will benefit from each other.

A distance graph is also used for reals and bitvectors.

4.2 Propagations

We first focus on the propagation inside each domain. Starting with direct propagation in the interval domain, then a specific handling of difference logic for FPNums, finally a relaxation method to use a rational simplex for FPNums constraints.

4.2.1 Domain propagation

Our AdaCore partner provided numerous floating-point examples coming from problems found by their customers. These examples are code fragments that they were unable to prove with their current automatic provers, which compelled them to add those fragments as user defined axioms. We show here the mathematical version of these Ada programs.

$$(-0 \leq_f x \leq_f 16777216.0) \wedge (-0 \leq_f y \leq_f 16777216.0) \wedge (-0 \leq_f z) \Rightarrow -\mathtt{fp.mul}(x, y) \leq_f z$$

Domain propagation is at the heart of constraint programming, on which COLIBRI is based. It consists in associating domains to each term, here a disjoint union of floating-point intervals and a Boolean domain, and to each function f a design function that continuously improves domains of x, y, and $f(x,y)$ using the domain of the other terms.

In this example, initially, the domains of Boolean terms are in $\{\bot;\top\}$ and the domains of all the floating-point terms are $[-\infty;+\infty]$, COLIBRI handles infinite and finite FPNums (32 bits and 64 bits with all rounding modes. With the exception of nearest-to-even rounding, reasoning with other roundings is rather minimal in COLIBRI). The domain of the conclusion $-\texttt{fp.mul}(x,y) \leq_f z$ is set to $\{\bot\}$ and COLIBRI looks for a counter-example. If none exists, then the property is valid.

The hypothesis restrains the domains of the variables to $x, y \in [-0; 16777216.0]$, $z \in [-0;+\infty]$. The propagators then improve successively the following domains:

$$\begin{aligned}\texttt{fp.mul}(x,y) &\in [+0; 281474976710656.]\\ -\texttt{fp.mul}(x,y) &\in [-281474976710656.; -0]\\ (-\texttt{fp.mul}(x,y) \leq_f z) &\in \{\top\}.\end{aligned}$$

The last line contradicts the initial setting to $\bot$ of the relation, so the domain of the relation is empty. An empty domain means a conflict, which excludes the existence of a counter-example: the property is proved.

The propagators used in this example are called *forward* propagators since they improve the knowledge on the result using the knowledge on the arguments. These propagators use the property of part-wise monotonicity of the functions.

If the propagators are executed in another order (mostly non-deterministic), *backward* propagators will be used. Let s be the next FPNum after +0, *i.e.,* the smallest positive subnormal number:

$$\begin{aligned}(-\texttt{fp.mul}(x,y) \leq_f z) &\in \{\bot\}\\ -\texttt{fp.mul}(x,y) &\in [s;+\infty]\\ \texttt{fp.mul}(x,y) &\in [-\infty;-s]\\ \texttt{fp.mul}(x,y) &\in ([-\infty;-s] \cap [+0; 281474976710656.]).\end{aligned}$$

This also leads to an empty domain.

Albeit simple, these propagators make it possible to prove an important part of the proof obligations that were not provable with an axiomatic approach, because they require the ability to compute operators on floating-point constants.

COLIBRI uses other propagators [21, 1] that rely on properties other than monotonicity. Contrary to rationals, the result of a floating-point subtraction strongly limits the ranges of its arguments. For example, starting with $x, y \in [+0; 1000]$ and the constraint $\texttt{fp.sub}(x,y) = \mathrm{o}(0.1) \simeq 0.10000000149$ (in simple format), the above propagators applied once do not reduce the domains significantly. The intervals would become $x \in [\mathrm{o}(0.1); 1000]$, $y \in [0; 1000]$. With the new propagations [21, 1] however, the intervals get restricted to $x \in [\mathrm{o}(0.1); \mathrm{o}(0.225)]$ and $y \in [+0; \mathrm{o}(0.12499999)]$.

COLIBRI has also other, more relational, propagators. For example, from the knowledge that an argument is equal to the result, if $x, y \in [0; 1024]$ and $y = x + y$, then it can infer $x \in [0; 2^{-14}]$ because x must be small enough in order to be absorbed by y.

These propagations are local in that they involve only one constraint.

4.2.2 Difference constraints

Populating the deltas often exhibits early on unsatisfiabilities that would be otherwise detected late, loosing efficiency. Such detections can take place through monotonicity, type cast and rounding. We discuss this in the following.

Through monotonicity. There are multiple lemmas that are widely known to computer arithmeticians, but strangely scarcely used in automated floating-point reasoning. We rely on the following such lemmas.

Lemma 4.1.

$$\forall x, y, z \in \mathcal{F}, 0 <_f y \Rightarrow \texttt{fp.div}(x, y) <_f z \Rightarrow x \leq_f \texttt{fp.mul}(z, y)$$

Thus we can use the following implications:

$$\texttt{fp.add}(A, B) >_f C \;\Rightarrow\; A \geq_f \texttt{fp.sub}(C, B) \qquad \texttt{fp.add}(A, B) <_f C \;\Rightarrow\; A \leq_f \texttt{fp.sub}(C, B)$$
$$\texttt{fp.sqrt}(A) >_f C \;\Rightarrow\; A \geq_f \texttt{fp.mul}(C, C) \qquad \texttt{fp.sqrt}(A) <_f C \;\Rightarrow\; A \leq_f \texttt{fp.mul}(C, C)$$

which we justify as follows in the finite case (the non finite case where `NaN` cannot appear is trivial). Consider that, according to the IEEE standard, $\texttt{fp.add}(A, B) = \mathrm{o}(A + B)$. Further consider that through monotonicity of the rounding o, $\mathrm{o}(A + B) > C$ implies, $A + B > C$, leading us to conclude $A > C - B$ which, by rounding, gives $A \geq \mathrm{o}(C - B)$. The latter corresponds to the non-strict inequality $A \geq_f \texttt{fp.sub}(C, B)$. For example, COLIBRI uses this results if the deltas also know that $A <_f \texttt{fp.sub}(C, B)$, it can immediately detect unsatisfiability, and if it knows that $A \leq_f \texttt{fp.sub}(C, B)$, it can deduce that $A = \texttt{fp.sub}(C, B)$.

By exploiting monotonicity, other inequalities can be deduced:

$$X >_f Y \Rightarrow \begin{cases} \texttt{fp.add}(A, X) \geq_f \texttt{fp.add}(A, Y) \\ \texttt{fp.sub}(A, X) \leq_f \texttt{fp.sub}(A, Y) \\ \texttt{fp.sqrt}(X) \geq_f \texttt{fp.sqrt}(Y) \end{cases}$$

Similar implications can be valid for the operators `fp.mul` and `fp.div` depending on the sign of their operands and whether they are decreasing or increasing (absolute value is greater or less than 1) to orient the resulting non-strict equality.

Notice that when the operations are exact strictness of inequalities is preserved. For example, proving the unsatisfiability of $X \leq_f Y \wedge Y >_f 0.0 \wedge \texttt{fp.div}(X, Y) >_f 1.0$ when X, Y and $\texttt{fp.div}(X, Y)$ are finite can be reached through the following steps: deduce $\mathrm{o}(\frac{X}{Y}) > \mathrm{o}(1.0)$ since $1.0 \in \mathcal{F}$, then $\frac{X}{Y} > 1.0$ and $X > Y$ since $Y > 0$, which contradicts $X \leq Y$.

Lemma 4.2. *Let $D, E \subset \mathcal{R}$, $f : D \mapsto E$ and $f^{-1} : E \mapsto D$ such that:*
$\forall x : D$, $f^{-1}(f(x)) = x$ and f is monotonically increasing. Then we have

- $\forall x \in D$, $\mathrm{o}(y) \in E$, $\mathrm{o}(f(x)) < \mathrm{o}(y) \Rightarrow \mathrm{o}(x) \leq \mathrm{o}(f^{-1}(\mathrm{o}(y)))$,
- $\forall x \in D$, $y \in E$, $\mathrm{o}(f(x)) < \mathrm{o}(f(y)) \Rightarrow x < y$.

Lemma 4.3. *Let $x, y, z \in \mathcal{R}$ and n a natural number, if $0 < y$, and $2^n < \mathrm{o}\left(\frac{\mathrm{o}(x)}{\mathrm{o}(y)}\right)$ we have $\mathrm{o}(2^n \times y) < \mathrm{o}(x)$*

For example, through lemma 4.3 then lemma 4.2 with the function $t \mapsto x - t$, we can prove :

$$(z \geq 0.0) \wedge (x \geq y) \wedge (y \geq z) \wedge (x > z) \wedge (a \geq 1.0) \Rightarrow \mathrm{o}\left(\frac{\mathrm{o}(x-y)}{\mathrm{o}(x-z)}\right) \leq a$$

Lemma 4.4. *Let $x \in \mathcal{F}$, such that* $\texttt{fp.mul}(x,x)$ *is a normal number, then the floating-point square root is the inverse of the square function:* $\texttt{fp.sqrt}\,(\texttt{fp.mul}(x,x)) = \texttt{fp.abs}(x)$ *[7].*

The next example of AdaCore (`User_Rule_16`) is interesting because it is false and involves the square root function. The formula to prove unsatisfiable is

$$(x \in [-7800; +7800]) \wedge (y \in [-7800; +7800]) \wedge (x > |y|) \wedge \mathrm{o}\left(\sqrt{\mathrm{o}\left(\mathrm{o}(x^2) - \mathrm{o}(y^2)\right)}\right) > x$$

Using lemma 4.2, we have the relation $\mathrm{o}\left(\mathrm{o}(x^2) - \mathrm{o}(y^2)\right) \geq \mathrm{o}(x^2)$, since $0 \leq \mathrm{o}(y^2)$ we get that it is an equality. So $\mathrm{o}\left(\sqrt{\mathrm{o}\left(\mathrm{o}(x^2)\right)}\right) > x$, which implies by interval domain propagation that $x \in [2^{-537.5}; 7800^-]$, and by deciding on the bounds (section 4.4) we find that $x = 2^{-537.5}$ is a counter-example. If the formula contained the additional hypothesis that $x >= 0.000001$, then $\mathrm{o}(x^2)$ is a normal number and lemma 4.4 implies that $x > x$, unsatisfiable.

Other examples of applications of the lemmas above can be found in the tech report[6].

Through rounding. Let F be a non-`NaN` floating-point number, the following properties are true for all rounding mode $r \in \{\texttt{RNE}, \texttt{RNA}, \texttt{RTP}, \texttt{RTN}, \texttt{RTZ}\}$:

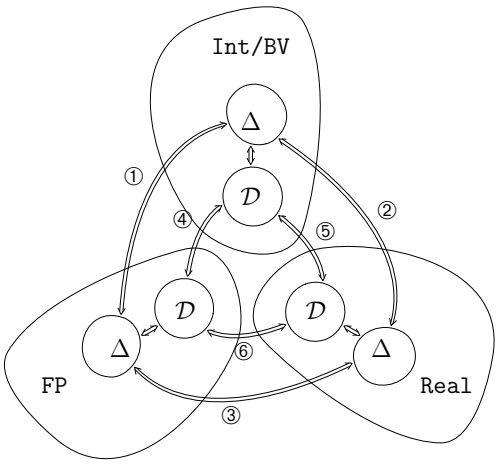
$$\begin{aligned}
F \geq 0.0 &\Rightarrow \texttt{rti}(\texttt{RTZ}, F) \leq F \qquad & \texttt{rti}(\texttt{RTN}, F) \leq F \\
F \leq 0.0 &\Rightarrow \texttt{rti}(\texttt{RTZ}, F) \geq F \qquad & \texttt{rti}(\texttt{RTP}, F) \geq F \\
\texttt{rti}(r, F) = F &\Longleftrightarrow (\texttt{fp.isInfinite}(F) \vee \texttt{isintegral}(F)) \\
\texttt{rti}(\texttt{RTN}, F) = \texttt{rti}(\texttt{RTP}, F) &\Rightarrow (\texttt{fp.isInfinite}(F) \vee \texttt{isintegral}(F))
\end{aligned}$$

Through type cast. The integral status used above is fixed by the casting operators from integer to FPNums. This status is propagated through operators $uop \in \{\texttt{fp.neg}, \texttt{fp.abs}\}$ and $op \in \{\texttt{fp.add}, \texttt{fp.sub}, \texttt{fp.mul}\}$:

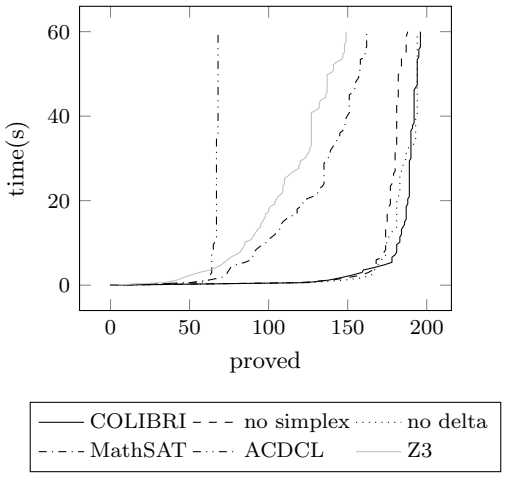
$$\begin{aligned}
\texttt{fp.isFinite}(\texttt{fp.to_fp}(r, I)) &\Rightarrow \texttt{isintegral}(\texttt{fp.to_fp}(r, I)) \\
\texttt{isintegral}(F) &\Rightarrow \texttt{isintegral}(uop(F)) \\
\texttt{isintegral}(F_1) \wedge \texttt{isintegral}(F_2) \wedge \texttt{fp.isFinite}(op(F_1, F_2)) &\Rightarrow \texttt{isintegral}(op(F_1, F_2))
\end{aligned}$$

If the FP arguments and result of an operation $op \in \{\texttt{fp.add}, \texttt{fp.sub}, \texttt{fp.mul}\}$ are of an integral status and if they are representable exactly (*e.g.,* $\texttt{fp.abs}(F) \leq_f 2^m$ where m is the size of the mantissa), then the treatment of op can be delegated to the integer domain machinery, $op_i \in \{\texttt{bv\_add}, \texttt{bv\_sub}, \texttt{bv\_mul}\}$, r a rounding mode.

$$\left.\begin{aligned}
&\texttt{isintegral}(F_1) \\
&\texttt{isintegral}(F_2) \\
&\texttt{fp.abs}(F_1) \leq_f 2^m \\
&\texttt{fp.abs}(F_2) \leq_f 2^m \\
&\texttt{fp.abs}(op(r, F_1, F_2)) \leq_f 2^m \\
&\neg\texttt{fp.isZero}(op(r, F_1, F_2))
\end{aligned}\right\} op(r, F_1, F_2) = \texttt{fp.to_fp}(r, op_i(\texttt{fp.to_sbv}(r, F_1), \texttt{fp.to_sbv}(r, F_2)))$$



(a) Interreductions



(b) Comparison of provers

4.2.3 Linearization

Linearization of floating-point arithmetic formulas has been presented in [3]. The goal is to transform relations on floating-point formulas to relations on linear rational formulas by relaxation, and so to be able to use a simplex. The idea is to use already known techniques on linearization of rational arithmetic formulas to linear one and to add the linearization of the rounding operator o.

The main idea for the linearization of o is that if $x \in \mathcal{R}$ is in the range of normal positive FPNum with double precision:

$$(1 - \frac{1}{2^{52}-1}) \cdot x \leq \mathrm{o}(x) \leq (1 + \frac{1}{2^{52}+1}) \cdot x.$$

For each operation with finite argument we add a constraint that bound the relative error between the floating-point operation and the operation on reals using the current domain of the arguments.

The addition of this technique makes immediate the proof of the following example:

$$(0. \leq_f x \leq_f 10.0) \wedge (0. \leq_f y \leq_f 10.0) \Rightarrow$$
$$\texttt{fp.sub}(\texttt{RNE}, \texttt{fp.sub}(\texttt{RNE}, \texttt{fp.add}(\texttt{RNE}, x, y), x), y) \leq_f 0.0001$$

The simplex is costly so the distance graph is used in COLIBRI to know when it is interesting to use the linearization and on which part of the problem. On the Griggio Benchmark it shows a significant gain.

4.3 Interreduction, Simplifications

The available interreductions are pervasive as shown in fig. 1a. The simplest ④, ⑤, ⑥ are forward and backward propagators of cast constraints (rounding, truncation, ...) on the interval domains. More particular is the setting of the integral status for casts from Int to FP ④ and the propagation between the bit-vector domain [10] and the IEEE representation of floats. These propagations are able to share information between the intervals of FPNums, the integral status and the bit-vector domain which tells which bits are known.

The different delta domains also communicate ①, ②, ③ by propagating the distance of one edge (*e.g.,* $d_{\mathcal{F}}(x, x')$) to the corresponding casted edge (*e.g.,* $|\texttt{fp.to\_sbv}(x) - \texttt{fp.to\_sbv}(x')|$). Currently the propagation is coarse by only propagating if the distance is positive, positive-or-null, negative, negative-or-null. The other casts `fp.to_real`, `to_fp`, ... are treated the same way and not only by using the same cast in both ways but also by coming back using the cast in the other way.

COLIBRI does congruence closure, which is not common in CP, for the operators taking into account specific algebraic properties (commutativity, $(a + (-b)) = a - b$, ...). Simplifications corresponds to replacing a constraint by another. Simplifications are more numerous and use domains information: replacing a constraint by its argument that absorbs the other, replacing composition of cast to and from integer by identity when the FPNum is known to be integral, replacing a constraint on FPNums known to be integral into the equivalent integer constraint (delegation).

4.4 Decisions

COLIBRI uses classical heuristics of the CP community, for both the variable choice and the domain splitting. We briefly highlight some of the procedures for the latter to give an insight: if a FP union of intervals contains more than one interval, pick the interval containing the smallest absolute value. If the union of intervals contains a single interval, then split on the zero value, if it is in that interval (thus creating two intervals, one positive and one negative). If the zero is not contained in that interval, pick one of the bounds (upper or lower). Then try a random value.

Note that COLIBRI does not do any conflict analysis or learning.

5 Conclusion and future work

unsat/sat	COLIBRI	no simplex	no delta	MathSAT	ACDCL	Z3
COLIBRI	-	18/ 0	15/ 2	21/21	81/49	19/33
no simplex	2 / 8	-	4 / 3	21/28	65/57	20/40
no delta	4 /11	9 / 4	-	21/30	70/58	21/43
MathSAT	1 / 7	17/ 6	12/ 7	-	62/38	5 /22
ACDCL	0 / 2	0 / 2	0 / 2	1 / 5	-	3 / 6
Z3	1 / 4	18/ 3	14/ 5	7 / 7	66/24	-

Figure 2: Number of goals (unsat/sat) the solver in the row is able to solve that the solver in the column cannot (*e.g.,* COLIBRI solves 19 unsat and 33 sat goals that Z3 does not solve).

In this paper we described an efficient CP-based approach for solving QF_FP problems, bringing forth an accessible tool, COLIBRI, and improving the capabilities of verification of programs that use floating-point data types. This work exhibits, in particular, the advantage of the natural interreduction machinery offered by Constraint Programming. While the verification community generally relies on the SMT framework, we explore and demonstrate the feasibility of an efficient alternative CP approach. Theory combination is also obtained through communications with bit-vectors, integers and reals.

We made a comparison[4] of FP solvers on SMTLIB 2017 Griggio and Schanda's benchmarks in fig. 1b and fig. 2; "no simplex" is COLIBRI without the simplex and linearization, "no delta"

[4]available at https://www.starexec.org/starexec/secure/details/job.jsp?id=21872

is COLIBRI no simplex without the domain of deltas and ACDCL is MathSAT with option `-theory.fp.mode=2`. It shows a clear advantage of COLIBRI techniques with 196 problems solved against 162 for MathSAT with a timeout of 1min. In our practice for program verification a timeout of more than 10s is rare. Results with a greater timeout will be available at the SMT competition 2017 where COLIBRI participates. On Schanda's benchmarks deltas help solve 10 problems. In [10] benchmarks showed that COLIBRI solved all the `QF_BVFP` examples of 2016 except one in 0.26s (around its start-up time).

Surprisingly COLIBRI no delta proves nearly as many goals as COLIBRI, fig. 2 shows that COLIBRI no delta is better at proving satisfiability and COLIBRI is better at proving unsatisfiability, a possible reason is that taking time to remove values for proving unsatisfiability is an overhead when the solution is just to find one value. We will look at improving the performance of the deltas.

The immediate next step seems to be investigating the benchmarks that reached timeout and understand how to solve them. We also plan on increasing the precision of our projections. Furthermore, COLIBRI currently propagates only from floating-point interval domains to deltas but not the converse, we would like to remove these restrictions. Following ideas in Gappa, we can extend the integral status with the $@\texttt{FIX}(f, p)$ predicate, indicating when f verifies $\exists\, z \in \mathcal{Z}.\, f = z.2^p$, which can also be seen as rational congruence in the reals. We would like also to extend COLIBRI to other parts of the SMTLIB, uninterpreted functions and quantifiers.

The techniques described in this paper are of course not limited to the CP world. They could be put inside a black box inside an SMT solver using any combination techniques, however the combination with other theories like bitvectors or reals is going to be restricted. Perhaps using new frameworks like MC-sat [19], it will be possible to overcome these difficulties and share the advantage of both worlds.

References

[1] R. Bagnara et al. "Filtering Floating-Point Constraints by Maximum ULP". In: vol. abs/1308.3847. 2013.

[2] S. Bardin and P. Herrmann. "OSMOSE: Automatic Structural Testing of Executables". In: *Softw. Test. Verif. Reliab.* 21.1 (2011).

[3] M. S. Belaid, C. Michel, and M. Rueher. "Boosting Local Consistency Algorithms over Floating-point Numbers". In: *Proceedings of the 18th International Conference on Principles and Practice of Constraint Programming*. 2012.

[4] A. Biere et al. "Symbolic Model Checking without BDDs". In: *Tools and Algorithms for the Construction and Analysis of Systems*. 1999.

[5] B. Blanc et al. "Handling State-Machines Specifications with GATeL". In: *Electr. Notes Theor. Comput. Sci.* 264.3 (2010).

[6] F. Bobot et al. *FPA Solver*. Tech. rep. http://soprano-project.fr/downloads/D3_1.pdf. ANR SOPRANO, ANR-14-CE28-0020, 2016.

[7] S. Boldo. "Stupid is as Stupid Does: Taking the Square Root of the Square of a Floating-Point Number". In: *Proceedings of the 7th and 8th International Workshop on Numerical Software Verification*. Vol. 317. 2015.

[8] M. Brain et al. "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*. 2015.

[9] M. Brain et al. “Deciding floating-point logic with abstract conflict driven clause learning”. In: *Formal Methods in System Design* 45.2 (2014).

[10] Z. Chihani et al. “Sharpening Constraint Programming approaches for Bit-Vector Theory”. In: *CPAIOR*. 2017.

[11] A. Cimatti et al. “The MathSAT5 SMT Solver”. In: *TACAS*. 2013.

[12] S. Conchon et al. “A three-tier strategy for reasoning about floating-point numbers in SMT”. In: *Computer Aided Verification*. 2017.

[13] S. Conchon et al. “Built-in Treatment of an Axiomatic Floating-Point Theory for SMT Solvers”. In: *SMT workshop*. 2012.

[14] R. David et al. “BINSEC/SE: A Dynamic Symbolic Execution Toolkit for Binary-Level Analysis”. In: *SANER 2016*. 2016.

[15] L. De Moura and N. Bjorner. “Z3: An efficient SMT solver”. In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008.

[16] T. Feydy, A. Schutt, and P. J. Stuckey. “Global Difference Constraint Propagation for Finite Domain Solvers”. In: *PPDP*. 2008.

[17] A. Gotlieb, M. Leconte, and B. Marre. “Constraint Solving on Modular Integers”. In: 2010.

[18] *IEEE standard for binary floating-point arithmetic*. Note: Standard 754–1985. Institute of Electrical and Electronics Engineers, 1985.

[19] D. Jovanović, C. Barrett, and L. de Moura. “The Design and Implementation of the Model Constructing Satisfiability Calculus”. In: *Proceedings of 13th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2013*. 2013.

[20] B. Marre and B. Blanc. “Test Selection Strategies for Lustre Descriptions in GaTeL”. In: *ENTCS*. Vol. 111. 2005.

[21] B. Marre and C. Michel. “Improving the Floating Point Addition and Subtraction Constraints”. In: *Principles and Practice of Constraint Programming - CP 2010*. 2010.

[22] C. Michel. “Exact Projection Functions for Floating-Point Number Constraints”. In: *International Symposium on Artificial Intelligence and Mathematics, AI&M 2002, Fort Lauderdale, Florida, USA, January 2-4, 2002*. 2002.

[23] R. A. D. Millo, R. J. Lipton, and A. J. Perlis. “Social Processes and Proofs of Theorems and Programs”. In: 1979.

[24] J.-M. Muller et al. *Handbook of Floating-Point Arithmetic*. ACM G.1.0; G.1.2; G.4; B.2.0; B.2.4; F.2.1., ISBN 978-0-8176-4704-9. Birkhäuser Boston, 2010.

[25] O. Ohrimenko, P. J. Stuckey, and M. Codish. “Propagation via lazy clause generation”. In: *Constraints* 14.3 (2009).

[26] P. Rümmer and T. Wahl. “An SMT-LIB theory of binary floating-point arithmetic”. In: *International Workshop on Satisfiability Modulo Theories (SMT)*. 2010.

[27] M. Veksler and O. Strichman. “Learning general constraints in CSP”. In: *Artificial Intelligence* 238 (2016).

[28] N. Williams, B. Marre, and P. Mouy. “On-the-Fly Generation of K-Path Tests for C Functions”. In: *ASE 2004*. 2004.