




**SOPRANO**  
*Novel Automatic Solver for Program Analysis*

SOPRANO

14/09/2016  
D1.1-v1-67-ge18edb6

# FPA Solver

 <b>SOPRANO</b> <i>Novel Automatic Solver for Program Analysis</i>	<b>Confidentiality</b>		<b>Client</b>
	Defense No	Industry No	ANR
	<b>Contractual</b> Yes		<b>Funding</b> ANR
	<b>WorkPackage</b> D3.1		
<b>Title</b>			
<b>FPA Solver</b>			
<b>Author(s)</b> François Bobot – CEA Zakaria Chihani – CEA Mohamed Iguernlala – OCamlPro Bruno Marre – CEA			
<b>Revision</b> D1.1-v1-67-ge18edb6			<b>Number of pages</b> <a href="#">25</a>
<b>Abstract</b>			
<b>Keywords</b> automated reasoning, floating-point arithmetic			
<b>Partners</b> CEA OCAMLPRO UPSUD		<b>Leader of the Workpackage</b> UPSUD	
SOPRANO (ANR-14-CE28-0020) from The National Research Agency (ANR) – © 2016			

---

# Contents

<b>Introduction</b>	<b>5</b>
<b>COLIBRI</b>	<b>7</b>
2.1 Domain propagation . . . . .	7
2.2 Difference logic . . . . .	9
2.3 Monotonicity . . . . .	12
2.4 Linearization . . . . .	14
2.5 Current and future works . . . . .	15
<b>Alt-Ergo</b>	<b>17</b>
3.1 Handling of arithmetic constraints in Alt-Ergo . . . . .	17
3.2 Semantic triggers . . . . .	17
3.3 Partial interpretation of arithmetic operators . . . . .	18
3.4 Inferring better bounds before semantic matching . . . . .	19
3.5 Handling of generated instances . . . . .	19
3.6 Current database of axioms . . . . .	19
3.7 Current results . . . . .	19
3.8 Current and future works . . . . .	21
<b>Conclusion</b>	<b>23</b>
References . . . . .	24
<b>Bibliography</b>	<b>25</b>



## Introduction

Floating-point arithmetic tackles an impossible problem: representing real numbers in finite space. It is an old invention that predates computers (used in 1914), standardized in early 1980 as the IEEE-754 [1]. Despite their ubiquity, floating-point numbers are unnatural to work with due to the different results obtained even from bounded arithmetic. Consider, for example, that  $1. + 2^{100} = 2^{100}$ , while  $\overbrace{0.1 + \dots + 0.1}^{10} \neq 0.1 \times 10. = 1.$  because 0.1 is not representable as a binary floating-point number.

However the semantics of floating-point operators, given by IEEE-754, are clear: the result of a floating-point operator is the result of the rounding of the real operator result ( $x +. y = o(x + y)$ ). Henceforth, we use only the formulation with the rounding.

Adacore provided in deliverable D1.2 a set of simple functions with specifications, for example fig. 1.1. Interestingly many of these examples come from actual industrial users that had to create new rules (lemmas) for proving their programs, such as fig. 1.2.

During multiple meetings of the Soprano project, we compared how each tool (Gappa, COLIBRI, Alt-Ergo +axioms, Alt-Ergo +FP) behaved on the Adacore examples, leading us to improve those tools. Only too specific prob-

```
1 procedure Range_Mult (X : Float_32; Res : out Float_32) is  
  begin  
3   pragma Assume (X in 5.0 .. 10.0);  
   Res := X * 2.0 - 5.0;  
5   pragma Assert (Res ≥ X);  
  end Range_Mult;
```

Figure 1.1: `Range_Mult` example from Adacore

```
2 procedure User_Rule_3 (X, Y : Float; Res : out Boolean) is  
3 begin  
4   pragma Assume (X < Y);  
5   pragma Assume (Y > 0.0);  
6   Res := X / Y ≤ 1.0;  
7   pragma Assert (Res);    -- valid  
8 end User_Rule_3;
```

Figure 1.2: `User_Rule_3` example from Adacore

lems remain unsolved (eg. fast inverse approximated square root). The tools are currently tested by Adacore directly in their toolchain. Some of these techniques will be used to add floating-point reasoning in the Popop solver.

In a first part we will see the current set of techniques used by COLIBRI for solving floating-point constraints. In a second part we will describe the techniques that have been added to Alt-Ergo. These two sets of techniques differ because the above solvers each already contained some machinery for handling floating-point numbers.

---

# COLIBRI

## 2.1 Domain propagation

The 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 rules*. We show here the mathematical version of these Ada codes.

$$\left. \begin{array}{l} -0 \leq x \leq 16777216.0 \\ -0 \leq y \leq 16777216.0 \\ -0 \leq z \end{array} \right\} \implies o(-o(x \times y)) \leq z$$

Domain propagation is at the heart of constraint programming, on which COLIBRI is based. It consists in associating domains to each term, here a floating-point interval and a boolean domain, and to each operator  $\otimes$  a design function that continuously improves the domain of  $x$ ,  $y$ , and  $x \otimes y$  using the domain of the other terms. For the floating-point operators, COLIBRI considers  $o(x \otimes y)$  as one operator. This simplifies the design of precise propagators, as we will see.

In this example, initially, the domains of boolean terms are in  $\{\perp; \top\}$  and the domains of all the floating-point terms are  $[-\infty; +\infty]$ , COLIBRI handles infinite and finite floating-point numbers (32 bits and 64 bits with nearest-to-even rounding) but currently not NaN. The domain of the conclusion  $o(-o(x \times y)) \leq z$  is set to  $\{\perp\}$  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 do-

mains:

$$\begin{aligned} o(x \times y) &\in [+0; 281474976710656.] \\ o(-o(x \times y)) &\in [-281474976710656.; +0] \\ (o(-o(x \times y)) \leq z) &\in \{\top\}. \end{aligned}$$

The last line contradicts the initial setting to  $\perp$  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 that  $\times$ ,  $-$ ,  $o$  are part-wise monotonic.

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

$$\begin{aligned} o(-o(x \times y)) \leq z &\in \{\perp\} \\ o(-o(x \times y)) &\in [s; +\infty] \\ o(x \times y) &\in [-\infty; -s] \\ o(x \times y) &\in ([-\infty; -s] \cap [-0; 281474976710656.]). \end{aligned}$$

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 propagations [6] that rely on other properties than monotonicity. For example, starting with  $x, y \in [+0; 1000]$  and the constraint  $o(x - y) = o(0.1) \simeq 0.10000000149$  (in 32 bit), the previous propagators applied once do not reduce the domains significantly. The intervals would become  $x \in [o(0.1); 1000]$ ,  $y \in [0; 1000]$ . With the new propagations [6] however, the intervals get restricted to  $x \in [o(0.1); o(0.225)]$  and  $y \in [+0; o(0.12499999)]$ . Indeed  $x$  and  $y$  cannot be large because of the position of the least-significant bit of the significand of  $o(0.1)$ :  $10011001100110011001101^2$ .

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.



## 2.2 Difference logic

On this example from Adacore, previous propagations are not enough:

$$5.0 \leq x \leq 10.0 \implies o(o(2.0 \times x) - 5.0) \geq x$$

The domain of  $o(o(2.0 \times x) - 5.0)$  obtained is  $[5.0; 15.0]$ . Thus we need some global propagations to solve this problem.

Without rounding, the rational numbers problem  $5 \leq x \leq 10 \implies (2x) - 5 \geq x$  can be solved using distance graph [?] where each node is a term, and each edge is labeled by an over-approximation of the distance between the two terms. Let  $d^{\mathcal{Q}}(x, y) = y - x$  be the signed distance between two terms. We have  $d^{\mathcal{Q}}(2x, 2x - 5) \in \{-5\}$ . From  $x \in [5; 10]$ , the distance graph deduces  $d^{\mathcal{Q}}(x, 2x) \in [5; 10]$ . Finally  $d^{\mathcal{Q}}(x, 2x - 5) = d^{\mathcal{Q}}(x, 2x) + d^{\mathcal{Q}}(2x, 2x - 5) \in [0; 5]$ , which proves the inequality.

One could extend that technique to floating-point numbers by computing  $d^{\mathcal{Q}}(y, o(y))$  using the size of the ULP (unit in the last place), the distance between two consecutive floating-point numbers, in the domain of  $y$ . Unfortunately that introduces too big of an over-approximation for proving the properties:  $d^{\mathcal{Q}}(x \times 2, o(x \times 2)) = [-\frac{1}{2} \frac{20}{2^{-53}}; +\frac{1}{2} \frac{20}{2^{-53}}]$  for 64-bit floating-point numbers, and so we cannot prove that  $d^{\mathcal{Q}}(x, x \times 2 - 5)$  is positive.

A way to regain precision is to consider the operation and the rounding as one operation, in other words to consider the floating-point operator instead of the rational one. In that case, properties of the floating-point operators can be used such as:

- multiplications by a power of two are exact, if  $y$  and  $2y$  are normalized,  $d^{\mathcal{Q}}(2^n y, o(2^n y)) = 0$  (because the only change the exponent);
- some subtractions are exact, if  $0 \leq x \leq y$  and  $y - x$  is normalized,  $d^{\mathcal{Q}}(y - x, o(y - x)) = 0$  (because  $y - x$  is in a binade at least more precise than the ones of  $x$  and  $y$ ).

Therefore, since

$$\begin{aligned}
 d^{\mathcal{Q}}(x, o(o(2.0 \times x) - 5.0)) = & \overbrace{d^{\mathcal{Q}}(x, 2.0 \times x)}^{\in [5; 10]} \\
 & + \overbrace{d^{\mathcal{Q}}(2.0 \times x, o(2.0 \times x))}^{\in \{0\}} \\
 & + \overbrace{d^{\mathcal{Q}}(o(2.0 \times x), o(2.0 \times x) - 5.0)}^{\in \{-5\}} \\
 & + \overbrace{d^{\mathcal{Q}}(o(2.0 \times x) - 5.0, o(o(2.0 \times x) - 5.0))}^{\in \{0\}}
 \end{aligned}$$

we have that  $d^{\mathcal{Q}}(x, o(o(2.0 \times x) - 5.0)) \in [0; 5]$ , which proves the fact.

However, this loses precision as soon as properties cannot be applied. For example if the assertion is

$$5.0 \leq x \leq 10.0 \implies o(o(x/2) + 2.5) \leq x$$

then this method cannot solve directly the problem, since  $d^{\mathcal{Q}}(y+2.5, o(y+2.5))$  is not always negative for  $y \in [2.5; 5]$ :

$y$	2.5	2.5 <sup>+</sup>	2.5 <sup>++</sup>	2.5 <sup>+++</sup>
$o(y + 2.5)$	5	5	5 <sup>+</sup>	5 <sup>++</sup>
$d^{\mathcal{Q}}(y + 2.5, o(y + 2.5))$	0	$-\frac{\text{ulp}(5)}{2}$	0	$+\frac{\text{ulp}(5)}{2}$
$d^{\mathcal{Q}}(y, o(y + 2.5))$	2.5	$2.5 - \frac{\text{ulp}(5)}{2}$	2.5	$2.5 + \frac{\text{ulp}(5)}{2}$

it follows that the distance  $d^{\mathcal{Q}}(x, o(o(x/2) + 2.5)) \in [2.5 - \frac{\text{ulp}(5)}{2}; + \frac{\text{ulp}(5)}{2}]$ . The problem comes from the non-monotonicity of the distance, the biggest error is not found on the extremity of the interval. Using this technique, the problem could still be solved by backward propagating that  $o(o(\frac{x}{2}) + 2.5) > x$  which reduces the domain of  $x$  to  $[5; 5^+]$  which proves by domain propagation, the domain is smaller, that  $o(o(\frac{x}{2}) + 2.5) \in \{5\}$ .

In COLIBRI another definition of distance  $d^{\mathcal{F}}$  is used to solve this problem with less propagation. The idea is to use a distance that is based on integers instead of rationals and tailored to floating-point numbers.

For  $x \in \mathcal{F}$ ,  $\text{num}(x)$  is defined by

$$\text{num}(x) = \begin{cases} -|\{z \in \mathcal{F} | x \leq z < 0\}| & \text{when } x < 0 \\ |\{z \in \mathcal{F} | 0 \leq z < x\}| & \text{when } 0 \leq x \end{cases}$$

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

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

As a trivia, if  $x \in \mathcal{F}$  is given in IEEE-754 format, it is easy to compute  $\text{num}(x)$  since it is the reinterpretation of  $x$  in int64 (sign,exponent,mantissa). The next floating-point number is obtained by the successor function and the previous floating-point number by the predecessor function.

This distance can be a little counter-intuitive because there are more floating-point numbers near 0, *e.g.*,  $d^{\mathcal{F}}(1, 2) > d^{\mathcal{F}}(2, 3)$ , which implies that, for  $x \in \mathcal{F}$  with  $x \in [1; 2]$ , the distance between  $x$  and  $x + 1$  is not a singleton:  $d^{\mathcal{F}}(x, x + 1) \in [d^{\mathcal{F}}(2, 3), d^{\mathcal{F}}(1, 2)]$ . Moreover, if  $x \in [-2; 3]$  then the distance is over-approximated by  $d^{\mathcal{F}}(x, x + 1) \in [d^{\mathcal{F}}(3, 4), d^{\mathcal{F}}(-0.5, 0.5)]$ . Generally, with  $x \in [\underline{x}; \bar{x}]$ ,  $y \in [\underline{y}; \bar{y}]$ , let  $\underline{m} = \max(\underline{x}, \frac{-\bar{y}}{2})$  and  $\bar{m} = \min(\bar{x}, \frac{-\underline{y}}{2})$ , then:

$$\begin{aligned}
d^{\mathcal{F}}(x, o(x+y)) &\in \left[ \begin{array}{l} \min(d^{\mathcal{F}}(\underline{x}, \underline{x} + \underline{y}), d^{\mathcal{F}}(\bar{x}, \bar{x} + \bar{y})), \\ \max(d^{\mathcal{F}}(\underline{m}, \underline{m} + \bar{y}), d^{\mathcal{F}}(\bar{m}, \bar{m} + \underline{y})) \end{array} \right] \\
d^{\mathcal{F}}(x, o(2 \times x)) &= d^{\mathcal{F}}(1, 2) \\
d^{\mathcal{F}}(x, o(2^n \times x)) &= n \times d^{\mathcal{F}}(1, 2) \\
d^{\mathcal{F}}(x, o(\frac{x}{2^n})) &= -n \times d^{\mathcal{F}}(1, 2) \\
\overline{d^{\mathcal{F}}(x, o(x \times y))} &\in \begin{cases} 0 & \text{when } \bar{y} = 1 \\ 0 & \text{when } \underline{x} = 0 \\ d^{\mathcal{F}}(o(\frac{[\bar{x} \times \bar{y}]_2}{\bar{y}}), [\bar{x} \times \bar{y}]_2) & \text{when } 0 < x, 0 < y, 1 < \bar{y} \\ d^{\mathcal{F}}(x_m, x_m \times y_M) & \text{when } 0 < x, 0 < y < 1, 0 = o(x_m \times y_m) \\ d^{\mathcal{F}}(o(\frac{[\underline{x} \times \underline{y}]_2}{\underline{y}}), [\underline{x} \times \underline{y}]_2) & \text{when } 0 < x, 0 < y < 1, 0 < o(x_m \times y_m) \\ \infty & \text{otherwise} \end{cases} \\
d^{\mathcal{F}}(x, o(x \times y)) &\in \begin{cases} -d^{\mathcal{F}}(0, \bar{x}) & \text{when } \underline{y} = 0 \\ d^{\mathcal{F}}(o([\bar{x}]_2, [\bar{x}]_2 \times \underline{y})) & \text{when } 0 < x, 0 < y, \underline{y} < 1 \\ 0 & \text{when } \underline{x} = 0, 1 < \underline{y} \\ 0 & \text{when } \underline{y} = 1, 1 < \underline{y} \\ -\infty & \text{otherwise} \end{cases}
\end{aligned}$$

$d^{\mathcal{F}}(1, 2)$  corresponds to the number of floating-point numbers in a binade and  $\lfloor c \rfloor_2 = 2^{\lfloor \log_2(c) \rfloor}$ . The propagations described are not tight especially for the multiplication.

If we look back at where  $d^{\mathcal{Q}}$  was losing precision, we see that  $d^{\mathcal{F}}$  is monotonic:

$y$	2.5	2.5 <sup>+</sup>	2.5 <sup>++</sup>	2.5 <sup>+++</sup>
$o(y + 2.5)$	5	5	5 <sup>+</sup>	5 <sup>++</sup>
$d^{\mathcal{F}}(y, o(y + 2.5))$	$2^{52}$	$2^{52} - 1$	$2^{52} - 1$	$2^{52} - 1$

COLIBRI uses a distance graph with  $d^{\mathcal{F}}$  for managing these informations. It is populated using ordering constraints and operators. The closure of the graph is done lazily when cycles are created or modified. Domains of the nodes are improved using the known costs.

With the first example:

- $d^{\mathcal{F}}(x, o(x \times 2.0)) \in \{d^{\mathcal{F}}(1, 2)\}$
- $d^{\mathcal{F}}(o(x \times 2.0), o(o(x \times 2.0) - 5.0)) \in [d^{\mathcal{F}}(10.0, 5.0); d^{\mathcal{F}}(20.0, 15.0)] = [-d^{\mathcal{F}}(1, 2); -d^{\mathcal{F}}(15.0, 20.0)]$
- $d^{\mathcal{F}}(x, o(o(x \times 2.0) - 5.0)) \in [0; d^{\mathcal{F}}(1, 2) - d^{\mathcal{F}}(15.0, 20.0)]$  by transitivity.

With the second example:

- $d^{\mathcal{F}}(x, o(\frac{x}{2.0})) \in \{-d^{\mathcal{F}}(1, 2)\}$
- $d^{\mathcal{F}}(o(\frac{x}{2.0}), o(o(\frac{x}{2.0})+2.5)) \in [d^{\mathcal{F}}(5.0, 7.5); d^{\mathcal{F}}(2.5, 5.0)] = [d^{\mathcal{F}}(5.0, 7.5); d^{\mathcal{F}}(1, 2)]$
- $d^{\mathcal{F}}(x, o(o(\frac{x}{2.0}) + 2.5)) \in [-d^{\mathcal{F}}(1, 2) + d^{\mathcal{F}}(5.0, 7.5); 0]$  by transitivity.

COLIBRI is able to solve considerably more problems using these techniques. The distance graph is also very useful for solving harder problems.

### 2.3 Monotonicity

The example `User_Rule_4` from Adacore states that

$$X \geq Y \implies Y > 0.0 \implies \frac{X}{Y} \leq 1.0$$

which leads to proving that the following constraints is unsatisfiable:

$$X \geq Y \wedge Y > 0.0 \wedge o\left(\frac{X}{Y}\right) > 1.0.$$

We can do the following deduction on the distance of the subterms:

$$\begin{array}{ll} o\left(\frac{X}{Y}\right) > 1.0 & \\ o\left(\frac{X}{Y}\right) > o(1.0) & \text{since } 1.0 \in \mathcal{F} \\ \frac{X}{Y} > 1.0 & \text{by contraposition of the monotonicity of } o \\ X > Y & \text{since } Y > 0 \end{array}$$

which contradicts  $X \leq Y$ , and proves the assertion. More generally:

**Lemma 1.**

$$\forall x, y, z \in \mathcal{Q}, 0 < y \implies o\left(\frac{x}{y}\right) < o(z) \implies o(x) \leq o(z \times y)$$

Since the two sides of the implication are on floating-point numbers, it is used for populating the COLIBRI distance graph. For example, for every edge that joins a node  $z$  and a node that corresponds to the floating-point division of  $x$  by  $y$ , such that the floating-point multiplication of  $z$  by  $y$  already appears in the graph and such that  $y \in (0; +\infty]$ , the distance of the edge can be used to improve the edge between  $x$  and the multiplication. The limitation to already

known multiplications in the distance graph, and more generally known terms, is a usual technique for avoiding creating new terms, but it has the drawback of sometimes losing interesting propagations.

What have been done with division and multiplication can be done with any monotonic function and inverse function:

**Lemma 2.** *Let  $D, E \subset \mathcal{R}$ ,  $f : D \mapsto E$  and  $f^{-1} : E \mapsto D$  such that:*

- $\forall x : D, f^{-1}(f(x)) = x,$
- $f$  increasing,

we have

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

Computer arithmetics are used to this kind of reasoning. However, we are not aware of a large use in automated floating-point reasoning. Computer arithmetics are more interested in cases where the first implication is strict. For example it is strict for the multiplications by a power of two because such multiplications are exact:

**Lemma 3.** *Let  $x, y, z \in \mathcal{R}$  and  $n$  a natural number, if  $0 < y$ , and  $2^n < o\left(\frac{o(x)}{o(y)}\right)$  we have  $o(2^n \times y) < o(x)$*

This improvement is necessary for `User_Rule_7` provided by Adacore:

$$\left. \begin{array}{l} z \geq 0.0 \\ x \geq y \\ y \geq z \\ x > z \\ a \geq 1.0 \end{array} \right\} \implies o\left(\frac{o(x-y)}{o(x-z)}\right) \leq a$$

which is proved by applying first lemma 3 then lemma 2 with the function  $t \mapsto x - t$ .

The next example of Adacore (`User_Rule_16`) is interesting because it is false and involves the square root function. The assertion is

$$\left. \begin{array}{l} x \in [-7800; +7800] \\ y \in [-7800; +7800] \\ x > \text{abs}(y) \end{array} \right\} \implies o\left(\sqrt{o(o(x^2) - o(y^2))}\right) \leq x$$

The propagations used to find a counter-example work as following:

$$\begin{aligned}
& \text{o}\left(\sqrt{\text{o}(x^2) - \text{o}(y^2)}\right) > x \\
& \text{o}\left(\sqrt{\text{o}(x^2) - \text{o}(y^2)}\right) > \text{o}(x) && \text{since } x \in \mathcal{F} \\
& \text{o}(x^2) - \text{o}(y^2) \geq \text{o}(x^2) && \text{by lemma 2} \\
& \text{however } \text{o}(x^2) - \text{o}(y^2) \leq \text{o}(x^2) && \text{by monotonicity of } - \text{ and } \text{o} \\
& \text{so } \text{o}(x^2) - \text{o}(y^2) = \text{o}(x^2) && \text{using last two properties}
\end{aligned}$$

The last deduction indicates that  $y^2$  is absorbed by  $x^2$  which indicates that  $y \in [-6.103515625 \cdot 10^{-5}; 6.103515625 \cdot 10^{-5}]$ . More importantly it merges the nodes of  $\text{o}(x^2) - \text{o}(y^2)$  and  $\text{o}(x^2)$  which implies that  $x \in [2^{-537.5}; 7800^-]$ . COLIBRI then tries the bound of the domain of  $x$  and indeed  $x = 2^{-537.5}$  is a counter-example.

On the contrary, if the hypothesis  $x \geq 0.000001$  is added, COLIBRI is able to prove the goal. The first part of the reasoning is the same, except that  $x^2$  is now normalized. When  $\text{o}(x^2) - \text{o}(y^2)$  is merged with  $\text{o}(x^2)$  there is an edge in the distance graph between  $\text{o}\left(\sqrt{\text{o}(x^2)}\right)$  and  $x$  with  $x^2$  normalized which allows to use this known result [4]:

**Lemma 4.** *Let  $x \in \mathcal{F}$ , such that  $\text{o}(x^2)$  is normalized, then  $\text{o}\left(\sqrt{\text{o}(x^2)}\right) = x$*

This lemma contradicts the strict distance, and proves the goal. The contrapositive of this lemma is also implemented which allows in the previous example to reduce the domain of  $x^2$  to subnormal numbers.

As with usual rational arithmetic, difference logic is not enough for all of arithmetic.

## 2.4 Linearization

Linearization of floating-point arithmetic formulas has been presented in [2]. The goal is to transform relations on floating-point formulas to relations on linear rational formulas. 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  $\text{o}$ .

The main idea is that if  $x$  is positive and normalized then for 64-bit floating-point arithmetic:

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

No new example given by Adacore has been proved after adding the linearization in COLIBRI. However simple examples are proved, in a reasonable time, only with this technique, such as:

$$\left. \begin{array}{l} 0 \leq x \leq 10.0 \\ 0 \leq y \leq 10.0 \end{array} \right\} \implies o(o(o(x + y) - x) - y) \leq 0.0001$$

We are currently experimenting with this technique in COLIBRI. The simplex is costly so the distance graph is used to know when it is interesting to use the linearization and on which part of the problem.

## 2.5 Current and future works

At the start of the project we knew that COLIBRI would solve the simplest examples of Adacore because of the built-in domain propagation, but we have been impressed with the number of harder examples solved because of the distance graph that uses the  $d^{\mathcal{F}}$  distance. This distance and this algorithm have never been published and we hope to publish them soon. The techniques developed during the SOPRANO project based on the monotonicity and the distance graph are general and make it to solve a large number of hard problems; we also plan on publishing work on that topic. Despite the encouraging results (fig. 2.1), we have been reluctant to participate in QF\_FP category of the 2016 SMT-COMP, because COLIBRI forbids models that introduce NaN, which created a few wrong results.

Our immediate next steps will be:

- Publish the techniques used in COLIBRI.
- Add NaN handling with SMT-LIB2 input language.
- Solving more SMT-LIB2 floating-point problems, the major difficulty being to create terms that do not appear in the goal in order to apply the monotonicity property of lemma 2.
- COLIBRI currently uses a simplex that computes with floating-point number (which could introduce errors); we want to replace it by the simplex of Alt-Ergo that uses rational number. This simplex has been extracted by OCamlPro in a standalone library, `ocplib-simplex`.

The discussions about how COLIBRI and Gappa solved these problems helped to design the axioms used by Alt-Ergo.

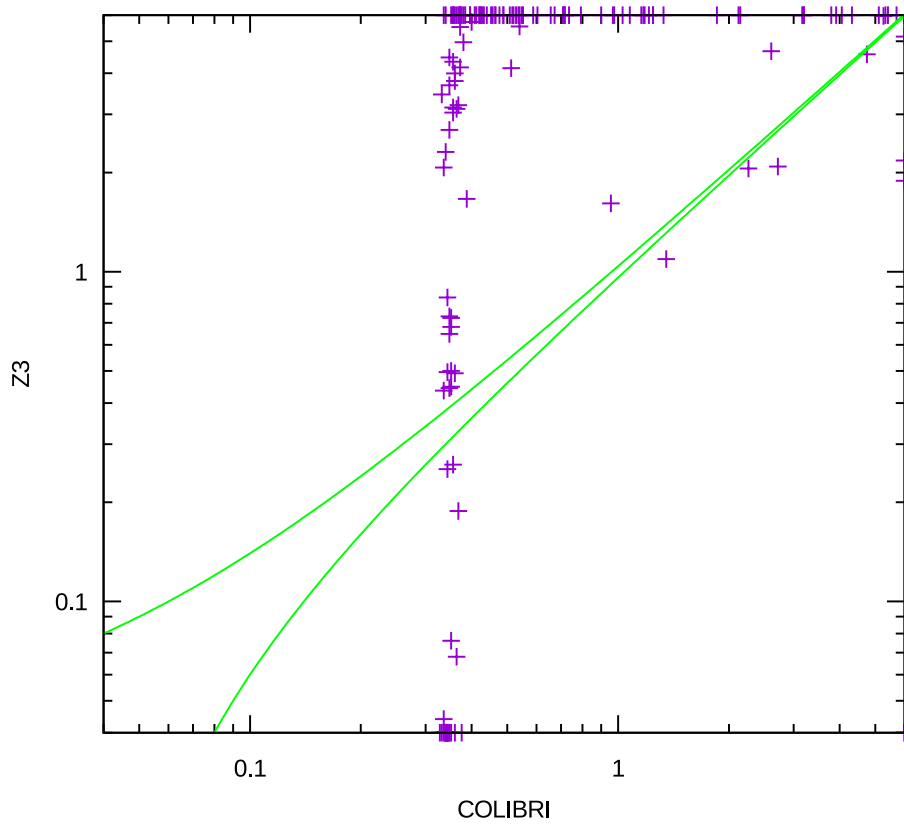


Figure 2.1: Results of COLIBRI on the `smtlib2` benchmarks of the `QF_FP` category. Each cross is a problem, its abscissa is the time taken by COLIBRI, its ordinate is the time taken by Z3. The timeout is fixed at 6s. We can see that after the startup time of 0.25s of COLIBRI (written in Eclipse Prolog) it solved nearly all the goal before Z3 solves them.



---

## Alt-Ergo

### 3.1 Handling of arithmetic constraints in Alt-Ergo

Alt-Ergo relies on a combination of dedicated algorithms to handle constraints involving (non-)linear integer and rational arithmetic. For linear arithmetic equalities, the mechanism of solving and substituting is used to build a convergent rewriting system modulo arithmetic. For inequalities, a Fourier-Motzkin algorithm and a propagation loop *à la CP* is used to maintain domains (unions of intervals) for arithmetic terms. This mechanism also includes some propagators of non-linear arithmetic. A more detailed description of this approach can be found in [5].

Our objective is to extend this framework in a modular way to be able to handle floating-point arithmetic. The approach relies on bounds refinement as for non-linear arithmetic.

### 3.2 Semantic triggers

In order to integrate floating-point arithmetic in Alt-Ergo, we investigated a lightweight approach that attempts to heavily reuse existing components of the solver. Our idea is to write floating-point arithmetic axioms in the (enriched) syntax of Alt-Ergo, and extend the solver to be able to cope with these axioms.

The starting point of the approach consists in extending the mechanism of triggers found in SMT solvers to be able to take theory reasoning into account when instantiating first-order formulas (*i.e.*, axioms). To illustrate this notion of triggers, consider the universally quantified formula below.

$$\forall x, y : real. \quad p(x, y) \rightarrow x \leq y + 1$$

To be able to take this formula into account when reasoning, SMT solvers usually generate finitely many ground instances from it (*i.e.*, substituting the variables  $x$  and  $y$  with some constants). This process is guided by heuristics to

limit the number of introduced facts, called triggers or guards. In the example above, a good guard is the term “ $p(x, y)$ ”: this means that a *ground* term of the form “ $p(a, b)$ ” should be present in the context of the solver to generate the instance “ $p(a, b) \rightarrow a \leq b + 1$ ”, where  $x$  is replaced with  $a$  and  $y$  is replaced with  $b$ .

Now, consider the following example, where  $\circ$  is a function that rounds a given rational into its FP representation *w.r.t* some FP format and to some rounding mode.

$$\forall x, i, j : \text{real}. \quad i \leq x \leq j \rightarrow \circ(i) \leq \circ(x) \leq \circ(j)$$

There are many (syntactic) triggers that can be chosen for this formula. However, the triggers are either too permissive or too restrictive. For instance, choosing the multi-guard “ $x, i, j$ ” (or even “ $\circ(x), i, j$ ”) is too permissive and will make the solver unusable in practice. On the other hand, the multi-trigger “ $\circ(x), \circ(i), \circ(j)$ ” is too restrictive and may prevent useful instances from being generated.

Our solution is to use a new kind of *theory-aware* triggers called *semantic triggers*. For the example above, a good guard consists in the combination of the syntactic trigger “ $\circ(x)$ ” with the semantic trigger “ $x \in [i, j]$ ”. This means that we will generate an instance “ $c_1 \leq a \leq c_2 \rightarrow \circ(c_1) \leq \circ(a) \leq \circ(c_2)$ ” only if there exists some ground term “ $\circ(a)$ ” that matches “ $\circ(x)$ ”, and such that the domain of  $a$  is  $[c_1, c_2]$ .

In our current implementation, we reuse the generic E-matching mechanism of Alt-Ergo to instantiate syntactic triggers. Semantic triggers are handled by extending the intervals calculus module with intervals matching capability.

### 3.3 Partial interpretation of arithmetic operators

Adding semantic triggers in Alt-Ergo is not sufficient to incorporate floating-point reasoning. We also extended the arithmetic part to be able to “compute” when operators are applied on constants. For instance, thanks to our semantic matching, we know that  $c_1$  and  $c_2$  in the fact below are constants. Therefore,  $\circ(c_1)$  and  $\circ(c_2)$  can be reduced to some constants  $d_1$  and  $d_2$  respectively, which makes it possible to deduce a domain for  $\circ(x)$ .

$$c_1 \leq a \leq c_2 \quad \rightarrow \quad \circ(c_1) \leq \circ(a) \leq \circ(c_2)$$

becomes

$$c_1 \leq a \leq c_2 \quad \rightarrow \quad d_1 \leq \circ(a) \leq d_2$$

where  $d_1 \equiv \circ(c_1)$  and  $d_2 \equiv \circ(c_2)$ .

In the current prototype, we added partial interpretation of FP rounding operators  $\circ$  (parametrized by the FP format and the rounding mode), for ra-

tional exponentiation, minimum, maximum and absolute value over integers and rationals, *etc.*

### 3.4 Inferring better bounds before semantic matching

As explained in Section 3.1, bounds inference in Alt-Ergo was mainly done by the Fourier-Motzkin algorithm and a propagation loop *à la CP*. However, these approaches may be incomplete. In addition, the terms we are interested in for the semantic matching step (like the term  $a$  in the semantic trigger  $a \in [i, j]$ ) may be uninitialized in the map of intervals. So, we cannot expect to get (refined) bounds for such terms.

In order to perform semantic matching on refined bounds, we incorporated a simplex algorithm with maximization capabilities to infer better lower and upper bounds of terms before intervals matching.

### 3.5 Handling of generated instances

Given an axiom with syntactic and semantic triggers, and once (syntactic and semantic) matching succeeded to find a relevant instance for the axiom, the instance is given back to the SAT solver. Then, the SAT uses the same mechanism as for generic instances to handle it.

Another possibility would be to process the instances inside the intervals calculus module to internally saturate the bounds of terms. This is what we did in our first prototype (deliverable D4.1). Actually, going back to the SAT provides a more precise idea on how information is propagated inside the demonstrator, since the SAT solver has a global vision of different reasoning components. In addition, it enjoys the advantages of (a) handling arbitrarily complicated axioms, and (b) being extensible for other axiomatized theories.

### 3.6 Current database of axioms

Instead of inlining floating-point axioms inside the solver, we decided to rather put them in a separate file written in Alt-Ergo's input language. This prelude consists of a set of functions and theories declarations (as shown below) that are loaded when FP reasoning is activated. It is currently made of approximately 80 axioms on floating-point arithmetic, absolute value, non-linear arithmetic, square root properties, *etc.* The advantages of putting the axioms in a separate text-file is that we get something that is easy to read, to debug, to modify, and to extend.

### 3.7 Current results

The prototype that implements the ideas described above is still under development. However current results are satisfactory. Our approach is able to

```

1 type fpa_rounding_mode = Ne | ToZero | Up | Down | Na
3 logic float : int , int , fpa_rounding_mode, real → real
5 logic abs_real : real → real
7 theory FPA_axioms extends NIA =
9   axiom rounding_operator_1 :
11     forall x : real .
12     forall i, j : real .
13     forall md : fpa_rounding_mode .
14     forall p,m : int
15     [ float (m,p,md,x), x in [i,j] ].
16     i ≤ x ≤ j →
17     float (m,p,md,i) ≤ float(m,p,md,x) ≤ float(m,p,md,j)
18     ...
19 end
21
23 theory Abs_Reals extends NIA =
25   axiom abs_real_interval_1 :
26     forall x : real
27     [ abs_real(x), abs_real(x) in [?i, ?j], 0. in ]?i, ?j[ ].
28     0. ≤ abs_real(x)
29     ...
31 end

```

prove 19 of the 27 very hard FPA benchmarks provided by AdaCore. We are currently considering a larger benchmark made of 1744 proof obligations (POs) that come from realistic C programs manipulating FP numbers<sup>1</sup> [3]. All these POs were proved valid by a combination of Alt-Ergo, Gappa, and the Coq proof assistant. Thanks to our approach, we improved the resolution success rate of Alt-Ergo by about 5%, as shown below. Moreover, a portfolio approach<sup>2</sup> allows us to prove all the POs that are discharged by Gappa.

Gappa	86.60 %
Alt-Ergo without FPA	90.20%
Alt-Ergo + FPA	94.80%
Alt-Ergo + FPA (Portfolio)	94.95%

<sup>1</sup><https://www.lri.fr/~sboldo/research.html>

<sup>2</sup>To be able to experiment with different options of the solver.

### 3.8 Current and future works

Different axes are currently considered:

1. We are still working on the benchmark made of 1744 POs. We think that we can still improve success rate and resolution time of our prototype,
2. We are also working on the translation of the FP benchmarks of SMT2 (more than 35,000 formulas) to Alt-Ergo via Why3. This would make it possible to better stress our approach for floating-point arithmetic. Note that, since our technique does not handle exceptional values, we need an intermediate layer (Why3) that is able to deal with them.



---

## Conclusion

The SOPRANO project made available two new and very efficient solvers for proving programs with floating-point numbers: COLIBRI which is now a standalone prover with new solving techniques for floating-point numbers, and Alt-Ergo which gained the ability to reason about them.

In the following table, Ok indicates that the problem has been solved in less than 1s, CE indicates that a counter-example has been found in less than one second. The column COLIBRI<sup>-</sup> corresponds to the version of COLIBRI before Soprano by using direct call to the API.

	COLIBRI <sup>-</sup>	COLIBRI	Alt-Ergo
Range_Add	Ok	Ok	Ok
Range_Mult	Ok	Ok	Ok
Range_Add_Mult	Ok	Ok	Ok
Guarded_Div	CE <sup>1</sup>	CE	
Fibonacci	CE <sup>1</sup>	CE	
Int_To_Float_Complex	Ok	Ok	
Int_To_Float_Simple1	Ok	Ok	
Int_To_Float_Simple2	Ok	Ok	
Float_To_Long_Float	Ok	Ok	Ok
Incr_By_Const		Ok	
Angle_Between			
Angle_Between			
User_Rule_2	Ok	Ok	Ok

---

<sup>1</sup>with a tailored strategy

	COLIBRI <sup>-</sup>	COLIBRI	Alt-Ergo
User_Rule_3	Ok <sup>2</sup>	Ok	Ok
User_Rule_4	Ok <sup>2</sup>	Ok	Ok
User_Rule_5	Ok	Ok	Ok
User_Rule_6	Ok <sup>2</sup>	Ok	Ok
User_Rule_7	Ok <sup>2</sup>	Ok	Ok
User_Rule_8	Ok	Ok	Ok
User_Rule_9		Ok	Ok
User_Rule_10		Ok	Ok
User_Rule_11			
User_Rule_12	Ok	Ok	Ok
User_Rule_13	Ok	Ok	Ok
User_Rule_14	Ok	Ok	Ok
User_Rule_15	Ok	Ok	Ok
User_Rule_16		CE	
Polynomial		Ok	Ok

The counter example of `User_Rule_11` uses NaN which are not yet supported in the two provers. The case where NaN is forbidden to appear is `User_Rule_12`.

Adacore is currently modifying the translations of the verification goal to the input of the solvers for benchmarking the new solving techniques on Ada programs. The CEA will do the same on C programs.

---

<sup>2</sup>particular case of monotony for division



---

## Bibliography

- [1] *IEEE standard for binary floating-point arithmetic*. Institute of Electrical and Electronics Engineers, New York, 1985. Note: Standard 754–1985.
- [2] Mohammed Said Belaid, Claude Michel, and Michel Rueher. Boosting local consistency algorithms over floating-point numbers. In *Proceedings of the 18th International Conference on Principles and Practice of Constraint Programming, CP'12*, pages 127–140, Berlin, Heidelberg, 2012. Springer-Verlag.
- [3] Sylvie Boldo. *Deductive Formal Verification: How To Make Your Floating-Point Programs Behave*. Thèse d’habilitation, Université Paris-Sud, October 2014.
- [4] Sylvie Boldo. Stupid is as stupid does: Taking the square root of the square of a floating-point number. In Sergiy Bogomolov and Matthieu Martel, editors, *Proceedings of the 7th and 8th International Workshop on Numerical Software Verification*, volume 317 of *Electronic Notes in Theoretical Computer Science*, pages 50–55, Seattle, WA, USA, April 2015.
- [5] Sylvain Conchon, Mohamed Iguernelala, and Alain Mebsout. A collaborative framework for non-linear integer arithmetic reasoning in Alt-Ergo. In *SYNASC 2013*.
- [6] Bruno Marre and Claude Michel. Improving the floating point addition and subtraction constraints. In *CP 2010*.