Flocq: A Unified Library for Proving Floating-point Algorithms in Coq

Sylvie Boldo, Guillaume Melquiond

Flocq: A Unified Library for Proving Floating-point Algorithms in Coq

Sylvie Boldo
INRIA Saclay - Île-de-France, ProVal, Orsay, F-91893
LRI, Univ Paris-Sud, CNRS, Orsay, F-91405
Email: Sylvie.Boldo@inria.fr

Guillaume Melquiond
INRIA Saclay - Île-de-France, ProVal, Orsay, F-91893
LRI, Univ Paris-Sud, CNRS, Orsay, F-91405
Email: Guillaume.Melquiond@inria.fr

***Abstract*—Several formalizations of floating-point arithmetic have been designed for the Coq system, a generic proof assistant. Their different purposes have favored some specific applications: program verification, high-level properties, automation. Based on our experience using and/or developing these libraries, we have built a new system that is meant to encompass the other ones in a unified framework. It offers a multi-radix and multi-precision formalization for various floating- and fixed-point formats. This fresh setting has been the occasion for reevaluating known properties and generalizing them. This paper presents design decisions and examples of theorems from the Flocq system: a library easy to use, suitable for automation yet high-level and generic.**

***Index Terms*—Floating-point arithmetic; formal proof system; program verification.**

I. Introduction

Modern floating-point arithmetic has been widely available for the past thirty years and is by far the most used approach for approximating computations on real numbers. While making it an efficient arithmetic, limits on the precision and exponent range of floating-point numbers are often a cause of unexpected behaviors. Testing of algorithms, software, and hardware, alleviate this issue. Unfortunately, exhaustive testing is usually out of reach and there is no guarantee that all the troublesome inputs have been uncovered.

A higher level of safety can be reached by formal methods in addition to testing. In the broader sense, formal methods cover any approach that provides a mathematical certificate of correctness. The mathematical foundations for such a certificate are given by the IEEE-754 standard [1] which precisely describes the formats and operations. Among formal methods, we find model checking, satisfiability, temporal logic, abstract interpretation, and so on. These approaches are automated and mostly useful on big applications but simple from a numerical point of view.

We are mostly interested in small numerical devices (be they algorithms, programs, microcodes, and so on) that implement state-of-the-art results. Fully automated approaches are useless in this case, so we are left with interactive theorem provers, such as ACL2, Coq, HOL Light, PVS, and so on. They allow the user to guide the reasoning manually, but only experts are able to prove useful properties in a reasonable amount of time. It is therefore important to provide simple yet powerful formalisms to promote the usage of formal methods and to improve device safety. Bigger systems may not be amenable to a full formal proof but this approach can be applied for critical parts or test cases.

We can distinguish several motivations for the many existing formalizations of floating-point arithmetic. Some of them focus only on formally describing the content of the IEEE-754 and -854 standards for Z [2], for PVS [3], for Coq [4]. Others are intended for low-level implementations, *e.g.* guaranteeing that a circuit or microcode implements a given arithmetic operator for ACL2 [5], [6], for Forte [7], for PVS [8]. Finally, there are some formalizations dedicated to higher-level properties and algorithms for HOL Light [9], [10], for Coq [11], [12]. To ease the burden of these kinds of proofs, some libraries have focused on automation for Coq [13], [14].

We are mainly interested in a formalization of floating-point arithmetic for the Coq proof assistant. As detailed in Section II, there are already several of them with various motivations. As long as the focus is on a particular domain, one can find a suitable library among the existing ones. Unfortunately, the process of formally certifying programs has shown that it was useful to mix several formalisms [15]. For instance, automation is necessary for dealing with non-overflow proofs and propagation of rounding errors, while a rich formalism is needed for expressing subtle properties such as error-free computations. The usage of several formalisms in a single proof is painful and time-consuming, since it requires one to prove equivalence lemmas between them and to invoke these lemmas repeatedly. That was our motivation: a single library that encompasses several aspects of the existing formalisms.

We have developed the Flocq library available at

`http://flocq.gforge.inria.fr/`

It provides a generic framework: multiple radixes (2, 10, but also odd radixes), multiple precision, several formats for both fixed-point arithmetic and floating-point arithmetic, various rounding modes, and so on (Section III). Special care was taken so that nonstandard arithmetics like flush-to-zero were natively supported. On top of this core library, we have built computable operators to ease automation (Section IV-A). We have also proved several high-level theorems (Section IV-B) that are sometimes improved versions of the theorems found in the existing libraries where they were proved for a single family of formats only.

This work was funded by the Fφst (ANR-08-BLAN-0246-01) project of the French national research organization (ANR).

II. Previous libraries in Coq

The Flocq library is based on our experience using and/or developing floating-point libraries for Coq. This section will describe three of our main sources of inspiration.

A. PFF

PFF is a Coq library initially developed in [11] and various results have been added since then [16], [17], [12]. It has been purely designed to be a high-level formalization of IEEE-754: only floating-point formats with gradual underflow are supported. This is expressed by a formalization where floating-point numbers are pairs (n, e) associated with real values $n \times \beta^e$. The requirements for a number to be in the format $(e_{\min}, \beta^p)$ are:

$$|n| < \beta^p \quad \wedge \quad e_{\min} \leq e.$$

As there may be several bounded floating-point numbers with the same value, PFF defines a canonical representative when needed. Still, this cohort of equal representatives is sometimes useful as in [17].

A first advantage of this library is its genericity: the radix is any integer greater than 1 and the precision is any positive integer (it may be 1 in some theorems). It is designed to be as generic as possible in that respect.

As for rounding, the choice was to use an axiomatic approach: there is no function that computes a rounded value, but there is a relation between a real value and a floating-point number that says that this floating-point number is a rounding down (up, to nearest, to nearest ties to even…) of the real value. It is practical for proving high-level properties. For example, we can prove theorems for any rounding to nearest, whatever the tie: we simply state that the floating-point number is one of the possibly two roundings to nearest of the real value. A drawback is the lack of automation: to prove that a floating-point number is correctly rounded is very long and tedious.

This formalization is efficient for high-level algorithms containing floating-point technical points. It is convenient for human interactive proofs as shown by the many proofs based on it. The high number of lemmas (about 1400) makes it suitable for a large range of applications.

B. Gappa

Gappa is a tool for automatically proving mathematical properties related to numerical codes [14], [15]. These properties deal with real numbers, but they can tackle floating- and fixed-point arithmetic through the usage of rounding operators, which are functions from $\mathbb{R}$ to $\mathbb{R}$. Indeed, the IEEE-754 standard states that "each of the computational operations that return a numeric result specified by this standard shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then rounded that intermediate result, if necessary, to fit in the destination's format" [1]. So the result of a floating-point adder, assuming it is not an exceptional value, is just the combination of the real addition with a rounding operator.

When Gappa proves that a property is true, it also generates a formal proof of it. This certificate can be mechanically checked by Coq, and it relies on a support library that contains all the basic facts, *e.g.* the result of a rounding to binary double precision can be represented with a mantissa of at most 53 bits. Gappa has been designed for radix-2 arithmetic; therefore, that is the only radix the support library has theorems for.

So that generated certificates can be checked without any human intervention, they rely on computations performed inside the proof checker. In particular, given a number $m \cdot 2^e$, the support library is able to correctly round it to a number $m' \cdot 2^{e'}$ that fits in a given format. The library is not restricted to some specific formats; it can handle any floating- or fixed-point format that can be described by an integer relation between exponents. This mechanism has been reused for Flocq and is described in more details in Section III-D.

To summarize, Gappa's support library is a complete formalization of binary arithmetics. It also provides some computable operators for performing rounding. But it is designed for a usage by proofs generated automatically, so its theorems have awkward statements that are not really suitable for interactive proofs.

C. Coq.Interval

The Coq.Interval library [18] provides a tool based on interval arithmetic and Taylor series for automatically proving inequalities on real-valued expressions inside Coq. The methods are well-known, if not outdated, but their primary characteristic is that they are performed inside the logic formalism of the proof checker. The key concept is that the whole formalization has to be computable, in order for the interval operators to return actual results. They rely on a computable floating-point arithmetic that is formalized in Coq [13]. It supports a single family of formats: multi-precision with unlimited exponent range. From an interface point of view, the library is therefore similar to MPFR [19], but it supports any radix.

Unlike Gappa, there is no external oracle for precomputing arithmetic results this time. So Coq.Interval not only provides computable rounding operators but also any operator the interval methods may need: division, square root, some elementary functions, and so on. Some of these algorithms have been reimplemented in Flocq and are described in Section IV-A.

Since this floating-point kernel is primarily intended for interval arithmetic, rounding modes other than directed toward infinities have been neglected in the formalization. Moreover, the library does not provide many facts about floating-point arithmetic, except its own proofs of correctness.

III. Core library

In this section, we will describe the core of the library, namely the main definitions and choices of representation.

A. Rounding predicates

Here, we consider the floating-point numbers as any subset $\mathbb{F}$ of the real numbers. This means there is no requirement

that $\mathbb{F}$ be discrete or that $0 \in \mathbb{F}$. There will be two definitions of the rounding modes. The first one, described here, is that of Section II-A: rounding modes are relations between real numbers and their rounded values. The second definition, based on computable functions, will be described in Section III-E but it does not apply to the most exotic formats.

A rounding predicate Q has the Coq type $\mathbb{R} \to \mathbb{R} \to \mathrm{Prop}$, meaning it is a relation between two real numbers (as $\leq$ or $=$). For it to be a rounding predicate, it must fulfill two requirements. First, it must be total (any real can be rounded): $\forall x \in \mathbb{R},\ \exists f \in \mathbb{R},\ Q(x, f)$. Second, it must be monotone: $\forall x, y, f, g \in \mathbb{R},\ Q(x, f) \Rightarrow Q(y, g) \Rightarrow x \leq y \Rightarrow f \leq g$ (that is to say nondecreasing). These two properties are enough for a rounding predicate to have reasonable properties. For example, those properties imply the uniqueness of rounding: two reals that are roundings of the same value are equal.

A more interesting property is the fact that, from any rounding predicate Q, we can construct a function r such that for all real x, $Q(x, r(x))$.

We can then define the common rounding modes by their mathematical properties as in Section II-A for a given $\mathbb{F}$:

- Rounding toward $-\infty$ is denoted by $\bigtriangledown(x, f)$ and defined by $f \in \mathbb{F} \wedge f \leq x \wedge (\forall g \in \mathbb{F}, g \leq x \Rightarrow g \leq f)$.
- Rounding toward $+\infty$ is denoted by $\bigtriangleup(x, f)$ and defined by $f \in \mathbb{F} \wedge x \leq f \wedge (\forall g \in \mathbb{F}, x \leq g \Rightarrow f \leq g)$.
- Rounding toward 0 is defined by $(0 \leq x \wedge \bigtriangledown(x, f)) \vee (x \leq 0 \wedge \bigtriangleup(x, f))$.

It is easy to prove that these directed modes are monotone and that a rounded value is unique (if $\bigtriangleup(x, f_1)$ and $\bigtriangleup(x, f_2)$ then $f_1 = f_2$). But what about the existence of these values?

For instance, if 0 is not in $\mathbb{F}$, strange things happen: consider for example $\mathbb{F} = \{n \times 2^{e_{\min}} \text{ with } n \in \mathbb{Z}^*\}$ in radix-2. Then $0 \notin \mathbb{F}$ and if we round down the positive value $2^{e_{\min}-1}$, we get the negative value $-2^{e_{min}}$, and $-2^{e_{\min}-1}$ is rounded up to $2^{e_{min}}$. So such a rounding toward zero would not be monotone.

The set $\mathbb{F}$ must also be such that we are able to round down: $\forall x \in \mathbb{R},\ \exists f \in \mathbb{R},\ \bigtriangledown(x, f)$. This is needed as roundings do not exist for any $\mathbb{F}$: consider for example the set of rational numbers. Then an irrational number x cannot be rounded: there is always another rational that is smaller than x and nearer to x.

At last, we assume the symmetry of $\mathbb{F}$. This is not a necessary condition, but it is reasonable and greatly helps the following proofs. The idea is that, if we know how to round down, a symmetric set allows us to round up for free: $\bigtriangledown(x, f) \Leftrightarrow \bigtriangleup(-x, -f)$.

Theorem 1 (satisfies_any_imp_{DN,UP,ZR}) *For any set $\mathbb{F}$ such that $0 \in \mathbb{F}$, $\mathbb{F}$ is symmetrical, and rounding to $-\infty$ exists, then rounding toward $-\infty$, toward $+\infty$, and toward zero are rounding predicates (they are total and monotone).*

B. Rounding to nearest

The most used (and default mode) is rounding to nearest, ties to even (when in the middle, choose the one with the even mantissa). But the revision of the IEEE-754 standard [1] added the rounding to nearest, ties to away (when in the middle, choose the one with the biggest absolute value). With these two roundings having much in common, it is natural to define a generic rounding to nearest, whatever the tie-breaking rule. So we first define rounding to nearest without ties by $f \in \mathbb{F} \wedge (\forall g \in \mathbb{F}, |f - x| \leq |g - x|)$ and denote it by $\circ(x, f)$.

This rounding to nearest does not have a tie-breaking rule: it is not equivalent to a function as two different real numbers may be correct roundings to nearest of the same real (when it is the middle of two floating-point values). For ties away from zero, the definition is easy: $\circ(x, f) \wedge (\forall g, \circ(x, g) \Rightarrow |g| \leq |f|)$.

For a more generic tie-breaking rule, we define a rounding to nearest that depends on a property P. This generic rounding to nearest with ties is denoted by $\circ^P(x, f)$ and defined by $\circ(x, f) \wedge (P(x, f) \vee \forall g, \circ(x, g) \Rightarrow f = g)$. This means that either f has the property P, or it is the only rounding to nearest. You may think of P as being "f is even" or $|f| \geq |x|$. This generic rounding is unique and monotone under the following assumption on P: if $\bigtriangledown(x, d)$ and $\bigtriangleup(x, u)$ and $\circ^P(x, d)$ and $\circ^P(x, u)$ and $P(x, d)$ and $P(x, u)$, then we have $d = u$. We also proved that the symmetry of P implies the symmetry of $\circ^P$ and that rounding to nearest, ties away from zero is exactly this generic rounding to nearest with $P(x, f) \stackrel{def}{=} |f| \geq |x|$.

The most difficult point is of course the existence of rounding to nearest. As far as rounding to nearest without ties is concerned, the existence is guaranteed as soon as the three properties of the preceding subsection are fulfilled (but it is not a rounding mode as it is not monotone). For the generic rounding to nearest, we need this property on P: if $x \notin \mathbb{F}$ and $\bigtriangledown(x, d)$ and $\bigtriangleup(x, u)$, then we either have $P(x, d)$ or $P(x, u)$. If this last property (and the three preceding ones) are fulfilled, then $\circ^P$ is a rounding mode. We easily deduce that the rounding to nearest, ties away from zero is a rounding mode.

For rounding to nearest ties to even, we cannot define it here: the reason is that we cannot decide if a value is even or odd. Let us consider $x = 2^e \in \mathbb{F}$: is it odd as $x = 1 \times 2^e$ or is it even as $x = 2 \times 2^{e-1}$? We need both a unique canonical representation of a floating-point number and a definition of ties to even. The representation will be defined in Section III-D and the rounding will be defined in Section III-E.

C. Floating-point numbers

Now, let us define the most useful formats. They will be defined differently in the next section, but we begin with the most intuitive definitions. We consider floating-point numbers as a mantissa and an exponent. The mantissa is shifted in order to get an integer rather than a fixed-point value. A floating-point number f is then a pair of integers (n_f, e_f) and its value is $\mathrm{F2R}(f) = n_f \cdot \beta^{e_f}$, where β, the radix, is an integer greater than one.

From that, we can define a format as a set of real numbers corresponding to all the expected floating-point numbers. For example, the set of fixed-point numbers with exponent $e_{\min}$ is the set of real numbers such that there exists a floating-point number with exponent $e_{\min}$ having this value. Note that a format is therefore a subset of $\mathbb{R}$ such that there exists a floating-point number equal to the real that has such and such property.

Here are the common formats: fixed-point (FIX), floating-point with unbounded exponents (FLX), normalized floating-point with unbounded exponents (FLXN), floating-point with gradual underflow (FLT), and floating-point with flush-to-zero (FTZ). The array below formally defines these formats.

Format	is defined by $\exists f,\ \mathrm{F2R}(f) = x\ \wedge \ldots$
$\mathrm{FIX}_{e_{\min}}(x)$	$e_f = e_{\min}$
$\mathrm{FLX}_{p}(x)$	$\lvert n_f\rvert < \beta^p$
$\mathrm{FLXN}_{p}(x)$	$x \neq 0 \Rightarrow \beta^{p-1} \leq \lvert n_f\rvert < \beta^p$
$\mathrm{FLT}_{p,e_{\min}}(x)$	$e_{\min} \leq e_f \wedge \lvert n_f\rvert < \beta^p$
$\mathrm{FTZ}_{p,e_{\min}}(x)$	$x \neq 0 \Rightarrow e_{\min} \leq e_f \wedge \beta^{p-1} \leq \lvert n_f\rvert < \beta^p$

To ease further proofs, we proved that the FLX and FLXN formats are equivalent.

These are high-level definitions, and the FLT format is exactly the format used in PFF (see Section II-A). Nevertheless, these are not the definitions we will mainly use: we will base the format on the definition of a φ function and prove the equivalence between both definitions (see Section III-D).

D. Generic format

Representing formats as predicates and rounding modes as relations on real numbers makes it simple to manipulate them when rounded values are known beforehand. But they are a hindrance when a proof actually needs to compute a rounded value. Therefore we have chosen a more computational approach for another representation of formats and rounding modes.

In order to get closed formulas for rounded values, we had to put constraints on the number sets. All the formats of this family (called *generic format*) satisfy the following two main properties: they contain only floating-point numbers $m \cdot \beta^e$ and all the representable numbers in a given slice are equally distributed.

The slice of a real number x is given by its discrete β-logarithm $e = \mathrm{slice}(x)$:

$$\beta^{e-1} \leq |x| < \beta^e.$$

$\mathbb{F}_\varphi$ is entirely described by a function $\varphi : \mathbb{Z} \to \mathbb{Z}$ that transforms numbers' discrete logarithms into canonical exponents for this format. In other words, a number x is in $\mathbb{F}_\varphi$ if and only if it can represented as a floating-point number $m \cdot \beta^{\varphi(\mathrm{slice}(x))}$. Figure 1 shows the graph of the canonical exponents for the three usual families of floating-point formats.

More precisely, the canonical exponent of a real number x is $\mathrm{cexp}(x) = \varphi(\mathrm{slice}(x))$. Its scaled mantissa is $\mathrm{smant}(x) = x \cdot \beta^{-\mathrm{cexp}(x)}$. And the format $\mathbb{F}_\varphi$ is defined as the set of real numbers x such that

$$x = \mathcal{Z}(\mathrm{smant}(x)) \cdot \beta^{\mathrm{cexp}(x)}$$

with $\mathcal{Z}$ the integer part (rounded toward zero).

The above definition is equivalent to saying that $\mathrm{smant}(x)$ is exactly an integer; but stating it as a rewriting rule makes it a bit simpler to use in Coq. From there, one can deduce that generic formats contain zero and are symmetric. A slice e contains representable numbers, *e.g.* β^{e-1}, when $\varphi(e) < e$.

In order to show that $\mathbb{F}_\varphi$ is a suitable format as described in Section III-A, there is a property left to satisfy: any real number should have a rounded-down value. This value can be chosen as $\bigtriangledown(x) = \lfloor \mathrm{smant}(x) \rfloor \cdot \beta^{\mathrm{cexp}(x)}$, but we have to prove that this function is both increasing and onto $\mathbb{F}_\varphi$. Note that the function is the identity for values of $\mathbb{F}_\varphi$, by definition of $\mathbb{F}_\varphi$. Similarly, the rounded-up value would ideally be $\bigtriangleup(x) = \lceil \mathrm{smant}(x) \rceil \cdot \beta^{\mathrm{cexp}(x)}$. What are the constraints on function φ so that these functions are proper rounding?

First, let us consider the slice $[\beta^{e-1}, \beta^e)$. Let us assume that $\varphi(e) < e$, so that the slice contains some representable numbers. Ideally, a real number $x = \beta^e - \varepsilon$ (with ε positive but negligible) should be rounded up to $\bigtriangleup(x) = \beta^e$, but this power might not be representable since it lies in a different slice $(e+1)$. In order to ensure that it is representable, we put the following constraint on φ:

$$\varphi(e) < e \Rightarrow \varphi(e+1) \leq e.$$

Second, let us consider a real number x inside a slice e such that $e \leq \varphi(e)$. We have $\bigtriangledown(x) = 0$ and $\bigtriangleup(x) = \beta^{\varphi(e)}$. Zero is representable, but $\beta^{\varphi(e)}$ might not be. Moreover, all the other real numbers inside the open interval $(0, \beta^{\varphi(e)})$ have to round down to 0 and up to $\beta^{\varphi(e)}$ too; otherwise the two functions would not be increasing. This requires the following constraint on φ:

$$e \leq \varphi(e) \Rightarrow \begin{cases} \varphi(\varphi(e)+1) \leq \varphi(e), \\ \forall e',\ e' \leq \varphi(e) \Rightarrow \varphi(e') = \varphi(e). \end{cases}$$

Above constraints are necessary; but we have also proved that they are sufficient for $\bigtriangledown$ and $\bigtriangleup$ to be rounding functions on $\mathbb{F}_\varphi$. If φ satisfies these constraints, it is said to be valid (predicate *valid_exp*). This is true for all the usual formats and we will give the corresponding φ functions below. The formats described by these functions are proved to be equivalent to the formats of Section III-C.

A fixed-point format containing all the multiples of $\beta^{e_{\min}}$ is described by

$$\mathrm{FIX\_exp}(e) = e_{\min}.$$

An unbounded floating-point format with a precision of p digits is described by

$$\mathrm{FLX\_exp}(e) = e - p.$$

Floating-point formats with precision p and bounded exponents (the smallest positive normalized number is $\beta^{e_{\min}+p-1}$)

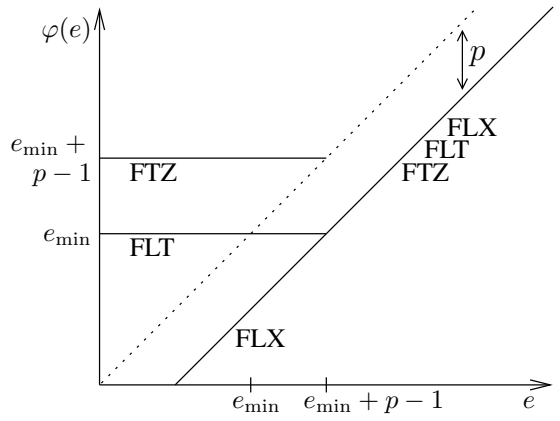

Fig. 1. Values of φ for formats FLX, FLT, and FTZ, with precision p. These functions are the same for normal numbers ($e \geq e_{\min} + p$), but they diverge for subnormal numbers. In particular, the φ function for FTZ is discontinuous.

depend on whether they support subnormal numbers:

$$\begin{aligned} \mathrm{FLT\_exp}(e) &= \max(e-p, e_{\min}), \\ \mathrm{FTZ\_exp}(e) &= \begin{cases} e-p & \text{if } e-p \geq e_{\min}, \\ e_{\min}+p-1 & \text{otherwise.} \end{cases} \end{aligned}$$

The *ulp* (unit in the last place) of a real number x is defined as $\beta^{\mathrm{cexp}(x)}$. This function is partial: it is not defined for 0 (neither was cexp). But it is defined for any other real, be it in $\mathbb{F}_\varphi$ or not. An immediate property of ulp is

Theorem 2 (ulp_DN_UP) *Assuming that φ is valid,*

$$\forall x \notin \mathbb{F}_\varphi,\ \triangle(x) = \bigtriangledown(x) + \mathrm{ulp}(x).$$

A generic format is said to not have the flush-to-zero property if $\mathrm{ulp}(x)$ is representable in $\mathbb{F}_\varphi$ for any real number x. The corresponding property on φ is predicate *not_FTZ_prop*:

$$\forall e \in \mathbb{Z},\ \varphi(\varphi(e+1)) \leq \varphi(e).$$

E. Rounding operators for generic formats

The expressions of the $\bigtriangledown$ and $\triangle$ rounding operators above can be generalized by using a function $\mathrm{Zrnd} : \mathbb{R} \to \mathbb{Z}$:

$$\mathrm{round}_\varphi(x) = \mathrm{Zrnd}(\mathrm{smant}(x)) \cdot \beta^{\mathrm{cexp}(x)}.$$

For round_φ to be a rounding mode, Zrnd has to be the identity for integer inputs and to be increasing on $\mathbb{R}$. That way, rounding modes are given by closed formulas instead of relations. Due to the monotonicity requirement, this is more than being a faithful rounding as defined by Priest [20].

The expression above is not the most generic, since $\mathrm{Zrnd}(x)$ has only access to the scaled mantissa and not the original real number. As a consequence, round_φ will behave in a similar way for all the slices that have the same precision. An additional argument could have been used for Zrnd, *e.g.* the canonical exponent, but we did not find a reasonable setting that was worth the noise.

As shown previously, choosing $\mathrm{Zrnd}(m) = \lfloor m \rfloor$ (resp. $\lceil m \rceil$) gives an operator that rounds downward (resp. upward). We have proved that they are equivalent to the rounding relations of Section III-A.

Rounding to nearest (with tie breaking to even) is not much harder to define: the scaled mantissa m just has to be rounded to the nearest integer, tie breaking to the even one. While the definition is simple, one still has to prove that the resulting operator satisfies the expected property on parity. Indeed, even if the integer is even, the scaled mantissa of the rounded value might be odd if it lies in a different slice. To avoid this parity discrepancy, either the radix should be odd, or the following properties should hold in addition to the previous constraints on φ:

$$\begin{aligned} &\varphi(e) < e \Rightarrow \varphi(e+1) < e, \\ &e \leq \varphi(e) \Rightarrow \varphi(\varphi(e)+1) = \varphi(e). \end{aligned}$$

We have proved that, under these requirements, the computable rounding to nearest is equivalent to a rounding to nearest that is the one described in Section III-B with a P that is "the canonical floating-point number (with exponent $\mathrm{cexp}(x)$) has an even mantissa".

These properties on φ are satisfied by the FIX format. For FLX and FLT formats of precision p, these properties degenerate to simply $1 < p$. For $p = 1$, we have neither the existence, nor the uniqueness of rounding to nearest, ties to even. The case of the FTZ format is special, since breaking tie to even is no longer deterministic: both 0 and its successor $\beta^{p-1} \cdot \beta^{e_{\min}}$ have even mantissas when $1 < p$ and β is even.

Note that rounding operators do not imply that we cannot use a generic rounding to nearest. We defined a Zrnd to nearest depending on a choice function that gives either the rounding to nearest when unique or one of the two possible values depending on the result of the choice function when in the middle. By proving some theorems whatever the choice function, we retrieve the behavior of the generic rounding to nearest without ties defined in PFF and in Section III-B.

Another interesting property of format FTZ is the way the rounding operators are defined on the subnormal range. There are two possible implementations, either small real values are correctly rounded toward the smallest normal numbers (sometimes called *abrupt underflow*), or they are simply flushed to zero, irrespective of the rounding direction. In the first case, the previous definitions of the rounding operators are still valid. In the second case, a variant must be used:

$$\mathrm{Zrnd}_{\mathrm{FTZ}}(x) = \begin{cases} \mathrm{Zrnd}(x) & \text{if } |x| \geq 1, \\ 0 & \text{otherwise.} \end{cases}$$

IV. Other parts

A. Computable functions

There are some situations where an axiomatic approach to floating-point arithmetic is no longer sufficient and one would like to actually compute with a proof assistant. That may be useful for generating test values, or for brute-forcing the few inputs that elude the correctness proof of an algorithm [21], or for automating proofs of theorems, be they about floating-point arithmetic [15] or real arithmetic [18].

1) Computable rounding operators: While the rounding formulas of Section III-E are closed, they cannot be used to effectively compute with real numbers. A subset on which $\lfloor\cdot\rfloor$-like functions are decidable has to be chosen. This could be the subset of rational numbers; it would be suitable for dealing with addition, multiplication, and division of floating-point numbers, but difficulties would arise for square root already. We have chosen a different approach: we have decided to over-approximate real numbers by a floating-point number (not necessarily in $\mathbb{F}_\varphi$) and a location relative to this number.

Let us consider a real number x and a floating-point number $d = m \cdot \beta^e$ such that $m \cdot \beta^e \leq x < (m+1) \cdot \beta^e = u$. This is the property *inbetween_float* which relates x, m, e, and a location ℓ. If x is equal to d, then ℓ tells so: $\ell = \mathrm{loc\_Exact}$. Otherwise, ℓ tells whether x is smaller, equal, or bigger than $(d+u)/2$. So there are four different locations of x with respect to d. They are similar to the usage of the *round* and *sticky* bits that can be found in hardware implementations.

This approach restricts the rounding operators that can be computed by our formalism. They have to round all the real numbers strictly between d and $(d+u)/2$ toward the same floating-point number of $\mathbb{F}_\varphi$, and similarly for $(d+u)/2$ and u, assuming that d and u are consecutive floating-point numbers in $\mathbb{F}_\varphi$. Fortunately, this property is sufficient to express all the standard rounding modes, as shown by Theorems *inbetween_float_DN*, *_UP*, *_NE*, and so on.

Given a lower bound $d = m \cdot \beta^e$ and a location ℓ of the real number x, one can compute a new lower bound $d' = \lfloor m/\beta \rfloor \cdot \beta^{e+1}$ and a location ℓ' of x with respect to d'. This location is computed from β, the remainder of the Euclidean division of m by β, and ℓ only. The process can be iterated until one obtains a lower bound in $\mathbb{F}_\varphi$. The resulting location then tells how to round the real number.

More precisely, $\mathrm{cexp}(d) = \varphi(\mathrm{slice}(m)+e)$ can be computed by counting the digits of m and applying φ. If $x \geq 0$ and $\mathrm{cexp}(d) \leq e$, then $\mathrm{cexp}(x) = \mathrm{cexp}(d)$. A suitable triple $(m', \mathrm{cexp}(x), \ell')$ (Theorem *truncate_correct*) can then be computed. At this point, Theorem *round_any_correct* explains how to perform the final rounding. Some simpler theorem instances that combine both theorems with a given rounding mode are provided for ease-of-use:

Theorem 3 (round_trunc_UP_correct) *Assuming φ is valid, for any positive real number x over-approximated by a triple (m, e, ℓ), if either $e \leq \varphi(\mathrm{digits}(m) + e)$ or $\ell = \mathrm{loc\_Exact}$, then*

$$(m', e', \ell') = \mathrm{truncate}_\varphi(m, e, \ell) \Rightarrow$$
$$\mathrm{round}_\varphi^{\mathrm{UP}}(x) = \begin{cases} m' \cdot \beta^{e'} & \text{if } \ell' = \mathrm{loc\_Exact}, \\ (m'+1) \cdot \beta^{e'} & \text{otherwise.} \end{cases}$$

Note that the computed floating-point number does not necessarily have a canonical exponent.

2) Computable arithmetic operators: Being able to compute a correctly-rounded floating-point number from a triple is only part of the work. One should also be able to compute triples for basic arithmetic operations. For addition and multiplication of floating-point numbers, the result of the real operation is representable as a floating-point number, so building a suitable triple is straightforward.

The situation is slightly more complicated for division and square root, since their results are often not representable. As a consequence, the operators $\mathrm{Fdiv\_core}$ and $\mathrm{Fsqrt\_core}$ also take a precision as input. The triple they compute is then guaranteed to locate the real result and to have at least the requested number of digits. Both operators are similar: they first shift the mantissas of the floating-point inputs and then perform an integer division or square root. The integer remainder is used to decide the location of the real result with respect to the computed floating-point number. The implementation of these operators was taken from the floating-point kernel of Coq.Interval [13].

Because the operators compute only triples and not rounded results, they are independent of formats and rounding modes (as can be seen by the position of Fcalc_div and Fcalc_sqrt on Figure 2). It it up to the user to combine them with rounding operators to get a full rounded operator. This combination amounts to choosing the number of digits of the triple and proving that it is an upper bound on the size of the rounded result. For instance, always asking for p digits when the format is FLT with precision p is sufficient, whether the result is normal or subnormal. The core of the correctness proof of such an operator is about 15 lines, which makes it tractable from a Coq point of view. The library provides Theorem *Fsqrt_FLT_ne_correct* as an example of instantiating the square root operator for a specific format and a specific rounding mode.

Note that these arithmetic operators were designed with correctness and computability in mind, not performance. For instance, adding to x a negligible number y will cause the adder to shift x until it has the same exponent as y, then the result will be truncated until x (or some value close to it) is obtained again. The efficient way of performing the addition would have been to truncate y until it has the same exponent as x instead. For other operators, performance issues will arise when the inputs have a precision much bigger than the output. This is nothing new; efficient algorithms have already been designed for the multi-precision library MPFR [19] but their correctness proofs are on a different level than the proofs of Flocq's operators.

B. High-level lemmas

To assess the usefulness and practicality of our library, we have chosen to prove well-known generic-radix theorems that are either helpful, or expectedly difficult, meaning that they are proved in PFF but they belong to where the formalizations differ the most.

1) Exact subtraction: This well-known fact [22] states that if x and y are floating-point numbers near enough one to another ($y/2 \leq x \leq 2y$), then the subtraction $x-y$ is computed without rounding.

Theorem 4 (sterbenz) *Assuming that φ is valid and satisfies*

$$\forall e_x, e_y \in \mathbb{Z},\ e_x \leq e_y \Rightarrow \varphi(e_x) \leq \varphi(e_y),$$

Then, for all x and y in $\mathbb{F}_\varphi$ such that $\frac{y}{2} \leq x \leq 2y$, we have that $x - y \in \mathbb{F}_\varphi$.

As our format is very generic, there is a requirement on the φ function, that is that φ is monotone. This is reasonable as both FIX, FLX, and FLT fulfill it.

This is also necessary: let us consider in radix 2 a function φ such that $\varphi(0) = -2$ and $\varphi(-1) = -1$. Let $x = 0.75$ and $y = 0.5$. The slice of both x and y is 0 as $2^{-1} \leq x, y < 2^0$. So x and y are in the generic format with exponent $\varphi(0) = -2$. Moreover, $y/2 \leq x \leq 2y$. Then $x - y = 0.25$ has a slice of -1 and therefore needs to be represented with an exponent $\varphi(-1) = -1$ but 0.25 cannot be represented as $m \times 2^{-1}$.

2) Relative error: The idea is that there exists a small ε such that $\mathrm{round}(x) = x \cdot (1 + \varepsilon)$. Then rounded values are replaced with such formulas and the various ε are added, multiplied, and so on. See [23] for examples of use.

We proved this property also with a requirement on the φ function:

Theorem 5 (generic_relative_error_ex) *Let us assume that φ is valid and that there exists p and $e_{\min}$ such that*

$$\forall k \in \mathbb{Z}, \quad e_{\min} < k \Rightarrow p \leq k - \varphi(k).$$

Then, for any rounding operator round_φ and for any real x such that $\beta^{e_{\min}} \leq |x|$, there exists ε such that

$$|\varepsilon| < \beta^{1-p} \quad \textit{and} \quad \mathrm{round}_\varphi(x) = x \cdot (1 + \varepsilon).$$

We also proved the special case of rounding to nearest where $|\varepsilon| \leq \frac{\beta^{1-p}}{2}$.

Note that the requirement on the φ function intuitively corresponds to the non-underflow cases of the FLT format. This cannot be applied to fixed-point formats for example.

From that theorem, we deduced its application to FLX and FLT: in FLX, there is no requirement on x for the theorem to hold with p being the precision. In FLT, we pose $e_{\min}$ and p as expected and we have the existence of ε provided $\beta^{e_{\min}+p-1} \leq |x|$.

3) Error of the addition: It is a well-known fact that, using rounding to nearest, the error of a floating-point addition is a floating-point number. It can even be computed using floating-point operations [24]. This theorem and the following ones about multiplication, division, and square root are the bases of error-free transformations [25].

As before, we need a hypothesis on the φ function (the same as for Sterbenz exact subtraction):

Theorem 6 (plus_error) *Assuming that φ is valid and monotone, for any rounding to nearest round_φ^N,*

$$\forall x, y \in \mathbb{F}_\varphi, \quad \mathrm{round}_\varphi^N(x + y) - (x + y) \in \mathbb{F}_\varphi.$$

The genericity of the theorem with respect to formats can be seen on Figure 2: the Fprop_plus_error file does not depend on the specialized formats defined in Fcore_FIX, _FLX, and so on.

4) Error of the multiplication: It is also well-known that the error of a floating-point multiplication is a floating-point number, whatever the rounding, but provided no underflow occur [24].

We then first proved this using the FLX format:

Theorem 7 (mult_error_FLX) *For any rounding operator $\mathrm{round}_{\varphi^{\mathrm{X}}}$ on a FLX format $\mathbb{F}_{\varphi^{\mathrm{X}}}$,*

$$\forall x, y \in \mathbb{F}_{\varphi^{\mathrm{X}}}, \quad \mathrm{round}_{\varphi^{\mathrm{X}}}(x \times y) - x \times y \in \mathbb{F}_{\varphi^{\mathrm{X}}}.$$

To get a result in the FLT format similar to [16], we first proved the preceding theorem while giving precisely its exponent. With this fact and results about translation between FLX and FLT, we easily proved the corresponding theorem in FLT with additional hypothesis for the gradual underflow:

Theorem 8 (mult_error_FLT) *For any rounding operator $\mathrm{round}_{\varphi^{\mathrm{T}}}$ on a FLT format $\mathbb{F}_{\varphi^{\mathrm{T}}}$ with a minimal exponent $e_{\min}$,*

$$\forall x, y \in \mathbb{F}_{\varphi^{\mathrm{T}}}, \quad \beta^{e_{\min}+2p-1} \leq |x \times y| \Rightarrow$$
$$\mathrm{round}_{\varphi^{\mathrm{T}}}(x \times y) - x \times y \in \mathbb{F}_{\varphi^{\mathrm{T}}}.$$

5) When the rounding is zero: Another useful fact is when $x + y$ rounds to zero, then it usually means that $x + y$ is mathematically equal to zero. This is correct in FLT and in FLX. We proved it for a generic format:

Theorem 9 (round_plus_eq_zero) *Assuming that φ is valid and satisfies the predicate* not_FTZ_prop*:*

$$\forall e \in \mathbb{Z},\ \varphi(\varphi(e + 1)) \leq \varphi(e),$$

for any rounding operator round_φ,

$$\forall x, y \in \mathbb{F}_\varphi, \quad \mathrm{round}_\varphi(x + y) = 0 \Rightarrow x + y = 0.$$

The assumption on φ seems strange. It corresponds to the fact that the format must not be FTZ. Indeed, this theorem does not hold in FTZ and this requirement (that is fulfilled in FLX, FLT, and FIX) is sufficient to avoid the bad cases.

6) Remainder of the division and square root: It is well-known that the remainder of the division, that is to say the real value $x - \circ(x/y) \times y$, and the remainder of the square root, that is to say $x - \circ(\sqrt{x}) \times \circ(\sqrt{x})$, usually are floating-point numbers. It is not the case when an underflow occurs [17]. This is the reason why we proved it in FLX format, meaning we assume there is no underflow.

Theorem 10 (div_error_FLX) *For any rounding operator* $\mathrm{round}_{\varphi^{\mathrm{x}}}$ *on a FLX format* $\mathbb{F}_{\varphi^{\mathrm{x}}}$,

$$\forall x, y \in \mathbb{F}_{\varphi^{\mathrm{x}}}, \quad x - \mathrm{round}_{\varphi^{\mathrm{x}}}(x/y) \times y \in \mathbb{F}_{\varphi^{\mathrm{x}}}.$$

The square root requires rounding to nearest.

Theorem 11 (sqrt_error_FLX) *For any rounding to nearest* $\mathrm{round}^N_{\varphi^{\mathrm{x}}}$ *on a FLX format* $\mathbb{F}_{\varphi^{\mathrm{x}}}$,

$$\forall x \in \mathbb{F}_{\varphi^{\mathrm{x}}}, \quad x - \left(\mathrm{round}^N_{\varphi^{\mathrm{x}}}(\sqrt{x})\right)^2 \in \mathbb{F}_{\varphi^{\mathrm{x}}}.$$

In contrast to intuition, the two preceding theorems do not have to assume anything on x and y (hence are easier to apply), due to some peculiarities of real arithmetic in Coq. For division, y does not have to be nonzero as $x - \mathrm{round}_{\varphi^{\mathrm{x}}}(x/y) \times y$ is provably equal to x when $y = 0$. The square root of a negative number is defined as zero in Coq, so $x - (\mathrm{round}^N_{\varphi^{\mathrm{x}}}(\sqrt{x}))^2$ is provably equal to x when $x < 0$.

7) Predecessor and successor: Floating-point numbers are a discrete set. This is still true for our generic format with no additional requirement on the φ function. Among the useful properties of the fact that the set of floating-point numbers is discrete is the existence of a predecessor and a successor. For $x > 0$ in the format, its successor is $x + \mathrm{ulp}(x)$ and its predecessor is

$$\mathrm{pred}(x) = \begin{cases} x - \beta^{\varphi(\mathrm{slice}(x)-1)} & \text{if } x = \beta^{\mathrm{slice}(x)-1}, \\ x - \mathrm{ulp}(x) & \text{otherwise.} \end{cases}$$

These values are proved to be in the format and are such that $\triangle(x + \varepsilon) = x + \mathrm{ulp}(x)$ and $\bigtriangledown(x - \varepsilon) = \mathrm{pred}(x)$ (if $\mathrm{pred}(x) > 0$) for a small enough ε.

Theorem 12 (pred_ulp) *Assuming* φ *is valid, for any* $x \in \mathbb{F}_\varphi$ *such that* $0 < \mathrm{pred}(x)$*, we have*

$$\mathrm{pred}(x) + \mathrm{ulp}(\mathrm{pred}(x)) = x.$$

Theorem 13 (le_pred_lt, succ_lt_le) *Assuming* φ *valid, for any* x *and* y *in* $\mathbb{F}_\varphi$ *such that* $0 < x < y$*, we have*

$$x \leq \mathrm{pred}(y) \quad \textit{and} \quad x + \mathrm{ulp}(x) \leq y.$$

V. Conclusion

In the formal proof community, people are aware that designing libraries is a challenge, as they should be both usable and correct. For example, [2] had subtle discrepancies with the IEEE-754 standard, as pointed out by Pr Kahan at the July 2002 meeting of the IEEE 754R group.[1] The correctness of Flocq comes from its mathematical definition of rounding predicates: it is hard to get them wrong when rounding down is defined as the biggest floating-point number smaller than a given real number. As for its usability, it was exercised by the many high-level theorems we were able to prove in a convenient way.

This paper presents only a small overview of the library, as it contains more than 450 theorems. The library is available under an open-source license at `http://flocq.gforge.inria.fr`. Its definitions and theorems can also be browsed online. The library has 23 files in three directories: the core of the library, its computable definitions and properties, and the high-level properties. Figure 2 shows how the files from the three directories depend on each other.

As a consequence of the generalization of the theorems to other formats (including possibly degenerate ones), this work has also been the occasion to uncover some pathological cases. In particular, our decision of natively handling flush-to-zero formats has led us to consider unusual requirements for common theorems. This extreme generalization puts Flocq apart from the existing libraries (be they for Coq or other systems) which tend to focus on one specific arithmetic.

Since Flocq was motivated by our experience with previous libraries (in particular PFF and Gappa), we will conclude by comparing them to Flocq.

A. PFF

The main characteristics of PFF are its practicality for the FLT format, its size, and its lack of automation. As the roundings are axiomatic, it is very difficult to prove that a given floating-point number is the rounding of a given real. Flocq solves this problem by easing the usage of automated tools (see Section V-B).

Compared to the PFF library, Flocq is more generic as it easily handles both FLT, FLX, FIX, and even more exotic formats such as formats without subnormal numbers. As PFF was dedicated to FLT, it has a different basic formalization from Flocq, therefore theorems cannot be automatically translated from one to another. This is unpleasant as it means re-doing all the proofs.

The worst expected case for translation is when the proofs of PFF heavily rely on the representation of floating-point numbers, such as the proofs of the facts that the error of the addition, multiplication, the remainder of the division and of the square root are representable floating-point-numbers (under various assumptions, see Section IV-B). This is the reason we chose them as experiments for the translations of theorems. Those pure FLT proofs have been translated and

[1] http://grouper.ieee.org/groups/754/meeting-minutes/02-07-18.html

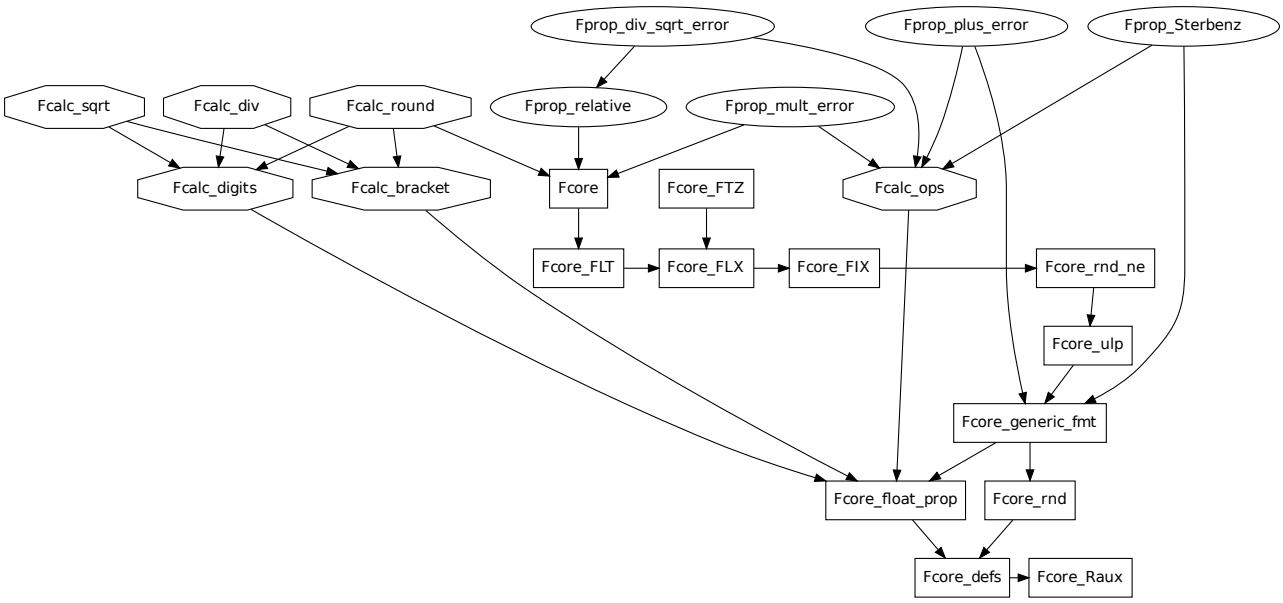

Fig. 2. Dependency graph of the library. Theorems from one file depend on the theorems from the pointed files. The shapes of the node classify the origin of the files: rectangles for the core library (Section III), octagons for the computable operators (Section IV-A), and ellipses for the high-level properties (Section IV-B).

generalized to more formats. The proofs had to be rethought as the basic formalizations were different. Even if the proof ideas are mainly the same, the details and the hypotheses are different.

Nevertheless, our experiments have proved it is possible. We are currently proving properties about the quasi-correctness of $\circ(\circ(a \times x) + y)$, when $|y|$ is much greater than $|a \times x|$ [17].

B. Gappa

Two of the motivations for the Flocq library were: avoiding proof duplication between formalisms for Coq and allowing automation within a high-level formalism. Gappa's support library has therefore been rewritten to depend on Flocq.

The first benefit of this change is a reduction of the size of the library by a third, while supporting more theorems. For instance, the usual semantic of rounding modes is now available for Gappa's operators, *e.g.* $\bigtriangledown(x)$ is the biggest representable number that is less or equal to x. Such properties were already true before, but they had to be proved outside Gappa's formalism [15]. More generally, Gappa's rounding operators are now simply Flocq's ones and hence benefit from all their theorems.

The second benefit of Gappa no longer defining its own operators is that the `gappa` tactic of Coq can be used to automate the proof of goals written with Flocq's formalism. Consider the example of Figure 3. Line 1 defines the standard double-precision floating-point format: radix-2 FLT format with a precision of 53 bits and a minimal exponent equal to -1074. Line 3 is just a notation for rounding to zero in order to shorten the proof. The goal to prove ranges from line 7 to 11; it states that if the real numbers $a \in [52/16; 53/16]$ and $b \in [22/16; 30/16]$ are representable in the format, their difference is representable too. Note that this is not an instance of exact subtraction as stated in Section IV-B1, since a/b is possibly bigger than 2. The proof of this property is lines 13–18. First, line 14 states that proving that $a - b$ is representable is the same as proving that it is equal to its rounded value (see Section III-D). For the same reason, lines 16 and 17 replace a and b by their rounding to zero. Finally, the Gappa tool is called to automatically complete the proof. The completed proof is entirely checked by Coq.

```
1  Definition format :=
2    generic_format radix2 (FLT_exp (-1074) 53).
3  Notation rnd :=
4    (round radix2 (FLT_exp (-1074) 53) rndZR).
5
6  Goal
7    forall a b : R,
8    format a -> format b ->
9    52 / 16 <= a <= 53 / 16 ->
10   22 / 16 <= b <= 30 / 16 ->
11   format (a - b).
12 Proof.
13   intros a b Ha Hb Ia Ib.
14   change (a - b = rnd (a - b)).
15   revert Ia Ib.
16   replace a with (rnd a).
17   replace b with (rnd b).
18   gappa.
19 Qed.
```

Fig. 3. Example of a proof script using automation.

C. Perspectives

Certifying the proof obligations associated to the correctness of numerical programs requires mixing several floating-point formalisms [15]. The tediousness of this approach was an incentive for developing the Flocq library. These proof obligations are generated by an automated tool: the Frama-C/Jessie/Why toolchain. It takes an annotated C program as input, performs weakest precondition computations on it, and generates theorem statements that, once proved, guarantee that the original program fulfills its specification. Why was using the formalism of PFF for generating the floating-point obligations [26]. It now uses Flocq.

Another objective is to fill the gap between a high-level formalization of floating-point arithmetic and the IEEE-754 description as bit vectors. The goal is to have the tools for a full certification of an algorithm from its C source code to its final assembly code. One of the building blocks is CompCert, a certified compiler written in Coq [27] and floating-point support is being added to this compiler. Therefore, related code constructs and optimizations have to be formally proved too. For instance, the conversion from integers to floating-point values may involve a mix of bit-level operations (writing a magic constant in the most significant word of a floating-point register) and floating-point operations. Proving the correctness of such a code sequence requires an extended formalization. We have started such a development to handle exceptional values (signed zeros, infinities, NaN) and behaviors.

References

[1] IEEE Computer Society, *IEEE Standard for Floating-Point Arithmetic*. IEEE Standard 754-2008, 2008.

[2] G. Barrett, "Formal methods applied to a floating-point system," *IEEE Transactions on Software Engineering*, vol. 15, no. 5, pp. 611–621, 1989.

[3] V. A. Carreño and P. S. Miner, "Specification of the IEEE-854 floating-point standard in HOL and PVS," in *Proceedings of the International Workshop on Higher Order Logic Theorem Proving and its Applications*, Aspen Grove, UT, USA, 1995.

[4] P. Loiseleur, "Formalisation en Coq de la norme IEEE-754 sur l'arithmétique à virgule flottante," 1997.

[5] J. S. Moore, T. W. Lynch, and M. Kaufmann, "A mechanically checked proof of the correctness of the kernel of the AMD5K86 floating-point division algorithm," *IEEE Transactions on Computers*, vol. 47, no. 9, pp. 913–926, 1998.

[6] D. M. Russinoff, "A case study in formal verification of register-transfer logic with ACL2: The floating point adder of the AMD Athlon processor," in *Proceedings of the 3rd International Conference on Formal Methods in Computer-Aided Design*, ser. Lecture Notes in Computer Science, W. A. Hunt and S. D. Johnson, Eds., vol. 1954, Austin, TX, USA, 2000, pp. 3–36.

[7] R. Kaivola and K. Kohatsu, "Proof engineering in the large: formal verification of Pentium 4 floating-point divider," *International Journal on Software Tools for Technology Transfer*, vol. 4, no. 3, pp. 323–334, 2003.

[8] C. Jacobi and C. Berg, "Formal verification of the VAMP floating point unit," *Formal Methods in System Design*, vol. 26, no. 3, pp. 227–266, 2005.

[9] J. Harrison, "A machine-checked theory of floating point arithmetic," in *Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics*, Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, Eds., Nice, France, 1999, pp. 113–130.

[10] ——, "Formal verification of floating-point trigonometric functions," in *Proceedings of the 3rd International Conference on Formal Methods in Computer-Aided Design*, ser. Lecture Notes in Computer Science, W. A. Hunt and S. D. Johnson, Eds., no. 1954, Austin, TX, USA, 2000, pp. 217–233.

[11] M. Daumas, L. Rideau, and L. Théry, "A generic library of floating-point numbers and its application to exact computing," in *Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics*, Edinburgh, Scotland, 2001, pp. 169–184.

[12] S. Boldo, "Pitfalls of a full floating-point proof: example on the formal proof of the Veltkamp/Dekker algorithms." in *Proceedings of the 3rd International Joint Conference on Automated Reasoning*, ser. Lecture Notes in Computer Science, U. Furbach and N. Shankar, Eds., vol. 4130, Seattle, WA, 2006, pp. 52–66.

[13] G. Melquiond, "Floating-point arithmetic in the Coq system," in *Proceedings of the 8th Conference on Real Numbers and Computers*, Santiago de Compostela, Spain, 2008, pp. 93–102.

[14] M. Daumas and G. Melquiond, "Certification of bounds on expressions involving rounded operators," *Transactions on Mathematical Software*, vol. 37, no. 1, pp. 1–20, 2010.

[15] S. Boldo, J.-C. Filliâtre, and G. Melquiond, "Combining Coq and Gappa for certifying floating-point programs," in *Proceedings of the 16th Calculemus Symposium*, ser. Lectures Notes in Artificial Intelligence, J. Carette, L. Dixon, C. S. Coen, and S. M. Watt, Eds., vol. 5625, Grand Bend, ON, Canada, 2009, pp. 59–74.

[16] S. Boldo and M. Daumas, "Representable correcting terms for possibly underflowing floating point operations," in *Proceedings of the 16th Symposium on Computer Arithmetic*, J.-C. Bajard and M. Schulte, Eds., Santiago de Compostela, Spain, 2003, pp. 79–86.

[17] ——, "A simple test qualifying the accuracy of Horner's rule for polynomials," *Numerical Algorithms*, vol. 37, no. 1–4, pp. 45–60, 2004.

[18] G. Melquiond, "Proving bounds on real-valued functions with computations," in *Proceedings of the 4th International Joint Conference on Automated Reasoning*, ser. Lectures Notes in Artificial Intelligence, A. Armando, P. Baumgartner, and G. Dowek, Eds., vol. 5195, Sydney, Australia, 2008, pp. 2–17.

[19] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "MPFR: A multiple-precision binary floating-point library with correct rounding," *ACM Transactions on Mathematical Software*, vol. 33, no. 2, 2007.

[20] D. M. Priest, "Algorithms for arbitrary precision floating point arithmetic," in *Proceedings of the 10th Symposium on Computer Arithmetic*. IEEE Computer Society Press, 1991, pp. 132–145.

[21] J. Harrison, "Formal verification of square root algorithms," *Formal Methods in Systems Design*, vol. 22, pp. 143–153, 2003.

[22] P. H. Sterbenz, *Floating-Point Computation*. Prentice-Hall, Englewood Cliffs, NJ, 1974.

[23] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*. SIAM, Philadelphia, 1996.

[24] D. Knuth, *The Art of Computer Programming*, 3rd ed. Addison-Wesley, Reading, MA, 1998, vol. 2.

[25] T. Ogita, S. M. Rump, and S. Oishi, "Accurate sum and dot product," *SIAM J. Sci. Comput*, vol. 26, p. 2005, 2005.

[26] S. Boldo and J.-C. Filliâtre, "Formal verification of floating-point programs," in *Proceedings of the 18th IEEE Symposium on Computer Arithmetic*, P. Kornerup and J.-M. Muller, Eds., Montpellier, France, 2007, pp. 187–194.

[27] X. Leroy, "A formally verified compiler back-end," *Journal of Automated Reasoning*, vol. 43, no. 4, pp. 363–446, 2009.