Some Formal Tools for Computer Arithmetic: Flocq and Gappa

Sylvie Boldo, Guillaume Melquiond

Some Formal Tools for Computer Arithmetic: Flocq and Gappa

Sylvie Boldo* and Guillaume Melquiond*
* Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, Laboratoire Méthodes Formelles, 91190, Gif-sur-Yvette, France.

***Abstract*—This invited paper presents two tools developed by the authors. Their purpose is to help the user in writing proofs regarding computer arithmetic, *e.g.*, certifying a bound on a round-off error, while aiming at a high level of guarantee. Flocq is a library of mathematical definitions and theorems for the Coq proof assistant; Gappa is meant to compute bounds of values and errors, while producing the corresponding formal proof. We describe here these tools, how they interact and how they fit in a larger verification process.**

***Index Terms*—Floating-point; Round-off error; Formal methods; Coq**

I. Introduction

Programs and circuits for computer arithmetic are far from being exempt of bugs, in part due to the quirkiness of floating-point arithmetic. Thus comes the question of the trust one can put in such algorithms, and how to increase it (*i.e.*, detect bugs before production). There is a wide set of methods that can be put to use, but they can be classified along two axes.

The first axis would characterize the depth of the analysis the developer is willing to perform. At one end, we have the safety bugs (access out of bounds, division by zero, etc). When looking for such bugs, the developer does not have to specify anything about the program or the circuit; tools will work in a fully automated way. In a similar category, but more specific to computer arithmetic, the developer might want to know how the computed value differs from the infinitely-precise value (unstable branching, round-off errors, etc). Then, as we move along this axis, the developer starts to express what the program or circuit is actually supposed to compute (which might not be apparent from the code). This might be a black box such as an MPFR-based function that computes the ideal value, *e.g.*, the limit of a converging sequence. This might also be a white box, such as a reference circuit for an arithmetic block. Eventually, one might need the full language of mathematics to precisely describe the expected properties of the computed values. The more exotic the specification is, the less automated the tools are, to the point where one might have to fall back to a pen-and-paper proof.

The second axis would be the level of trust we end up with. In an ideal world with unlimited resources, one would just validate the program or circuit (or combination thereof) for every input. This is actually possible in some cases, *e.g.*, checking a function with a single *binary32* argument. But in general, we have to turn to other approaches. So, at one end of the axis, we use validation, but by randomly testing a subset of the inputs only. Then some methods make it possible to increase the trust, that is, to reduce the chance bugs occur for some untested inputs. One might use stochastic arithmetic or other fault-injection methods to encompass as many potential behaviors as possible in a single run. One might also guide testing by code coverage or machine learning. But short of testing all the inputs, the trust cannot rise past a certain point. So, instead of validation, one might turn to verification, *i.e.*, mathematically proving that a program or circuit is correct for all the inputs, even those that were not tested. A wide range of approaches can then be put to use: abstract interpretation, model checking, deductive verification, and so on. But we are faced with a dilemma: To which extent can we trust a tool that claims that our algorithm is bug-free? As with the first axis, the further we go along the axis of trust, the more work might be needed from the developer.

In this invited paper, we present our work at the far end of both axes, that is, deductive verification using mechanically-checked proofs. This opens the way to the most expressive specifications with the highest level of trust, as exemplified by the Flocq formalization of floating-point arithmetic for the Coq proof assistant [1]. But this comes at a large verification cost for the user, which we try to alleviate with automated tools like Gappa.

II. Flocq

The starting point of any verification of algorithms in computer arithmetic is the formal definition of all the basic blocks, *i.e.*, formats, rounding, and so on. Flocq [2] is a Coq library developed for this purpose, with enough results so that both automation and hand-writing proofs are possible. We describe here our main design choices. More details can be found in [3].

A. Formats

Let us start with fixed- and floating-point formats, *i.e.*, the sets of representable numbers. At their simplest, they are just subsets of $\mathbb{R}$ (type `R -> Prop` in Coq), as this allows an easy interaction with real numbers. But that is a bit too vague to have meaningful properties, so we constrained them further using a function $\varphi : \mathbb{Z} \to \mathbb{Z}$ and an integer radix β. A real number x belongs to a format characterized by φ when

$$\exists m \in \mathbb{Z}, \quad x = m \times \beta^{\varphi(e_x)},$$

where $e_x = \lfloor \log|x| / \log\beta \rfloor + 1$, so that $|x| \in [\beta^{e_x-1}; \beta^{e_x})$. The equivalent Coq definition is

$$x = \mathrm{Ztrunc}\left(x\beta^{-\varphi(e_x)}\right) \times \beta^{\varphi(e_x)}$$

with $\mathrm{Ztrunc}(m)$ being $\lfloor m \rfloor$ when m is positive and $\lceil m \rceil$ otherwise.

Now let us focus on the format function φ. There are a few requirements for it to characterize a reasonable format (see [2], [3] for details) that are fulfilled by the following examples. The fixed-point format with lsb $e_{\min}$ corresponds to the constant function $\varphi_{\mathrm{FIX}}(e) = e_{\min}$. Then, as expected, a number x is in the format when $\exists m \in \mathbb{Z}, \quad x = m \times \beta^{e_{\min}}$. An unbounded floating-point format with a precision of p digits corresponds to $\varphi_{\mathrm{FLX}}(e) = e - p$. A floating-point format with precision p and a minimal exponent (the smallest positive normalized number is $\beta^{e_{\min}+p-1}$) with gradual underflow corresponds to $\varphi_{\mathrm{FLT}}(e) = \max(e - p, e_{\min})$.

The canonical exponent of the floating-point number (that is $\varphi(e_x)$) allows an easy definition of the ulp, and then the definition and properties of the predecessor and successor. For instance,

$$x \notin F \Rightarrow \triangle(x) = \triangledown(x) + \mathrm{ulp}(x).$$

B. Axiomatic Rounding

A rounding may be defined as a relation between a real and a floating-point value, given a format. For instance, f is the rounding of x toward $-\infty$ in the format F if

$$(f \in F) \wedge (f \leq x) \wedge (\forall g \in \mathbb{R},\ g \in F \Rightarrow g \leq x \Rightarrow g \leq f)\,.$$

Similar definitions exist for rounding toward $+\infty$, to zero, and to nearest. Note that the latter requires a tie-breaking rule. There is a subtlety when breaking ties to even, as it requires that, among two successive floating-point numbers, one and only one is even. It is not the case for instance when using a floating-point format with no subnormals (flush-to-zero) as the successor of 0 is $\beta^{p-1+e_{\min}}$ which has an even mantissa β^{p-1} when the radix β is even. These axiomatic roundings are useful for verifying properties of algorithms.

The IEEE-754 standard, however, follows a different approach to characterize floating-point operators. So, Flocq also expresses rounding as functions from $\mathbb{R}$ to $\mathbb{R}$ that provide results with the aforementioned format properties. For instance, let $e_x = \lfloor \log|x| / \log\beta \rfloor + 1$ as before, then the rounding of x toward $-\infty$ is:

$$\triangledown(x) = \lfloor x\beta^{-\varphi(e_x)} \rfloor \times \beta^{\varphi(e_x)}.$$

Flocq provides all the necessary lemmas to relate the relational and functional points of view. It also provides many lemmas, such as bounds on the error:

$$|\circ^{\tau}(x) - x| \leq \tfrac{1}{2}\mathrm{ulp}(\circ^{\tau}(x))$$

with $\circ^{\tau}$ being a rounding to nearest with an arbitrary tie-breaking rule.

Several theorems of Flocq are dedicated to exact computations, such as Sterbenz' lemma, or when rounding a number already in the format. Here is another example: If two numbers x and y are in an FLT format (format with a fixed precision and a minimal exponent) and $|x + y| \leq \beta^{p+e_{\min}}$, then $x + y$ is a floating-point number in the same format. Similarly, we have proved the existence of error-free transformations for addition and multiplication. But we have also proved that the remainder of division and square root is representable. In other words, assuming x and y are two floating-point numbers in the FLX format (unbounded exponent range) and $\square$ is a rounding function, then

$$x - \square\,(x/y) \cdot y$$

is representable in the FLX format. When $p > 1$,

$$x - \circ^{\tau}\left(\sqrt{x}\right)^2$$

is representable in the FLX format.

A more intricate case of lossless computation is the remainder of the integer division, where we have the following generic theorem. Consider a format described by a monotone function φ, a valid integer rounding $rnd : \mathbb{R} \to \mathbb{Z}$, and two numbers x and y representable in the format such that $|x/y| < \frac{1}{2} \Rightarrow rnd(x/y) = 0$ (which holds whenever rnd rounds toward zero or to nearest). Then $x - rnd\,(x/y) \cdot y$ is representable in the format.

Flocq also characterizes cases where double rounding is innocuous [4], from which one can get for example a property of double rounding for floating-point division. Consider two FLT formats with precisions p and p' and minimal exponents $e_{\min}$ and $e_{\min}{}'$ such that $e_{\min}{}' \leq e_{\min} - p - 2$ and $2p \leq p'$, and two arbitrary tie-breaking rules. Consider x and $y \neq 0$ in the smaller FLT format $(p, e_{\min})$, then

$$\circ_p^{\tau_1}\left(\circ_{p'}^{\tau_2}(x/y)\right) = \circ_p^{\tau_1}(x/y)\,.$$

This easily proves that double rounding for division is innocuous in the case of *binary64* and *binary32*.

Flocq provides several other theorems about double rounding: accuracy, faithfulness, and, when the first rounding is to odd, correctness. In the context of the CompCert C compiler, where the semantics of floating-point operations is built on top of Flocq, these theorems made it possible to formally verify the correctness of the generated assembly code that converts integers to floating-point numbers and *vice versa* [5].

C. Effective Computations

The previous rounding functions are quite abstract, as they are meant to round arbitrary real numbers. This is useful in proofs, but not so much for performing computations. When the inputs of a floating-point operation are numbers of the form $m \cdot \beta^e$, we can write actual algorithms and even perform effective computations inside the logic of Coq. Flocq provides verified operators for floating-point addition, subtraction, multiplication, but also division and square root.

This is especially useful in the context of the CompCert compiler, where the effective operators are used for parsing and output of floating-point literals and for performing optimizations such as constant propagation [5]. These operators also serve as a reference implementation in the CoqInterval library, making it possible to perform proofs by computing with arbitrary-precision floating-point numbers inside Coq [6]. They are also the basis for performing native *binary64* computations inside formal proofs [7].

III. GAPPA

Gappa is a tool meant to help the user analyze and verify fixed- and floating-point algorithms [8]. Given a logical formula over real numbers, it tries to fill any hole (*e.g.*, a bound on some round-off error) and generates a formal proof of the resulting proposition. Without losing much generality, a formula given to Gappa can be interpreted as follows:

$$\forall x_1, \ldots, x_k \in \mathbb{R},\ e_1 \in I_1 \wedge \ldots \wedge e_n \in I_n \Rightarrow e \in ?.$$

The expressions $e_1, \ldots, e_n, e$ are arithmetic over some real numbers $x_1, \ldots, x_k$. As in Flocq, these expressions can also mention rounding operators, so as to express properties of floating-point numbers. So, given some interval enclosures of $e_1, \ldots, e_n$, Gappa computes an enclosure of e (or verify it if given by the user) and generates a proof of it, which can be checked using Coq.

While only simple enclosures are usually encountered in user propositions, Gappa internally supports a wider range of properties over real numbers. For example, a proof might critically rely on the fact that some real number is a multiple of a given power of two or that it does not need more than a given number of bits to be represented. These properties are supported by the following predicates:

$$\begin{aligned} \mathrm{FIX}(x, n) &\triangleq \exists m \in \mathbb{Z},\ x = m \cdot 2^n, \\ \mathrm{FLT}(x, n) &\triangleq \exists m, e \in \mathbb{Z}, x = m \cdot 2^e \wedge |m| < 2^n. \end{aligned}$$

While Gappa's FIX predicate is related to Flocq's FIX format, Gappa's FLT predicate is actually related to Flocq's FLX format, due to historical reasons. To express the FLT format of Flocq, one should instead mix Gappa's FLT (for the floating-point precision) and Gappa's FIX (for the minimal exponent).

Another important predicate is REL. Indeed, while the absolute error between $\tilde{x}$ and x can easily be expressed as an enclosure $\tilde{x} - x \in ?$, the relative error $(\tilde{x} - x)/x$ is a bit more tedious to handle due to a potential division by zero. The REL relation avoids this division as follows:

$$\mathrm{REL}(\tilde{x}, x, I) \triangleq \tilde{x} = x \cdot (1 + \varepsilon) \wedge \varepsilon \in I.$$

Gappa contains a database of theorems about the above predicates (and a few others, less important). For example, theorems inspired from interval arithmetic tell the tool how to combine two properties $u \in I$ and $v \in J$ to obtain a new property $u \diamond v \in ?$. Gappa saturates over its database of theorems, starting from the properties $e_1 \in I_1, \ldots, e_n \in I_n$ and computing until it can no longer deduce anything new. At that point, it returns the best enclosure of e it found. It also keeps track of all the theorems that lead to this enclosure and generates a formal proof from them.

A special case of the above process is when Gappa encounters a contradiction, *e.g.*, an expression is proved to be enclosed in two disjoint intervals. In that case, anything can be deduced, so Gappa directly returns a formal proof. This might happen when an algorithm contains multiple conditionals and the conjunction of some of them is impossible. This might also happen if the user has explicitly provided an enclosure $e \in I$ rather than letting I unspecified. Indeed, in that case, Gappa assumes that $e \notin I$ holds and tries to derive a contradiction, which might be faster than trying to blindly prove $e \in I$.

Now let us go back to the database of theorems. They are meant to emulate the kind of reasoning a user would perform in a pen-and-paper proof. As mentioned earlier, some theorems are inspired by interval arithmetic; from some enclosures of two expressions u and v, they are able to deduce an enclosure of $u \diamond v$. We denote such a theorem as follows:

$$\mathrm{BND}(u) \wedge \mathrm{BND}(v) \Rightarrow \mathrm{BND}(u \diamond v).$$

where intervals are hidden for the sake of readability. Similarly, we will just omit the last argument of the predicates FIX, FLT, and REL, when mentioning their theorems. For example, Gappa knows how to compose relative errors:

$$\mathrm{REL}(u, v) \wedge \mathrm{REL}(v, w) \Rightarrow \mathrm{REL}(u, w).$$

It also knows how the relative error behaves when expression are multiplied and divided. For summed expressions, the behavior of relative errors is slightly more complicated, as it depends on potential cancellations:

$$\mathrm{REL}(\tilde{u}, u) \wedge \mathrm{REL}(\tilde{v}, v) \wedge \mathrm{BND}(u/(u+v)) \Rightarrow \mathrm{REL}(\tilde{u}+\tilde{v}, u+v).$$

This is not the only theorem mixing different predicates. For example, when both an enclosure and a power-of-two divisor are known, Gappa can deduce how many bits are needed for an expression:

$$\mathrm{BND}(u) \wedge \mathrm{FIX}(u) \Rightarrow \mathrm{FLT}(u).$$

In addition to the theorems about arithmetic operators, there are also theorems about rounding operators. Let us denote $\circ(u)$ the rounding of u to a given format. Gappa knows how to deduce $\mathrm{FIX}(\circ(u))$, $\mathrm{FLT}(\circ(u))$, $\mathrm{BND}(\circ(u)-u)$, $\mathrm{REL}(\circ(u), u)$, from other facts, *e.g.*, $\mathrm{BND}(|u|)$.

There are also numerous theorems that are just consequences of equality between expressions. For example, while we already mentioned a dedicated theorem for the relative error of the sum, Gappa reasons about the absolute error of the sum using the following equality:

$$(\tilde{u} + \tilde{v}) - (u + v) = (\tilde{u} - u) + (\tilde{v} - v).$$

When looking for an enclosure of the left-hand side, Gappa first tries to obtain an enclosure of the right-hand side.

These equalities are also the way the user can bring some extra intelligence to the tool. Indeed, while Gappa knows about 200 theorems, some algorithms might still elude it. For example, consider a Newton iteration for computing the multiplicative inverse of a:

$$\begin{aligned} \delta_n &\leftarrow \circ(1 - a \cdot y_n), \\ y_{n+1} &\leftarrow \circ(y_n + y_n \cdot \delta_n). \end{aligned}$$

Gappa does not know that such an iteration converges quadratically. So, the user can help the tool by providing the following equality as a premise of the logical formula:

$$(\bar{y}_{n+1} - a^{-1})/a^{-1} = -\varepsilon_n^2$$

with $\bar{y}_{n+1} = y_n + y_n \cdot (1 - a \cdot y_n)$ and $\varepsilon_n = (y_n - a^{-1})/a^{-1}$. Note that this equality mathematically holds since $\bar{y}_{n+1}$ contains no rounding operator; it can be easily verified using any computer algebra system (or Coq). Thanks to its known theorems and this user equality, Gappa is able to relate y_{n+1} to $\bar{y}_{n+1}$ and then to deduce the relative error between y_{n+1} and a^{-1} from the relative error between y_n and a^{-1}.

Gappa can be used either as a standalone tool, *e.g.*, to analyze some algorithm by filling holes ⑦ in logical formulas describing it. It can also be invoked directly from Coq as a tactic to automatically and formally discharge goals. In that case, the logical formulas contain no hole.

IV. Conclusion

We have shown two tools that help the user design algorithms and prove their correctness with the highest level of trust. Formal methods, and in particular the Coq proof assistant, play a crucial role. Proofs may be done automatically when possible and interactively when needed (especially when the precision is generic).

We have proved basic blocks with several kind of correctness specification. For instance, we have proved round-off error bounds for algorithms computing an order-2 discriminant and the area of a triangle. We have verified the correctness of algorithms that compute error-free transformations: FastTwoSum and TwoSum, which compute the error of an floating-point addition, as well as an algorithm that computes the error of an FMA.

We have also verified an integer division based on floating-point computations, and a correctly-rounded average. For all these examples and many more, we refer the reader to [3] for details.

A larger example comes from applied mathematics: a simple numerical scheme (the three-point scheme) that solves the 1D wave equation. Its verification relies on Gappa for overflow proofs and for bounding the round-off error of one step while Flocq is used to prove the propagation and partial compensation of these errors and to guarantee a small final round-off error. This example also shows that the correctness of a program might not lie only in its accuracy. Sometimes, mathematics come into play and we need results from analysis for a full correctness. It may be a Taylor expansion for approximations or the stability of a numerical scheme. If one wants to keep the highest level of guarantee, then mathematics need to be formally verified too. We have developed appropriate Coq libraries and tactics for that purpose. First, Coquelicot [9] is a library for (mostly real) analysis, more convenient than Coq's standard library; it comes with a few tactics for derivative and integration. CoqInterval [6] is a Coq tactic that automatically proves inequalities on real numbers, relying both on interval arithmetic and Taylor models.

Acknowledgments

This project has received funding from the European Commission under the Horizon 2020 research and innovation programme Grant agreement N°810367. This work was supported by the NuSCAP (ANR-20-CE48-0014) project of the French national research agency (ANR).

References

[1] The Coq Development Team, *The Coq Proof Assistant Reference Manual v8.12*, 2020.

[2] S. Boldo and G. Melquiond, "Flocq: A unified library for proving floating-point algorithms in Coq," in *20th IEEE Symposium on Computer Arithmetic*, E. Antelo, D. Hough, and P. Ienne, Eds., Tübingen, Germany, Jul. 2011, pp. 243–252.

[3] ——, *Computer Arithmetic and Formal Proofs: Verifying Floating-point Algorithms with the Coq System*. ISTE Press - Elsevier, Dec. 2017.

[4] P. Roux, "Innocuous double rounding of basic arithmetic operations," *Journal of Formalized Reasoning*, vol. 7, no. 1, pp. 131–142, 2014.

[5] S. Boldo, J.-H. Jourdan, X. Leroy, and G. Melquiond, "Verified compilation of floating-point computations," *Journal of Automated Reasoning*, vol. 54, no. 2, pp. 135–163, 2015.

[6] É. Martin-Dorel and G. Melquiond, "Proving tight bounds on univariate expressions with elementary functions in Coq," *Journal of Automated Reasoning*, vol. 57, no. 3, pp. 187–217, 2016.

[7] G. Bertholon, É. Martin-Dorel, and P. Roux, "Primitive floats in Coq," in *10th International Conference on Interactive Theorem Proving*, Portland, OR, USA, Sep. 2019.

[8] F. de Dinechin, C. Lauter, and G. Melquiond, "Certifying the floating-point implementation of an elementary function using Gappa," *Transactions on Computers*, vol. 60, no. 2, pp. 242–253, 2011.

[9] S. Boldo, C. Lelay, and G. Melquiond, "Coquelicot: A user-friendly library of real analysis for Coq," *Mathematics in Computer Science*, vol. 9, no. 1, pp. 41–62, 2015.