Behavioral Properties of Floating-Point Programs*

Ali Ayad[1,3] and Claude Marché[2,3]

[1] CEA LIST, Software Safety Laboratory, Point Courrier 94,
Gif-sur-Yvette, F-91191 France
[2] INRIA Saclay - Île-de-France, F-91893 Orsay cedex
[3] LRI, Univ. Paris-Sud, CNRS, F-91405 Orsay cedex

Abstract. We propose an expressive language to specify formally behavioral properties of programs involving floating-point computations. We present a deductive verification technique, which allows to prove formally that a given program meets its specifications, using either SMT-class automatic theorem provers or general interactive proof assistants. Experiments using the Frama-C platform for static analysis of C code are presented.

1 Introduction

Floating-point computations appear in many critical problems such as numerical analysis, physics, aeronautics (software embedded in satellites, robots), energy (nuclear centers), automotive, etc. As we rely more and more on software, the consequences of a bug are more and more dramatic, causing great financial and even human losses.

There are numerous approaches to checking that a program runs as expected. On the one hand, *dynamic* analyses run the program on selected inputs and checks if the outputs are the one expected; their completeness relies on some coverage criteria. On the other hand, *static* analyses check the program code without running it. Some static analyses are performing abstractions on the code; this is the case for model checking and of abstract interpretation techniques. *Deductive* verification techniques, originating from the landmark approach of Floyd-Hoare logic, perform static analysis of code without abstraction. A common approach is to generate automatically logic formulas called *verification conditions* (VCs for short), with techniques for instance based on Dijkstra's weakest precondition calculus [15]. The generated formulas must be checked valid hopefully by automatic theorem provers. Complex behavioral properties of programs can be verified by deductive verification techniques, since these techniques usually come with expressive specification languages to specify the requirements. Nowadays, several implementations of deductive verification approaches exist for standard programming languages, e.g., ESC-Java2 [11] and KeY [6] for Java, Spec# [3] for C#, VCC [28] and Frama-C [18] for C. In each of them, *contracts* (made of preconditions, postconditions, and several other kinds of annotations) are inserted into the program

* This work was supported by the French national projects: *CerPan* (*Certification of numerical programs*, ANR-05-BLAN-0281-04), *Hisseo* (*Static and dynamic analysis of floating-point programs*, Digiteo 09/2008-08/2011), and *U3CAT* (*Unification of Critical C Code Analysis Techniques*, ANR-09-ARPEGE)

source text with specific syntax, usually in a special form of comments that are ignored by compilers. The resulting annotations languages are called *Behavioral Interface Specification Languages*, e.g., JML [9] for Java, ACSL [5] for C.

To analyse FP (floating-point) computations, abstract interpretation-based techniques have shown quite successful on critical software, as examplified by tools *Fluctuat* [20] and *Astree* [7]. However, there are very few attempts to provide ways to specify and to prove behavioral properties of floating-programs in deductive verification systems like those mentioned above. A first proposal has been made in 2007 by Boldo and Filliâtre [8] for C code, using the Coq proof assistant [29] for discharging VCs. The approach presented in this paper is a follow-up of the Boldo-Filliâtre approach, which we extend in two main directions: first a full support of IEEE-754 standard for FP computations, including special floating-point values (for infinity, NaN (Not-a-Number)); and second the use of *automatic* theorem provers. Our contributions are the following:

- Additional constructs to specification languages for specifying behavioral properties of FP computations. This is explained in Section 3.
- Modeling of FP computations by a first-order axiomatization, suitable for a large set of different theorem provers, and interpretation of annotated programs in this modeling (Section 4). There are two possible interpretations of FP operations in programs: a *defensive* version which forbids overflows and consequently apparition of special values (Section 4.3); and a *full* version which allows special values to occur (Section 4.4).
- Combination of several provers, automatic and interactive ones, to discharge VCs (Section 5).

Our approach is implemented in the Frama-C [18] platform for static analysis of C code, and experiments performed with this platform are presented along this paper.

2 Preliminaries: the IEEE-754 Standard

The IEEE-754 Standard [1] defines an expected behavior of FP (floating-point) computations. It describes binary and decimal formats to represent FP numbers, and specifies the elementary operations and the comparison operators on FP numbers. It explains when FP exceptions occur, and introduces special values to represent signed infinities and NaNs. We summarize here the essential parts of the standard we need.

In this paper we focus on the 32-bits (type `float`) and 64-bits (type `double`) binary formats; adaptation to other binary formats decimal format is straightforward. Generally speaking, in any of these formats, an interpretation of the bit sequence under the form of a sign, a mantissa and an exponent is given, so that the set of FP numbers denote a finite subset of real numbers, called the set of *representable* numbers in that format.

For each of the basic operations (add, sub, mul, div, and also sqrt, fused-multiply-add, etc.) the standard requires that it acts as if it first computes a true real number, and then *rounds* it to a number representable in the chosen format, according to some *rounding mode*. The standard defines five rounding modes: if a real number x lies between two consecutive representable FP numbers x_1 and x_2, then the rounding of x is as follows.

- *roundTowardPositive*: the rounding of x is x_2.
- *roundTowardNegative*: the rounding of x is x_1.
- *roundTowardZero*: the rounding of x is x_1 if $x > 0$ and x_2 if $x < 0$.
- *roundTiesToAway*: the rounding of x is the closest to x among x_1 and x_2. If x is exactly the middle of the interval $[x_1, x_2]$ then the rounding is x_2 if $x > 0$ and x_1 if $x < 0$.
- *roundTiesToEven*: the rounding of x is the closest to x among x_1 and x_2. If x is exactly the middle of the interval $[x_1, x_2]$ then one chooses whose significand (of canonical representatives) is even.

The standard defines three special values: $-\infty, +\infty$ and NaN. It also distinguishes between positive zero (+0) and negative zero (-0) [19]. These numbers should be treated both in the input and the output of the arithmetic operations as usual. For example, $(+\infty) + (+\infty) = (+\infty)$, $1/(+\infty) = +0$, $1/(-0) = -\infty$, $(+0)/(+0) = \mathsf{NaN}$, $(-0) \times (+\infty) = \mathsf{NaN}$ and $\sqrt{-1} = \mathsf{NaN}$ but $\sqrt{-0} = -0$. Operations propagate NaN operands to their results.

3 Specification of Floating-Point Programs

The purpose of this section is to propose extensions to specification languages in order to specify properties of FP programs. As a basis for the specification language, we consider classical first-order logic with built-in equality and arithmetic on both integer and real numbers. We assume also built-in symbols for standard functions such as absolute value, exponential, trigonometric functions and such. Those are typically denoted with backslashes: `\abs`, `\exp`, etc.

3.1 The core annotation language

The core of the specification language is made of a classical Behavioral Interface Specification Language (ACSL [5] in our examples) which allows to function contracts (precondition, postconditions, frame clauses, etc.), code annotations (code assertions, loop invariants, etc.) and data invariants.

To deal with FP properties, we first make important design choices:

1. There is no FP arithmetic in the annotations: Operators $+$, $-$, $*$, $/$ denote operations on mathematical real numbers.
2. Consequently, there are neither rounding nor overflow that can occur in logic annotations.
3. In annotations, any FP program variable, or more generally any C left-value of type `float` or `double`, denotes the real number it represents.

The following example illustrates the impact of these choices. Notice that in our specification language, we use the C99 notation for hexadecimal FP literals, of the form `0x`hh`.`hh`p`dd where h are hexadecimal digits and dd is in decimal, which denotes number $hh.hh \times 2^{dd}$, e.g. `0x1.Fp-4` is $(1 + 15/16) \times 2^{-4}$.

```
/*@ requires \abs(x) <= 1.0;
  @ ensures \abs(\result - \exp(x)) <= 0x1p-4;
  @*/
double my_exp(double x) {
  /*@ assert \abs(0.9890365552 + 1.130258690*x + 0.5540440796*x*x
    @          - \exp(x)) <= 0x0.FFFFp-4;
    @*/
  return 0.9890365552 + 1.130258690 * x + 0.5540440796 * x * x;
}
```

Fig. 1. Remez approximation of the exponential function

Example 1. The C code of Fig. 1 is an implementation of the exponential function for double precision FP numbers in interval $[-1; 1]$, using a Remez polynomial approximation of degree 2.

The contract declared above the function contains a precondition (introduced by keyword `requires`) which states that this function is to be called only for values of x with $|x| \leq 1$. The postcondition (introduced by keyword `ensures`) states that the returned value (`\result`) is close to the real exponential, the difference being not higher than 2^{-4}. The function body contains an `assert` clause, which specifies a property that holds at the corresponding program point. In that particular code, it states that the expression $0.9890365552 + 1.130258690x + 0.5540440796x^2 - \exp(x)$ *evaluated as a real number*, hence without any rounding, is not greater than $(1-2^{-16}) \times 2^{-4}$.

The intermediate assertion thus naturally specifies the *method error*, induced by the mathematical difference between the exponential function and the approximating polynomial; whereas the postcondition takes into account both the method error and the *rounding errors* added by FP computations.

3.2 Exact computations within programs

Rounding errors naturally propagates across function calls, so when reasoning about FP programs, one wants to specify expected rounding errors on function parameters in precondition. A nice way to proceed is to refer to *exact* computations [8]: the computations that would be performed in an ideal mode where variables denote true real numbers.

To express such exact computations, we introduce two special constructs in annotations:

- `\exact`(x) denotes the value of the C variable x (or more generally any C left-value) as if it was computed in real numbers.
- `\round_error`(x) is a shortcut for $|x - $ `\exact`$(x)|$

Example 2. The code of Fig. 2 is a simple implementation of the cosine function for single precision FP numbers close to zero, based on Taylor approximation. The first precondition assumes that the exact value of x is less than $1/32$, and the second precondition expresses that accumulated rounding errors on x are less than 2^{-20}. The first postcondition states that the exact value of the result is close to the cosine of the exact

```
/*@ requires \abs(\exact(x)) <= 0x1p-5;
  @ requires \round_error(x) <= 0x1p-20;
  @ ensures \abs(\exact(\result) - \cos(\exact(x))) <= 0x1p-24;
  @ ensures \round_error(\result) <= \round_error(x) + 0x3p-24;
  @*/
float cosine(float x) {
  return 1.0f - x * x * 0.5f;
}
```

Fig. 2. Exact computations and rounding errors in annotations

value of x (difference less than 2^{-24}), whereas the second postcondition bounds the new accumulated rounding error.

3.3 Special values and rounding modes

So far we did not specify anything about the rounding mode in which programs are executed. In Java, or by default in C, the default rounding mode is NearestEven. In the C99 standard, there is a possibility for dynamically changing it using `fesetround()`. For efficiency issues, is not recommended to change it often, so usually a program will run in a fixed rounding mode set once for all.

To specify what is the expected rounding mode we chose to provide a special global declaration in the specification language:

`pragma roundingMode`(*value*)

where *value* is either one of the 5 IEEE mode, or 'variable', meaning that it can vary during execution. The default is thus `pragma roundingMode(NearestEven)`. In the 'variable' case, a special *ghost* variable is available in annotations, to denote the current mode. Since the first case is the general one, we focus on it in this paper.

Usually, in a program involving FP computations, it is expected that special values for infinities and NaNs should never occur. For this reason we chose that by default, arithmetic overflow should be forbidden so that special values never occur. This first and default situation is called the *defensive* model: it amounts to check that no overflow occur for all FP operations. For programs where special values are indeed expected to appear, we provide another global declaration

```
pragma allowOverflow
```

to switch from the default defensive model to the so-called *full* model. In that case, a set of additional predicates are provided:

- `\is_finite`, `\is_infinite`, `\is_NaN` are unary predicates to test whether an expression of type float or double is either finite, infinite or NaN.
- `\is_positive`, `\is_negative` are similar predicates to test for the sign.
- additional shortcuts are provided, e.g. `\is_plus_infinity`, etc. See [2] for details.
- Comparison predicates between expressions of type float or double, e.g. `\le_float`, `\lt_float`, `\eq_float`, etc.

```
//@ pragma allowOverflow
//@ pragma roundingMode(Down)

typedef struct { double l, u; } interval;
/*@ type invariant is_interval(interval i) =
  @     (\is_finite(i.l) || \is_minus_infinity(i.l)) &&
  @     (\is_finite(i.u) || \is_plus_infinity(i.u)) ;
  @*/
```

Fig. 3. Interval structure and its invariant

Since we chose a standard logic, where functions are total, caution must be taken not to talk about the value of some x if x is not known to be finite. This is a classical issue in specification languages [10].

Example 3. Interval arithmetic aims at computing lower bounds and upper bounds of real expressions. It is a typical example of a FP program that uses a specific rounding mode and makes use of infinite values.

An interval is a structure with two FP fields representing a lower and an upper bound. It represents the sets of all the real numbers between these bounds. Fig. 3 provides a C implementation of such a structure, equipped with a *data invariant* [5] which states that the lower bound might be $-\infty$ and the upper bound might be $+\infty$. The two pragmas at the beginning specify the Down rounding mode and that overflows are expected.

Addition of intervals is given in Fig. 4. A behavior specification for `add` is specified via a predicate `in_interval`(x, i) stating that a real x belongs to an interval i.

```
/*@ predicate double_le_real(double x,real y) =
  @           (\is_finite(x) && x <= y) || \is_minus_infinity(x);
  @ predicate real_le_double(real x,double y) =
  @           (\is_finite(y) && x <= y) || \is_plus_infinity(y);
  @ predicate in_interval(real x,interval i) =
  @   double_le_real(i.l,x) && real_le_double(x,i.u);
  @*/

/*@ ensures \forall real a,b;
  @    in_interval(a,x) && in_interval(b,y) ==>
  @          in_interval(a+b,\result);
  @*/
interval add(interval x, interval y) {
  interval z;
  z.l = x.l + y.l;
  z.u = -(-x.u - y.u);
  return z;
}
```

Fig. 4. Addition of intervals

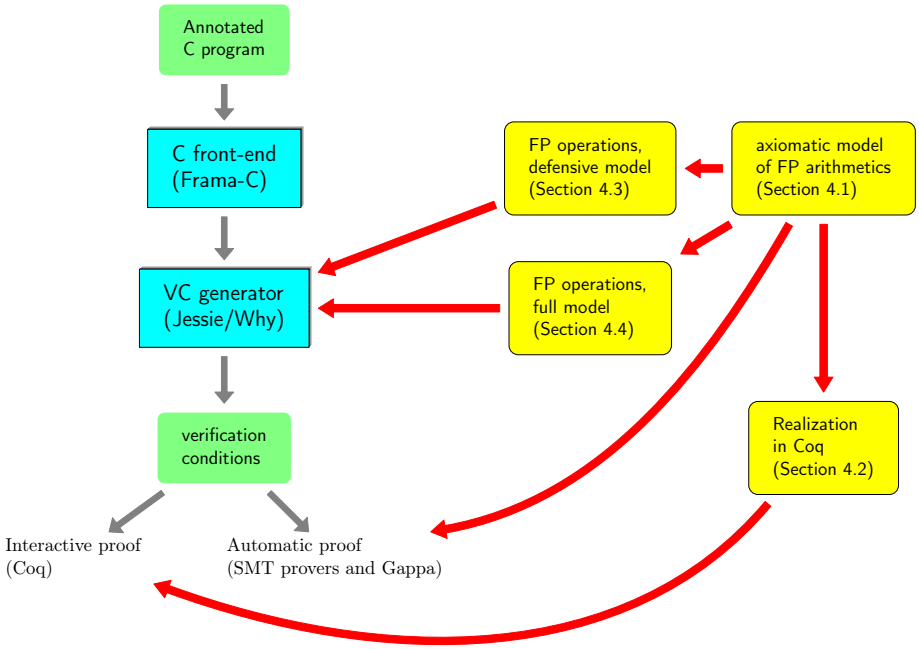

Fig. 5. Architecture of our FP modeling

4 Modeling FP computations

This section is devoted to modeling FP programs and their annotations, in order to reduce soundness to proper VCs. We proceed in four parts, which are schematized on the right part of Fig. 5. The left part represents the current state of the Frama-C setting.

4.1 Axiomatization of FP arithmetics

IEEE-754 characterizes FP formats by describing their bit representation, but for formal reasoning on FP computations, it is better to consider a more abstract view of binary FP numbers: a FP number is a pair of integers (n, e), which denotes the real number $n \times 2^e$, where $n \in \mathbb{Z}$ is the integer significand, and $e \in \mathbb{Z}$ is the exponent.

Example 4. In the `float` 32-bit format, the real number 0.1 is approximated by `0x1.99999Ap-4`, which can be denoted by the pair of integers $(13421773, -27)$

Notice that this representation is not unique, since, e.g, (n, e) and $(2n, e-1)$ represent the same number. The first-order modelization of such numbers starts with a datatype gen_float to denote the domain of numbers $\{n \times 2^e; n, e \in \mathbb{Z}\}$. We declare it *abstractly*, i.e., only by its name and equipped with observation functions:

– float_value : gen_float $\rightarrow \mathbb{R}$, which denotes the real number represented by a FP number

– exact_value : gen_float $\rightarrow \mathbb{R}$ which denotes the exact part of a FP number (as introduced in Section 3.2).

The notion of format is then modelled by a concrete datatype with 2 constructors

$$\mathsf{float\_format} = \mathsf{Single} \mid \mathsf{Double}$$

Indeed, the support for more formats amounts to add new constructors in it.

A suitable characterization of a given FP format f, for our representation with pairs of integers, is provided by a triple $(p, e_{\min}, e_{\max})$ where p is a positive integer called the *precision* of f, and $e_{\min}$ and $e_{\max}$ are two integers which define a range of exponents for f. A number $x = n \times 2^e$ is *representable* in the format f (*f-representable* for short) if n and e satisfy

$$|n| < 2^p \quad \text{and} \quad e_{\min} \leq e \leq e_{\max}.$$

If x is representable, its *canonical representative* is the pair (n, e) satisfying the property above for $|n|$ maximal.[4] The characterization of `float` format is $(24, -149, 104)$ and those of `double` is $(53, -1074, 971)$. The largest f-representable number, denoted $\mathsf{MAX_f}$, is $(2^p - 1)2^{e_{\max}}$. The latter is introduced in our modelling by a function

$$\mathsf{max\_float} : \mathsf{float\_format} \rightarrow \mathbb{R}$$

axiomatized by

$$\mathsf{max\_float}(\mathsf{Single}) = (2^{24} - 1) \times 2^{104}$$
$$\mathsf{max\_float}(\mathsf{Double}) = (2^{53} - 1) \times 2^{971}$$

The five IEEE rounding modes are naturally modelled by a concrete datatype

$$\mathsf{rounding\_mode} = \mathsf{Up} \mid \mathsf{Down} \mid \mathsf{ToZero} \mid \mathsf{NearestAway} \mid \mathsf{NearestEven}$$

In order to express whether an operation overflows or not, we introduce a notion of *unbounded representability* and *unbounded rounding*. The aim is to express what would happen if the exponents were not bounded.

Definition 1. *A FP number $x = n \times 2^e$ is* unbounded f-representable *for format $f = (p, e_{\min,}, e_{\max})$ if:*

$$|n| < 2^p \quad \textit{and} \quad e_{\min} \leq e.$$

The unbounded f, m-rounding operation *for given format f and rounding mode m maps any real number x to the closest (according to m) unbounded f-representable number. We denote that as round$_{f,m}$.*

In our modeling we then introduce an (underspecified) logic function

$$\mathsf{round\_float} : \mathsf{float\_format}, \mathsf{rounding\_mode}, \mathbb{R} \rightarrow \mathbb{R}$$

which is supposed to denote unbounded f, m-rounding. Then, the following predicate indicates when the rounding *does not overflow*:

$$\mathsf{no\_overflow}(f : \mathsf{float\_format}, m : \mathsf{rounding\_mode}, x : \mathbb{R}) := |\mathsf{round\_float}(f, m, x)| \leq \mathsf{max\_float}(f).$$

[4] This definition allows an uniform treatment of *normalized* and *denormalized* numbers [1].

Example 5. In double precision, computing $10^{200} \times 10^{200}$ overflows. In our model, it is represented by $\mathsf{round\_float}(\mathsf{Double}, \mathsf{NearestEven}, 10^{200} * 10^{200})$ which is something close to 10^{400}. It exceeds $\mathsf{max\_float}(\mathsf{Double})$, thus $\mathsf{no\_overflow}(\mathsf{Double}, \mathsf{NearestEven}, 10^{200} * 10^{200})$ is false.

The rounding function round_float is not directly defined. We axiomatize it by some incomplete set of axioms. E.g., the following appear useful in the examples that will come later: $\forall f : \mathsf{float\_format}; m : \mathsf{rounding\_mode}; x, y : \mathbb{R}$,

$$|x| \leq \mathsf{max\_float}(f) \Rightarrow \mathsf{no\_overflow}(f, m, x) \tag{1}$$

$$x \leq y \Rightarrow \mathsf{round\_float}(f, m, x) \leq \mathsf{round\_float}(f, m, y) \tag{2}$$

In order to annotate FP programs that allow overflows and special values, we extend the above logical constructions with new types, predicates and functions. A natural idea would have been to introduce new constants to represent NaN, $+\infty$, $-\infty$. We do not do that for two reasons: First, there are several NaNs, and second, this approach was not suitable to reason with. Our proposal is to add two new functions, which are similar to float_value to give the *class* of a float, either *finite*, *infinite* or *NaN*; and its *sign*. We thus introduce two concrete types

$$\mathsf{Float\_class} = \mathsf{Finite} \mid \mathsf{Infinite} \mid \mathsf{NaN}$$
$$\mathsf{Float\_sign} = \mathsf{Negative} \mid \mathsf{Positive}$$

and two additional functions

$$\mathsf{float\_class} : \mathsf{gen\_float} \rightarrow \mathsf{Float\_class}$$
$$\mathsf{float\_sign} : \mathsf{gen\_float} \rightarrow \mathsf{Float\_sign}$$

which indicate respectively the class and the sign of a gen_float.

Additional predicates are defined to test if a gen_float variable is finite, infinite, NaN, etc. For example,

$$\mathsf{is\_finite}(x : \mathsf{gen\_float}) := \mathsf{float\_class}(x) = \mathsf{Finite}$$

Definitions are similar for is_infinite, is_NaN, is_plus_infinity, is_minus_infinity, etc.

Comparison between two gen_float numbers is given by the predicates le_float, lt_float, eq_float, etc. For example, for x, y : gen_float,

$$\begin{aligned}\mathsf{le\_float}(x, y) \;:=\; & (\mathsf{is\_finite}(x) \;\wedge\; \mathsf{is\_finite}(y) \;\wedge\; \mathsf{float\_value}(x) \leq \mathsf{float\_value}(y)) \\ & \vee\; (\mathsf{is\_minus\_infinity}(x) \wedge\, !\, \mathsf{is\_NaN}(y)) \\ & \vee\; (!\, \mathsf{is\_NaN}(x) \;\wedge\; \mathsf{is\_plus\_infinity}(y))\end{aligned}$$

We must constrain our model to ensure that the sign function is consistent with the sign of real numbers: whenever x represents a finite number, $\mathsf{float\_sign}(x)$ should have the sign of $\mathsf{float\_value}(x)$. This is achieved by the following definitions

$$\begin{aligned}&\mathsf{same\_sign}(x : \mathsf{gen\_float}, y : \mathsf{gen\_float}) := \mathsf{float\_sign}(x) = \mathsf{float\_sign}(y) \\ &\mathsf{diff\_sign}(x : \mathsf{gen\_float}, y : \mathsf{gen\_float}) := \mathsf{float\_sign}(x) \neq \mathsf{float\_sign}(y) \\ &\mathsf{same\_sign\_real}(x : \mathsf{gen\_float}, y : \mathbb{R}) := \\ &\qquad (y < 0 \wedge \mathsf{float\_sign}(x) = \mathsf{Negative}) \vee (y > 0 \wedge \mathsf{float\_sign}(x) = \mathsf{Positive})\end{aligned}$$

and an axiom: $\forall x$: gen_float,

$$(\mathsf{is\_finite}(x) \wedge \mathsf{float\_value}(x) \neq 0) \Rightarrow \mathsf{same\_sign\_real}(x, \mathsf{float\_value}(x)) \quad (3)$$

4.2 A Coq realization of the axiomatic model

Our formalization of FP arithmetic is a first-order, axiomatic one. It is clearly underspecified and incomplete.

We realized this axiomatic model in the Coq proof assistant. This realization has two different goals: first, it allows us to prove the lemmas we added as axioms, thus providing an evidence that our axiomatization is consistent. Second, when dealing with a VC in Coq involving FP arithmetic, we can benefit from all the theorems proven in Coq about FP numbers. ,

Indeed, we build upon the Gappa [5] [24] library which provides:

- a definition of binary finite FP numbers: type float2 (a pair of integers as in section 4.1) together with a function float2R mapping (n, e) to the real $n \times 2^e$;
- a complete definition of a rounding function.

Our realization amounts to declare types float_format, rounding_mode, Float_class and Float_sign as inductive types, and defines max_float by cases. The abstract type gen_float is realized by a Coq record whose fields are:

- genf of type float2;
- The float_value field which is equal to (float2R genf);
- The exact_value field, of type R;
- The float_class field, of type Float_class;
- The float_sign field, of type Float_sign;
- An *invariant* corresponding to axiom (3).

The last field is a noticeable point: it allows us to realize properly the finite_sign axiom above.

Finally, the round_float operator is translated into the corresponding one in Gappa.

4.3 The Defensive Model of FP Computations

To model the effect of the basic FP operations, we need now to make an important assumption: we assume that both the compiler and the processor implement *strict* IEEE-754: any single operation acts as if it first computes a true real number, and then rounds the result to the chosen format, according to the rounding mode. For example, addition of FP numbers is $\mathrm{add}_{f,m}(x, y) = \mathrm{round}_{f,m}(x + y)$ for x, y non-special values, where the $+$ on the right is the mathematical addition of real numbers. This means in particular that addition overflows whenever the rounding overflows. We will discuss this assumption in Section 7.

[5] http://lipforge.ens-lyon.fr/www/gappa/

We model FP operations in FP programs by abstract functions, using the Hoare-style notation

$$f(x_1, \ldots, x_n) : \{P(x_1, .., x_n)\}\tau\{Q(x_1, .., x_n, \mathsf{result})\},$$

which specifies that operation f expects arguments $x_1, \ldots, x_n$ satisfying P (this leads to a VC at each call site) and returns a value r (denoted by keyword result) of type τ, such that $Q(x_1, .., x_n, r)$ holds. In other words, in our modeling we do not say exactly how an operation is performed, but only what is its intended effect.

The defensive model must ensure that no overflows and no NaNs should never occur. This can we done by a proper precondition to operations. Thus, addition of FP numbers is modeled by an abstract function

$$\begin{array}{l}\mathsf{add\_gen\_float}(f : \mathsf{float\_format}, m : \mathsf{rounding\_mode}, x : \mathsf{gen\_float}, y : \mathsf{gen\_float}) : \\ \quad \{\ \mathsf{no\_overflow}(f, m, \mathsf{float\_value}(x) + \mathsf{float\_value}(y))\ \} \\ \quad \mathsf{gen\_float} \\ \quad \{\ \mathsf{float\_value}(\mathsf{result}) = \mathsf{round\_float}(f, m, \mathsf{float\_value}(x) + \mathsf{float\_value}(y)) \wedge \\ \qquad \mathsf{exact\_value}(\mathsf{result}) = \mathsf{exact\_value}(x) + \mathsf{exact\_value}(y)\ \}\end{array}$$

This reads as: the computation of a FP addition requires to check that the result of the addition in $\mathbb{R}$ does not overflow, and it returns a generic float whose real value is the rounding of the real result and exact value is the sum of those of the arguments. Other operations for subtraction, unary negation and multiplication are defined similarly, and also cast operations between float formats. The division additionally requires that the divisor is not zero:

$$\begin{array}{l}\mathsf{div\_gen\_float}(f : \mathsf{float\_format}, m : \mathsf{rounding\_mode}, x : \mathsf{gen\_float}, y : \mathsf{gen\_float}) : \\ \quad \{\ \mathsf{float\_value}(y) \neq 0\ \wedge \mathsf{no\_overflow}(f, m, \mathsf{float\_value}(x)/\mathsf{float\_value}(y))\ \} \\ \quad \mathsf{gen\_float} \\ \quad \{\ \mathsf{float\_value}(\mathsf{result}) = \mathsf{round\_float}(f, m, \mathsf{float\_value}(x)/\mathsf{float\_value}(y)) \wedge \\ \qquad \mathsf{exact\_value}(\mathsf{result}) = \mathsf{exact\_value}(x)/\mathsf{exact\_value}(y)\ \}\end{array}$$

The square root function is defined similarly, requiring that the argument is non-negative.

Notice that, for a given operation in a program, it is known at compile-time, by static typing, which is the expected format of the result. But on the contrary, it should be clarified what is the rounding mode to choose: we use whatever have been declare by the pragma roundingMode in Section 3.

A particular care has to be taken for FP constants literals: they are not necessarily representable and they are rounded (usually at compile-time) to a gen_float according to a certain rounding direction (usually NearestEven). This is modeled by the following abstract function:

$$\begin{array}{l}\mathsf{gen\_float\_of\_real}(f : \mathsf{float\_format}, m : \mathsf{rounding\_mode}, x : \mathbb{R}) : \\ \quad \{\ \mathsf{no\_overflow}(f, m, x)\ \} \\ \quad \mathsf{gen\_float} \\ \quad \{\ \mathsf{float\_value}(\mathsf{result}) = \mathsf{round\_float}(f, m, x)\ \wedge\ \mathsf{exact\_value}(\mathsf{result}) = x\ \}\end{array}$$

This reads as: the real value of the literal must be able to be rounded without overflow, and then the result is the gen_float, whose value is rounded.

Once these operations are specified using those Hoare-style triples, a general purpose verification conditions generator can be executed to build required proof obligations.

Example 6 (Cosine Example 2 continued). On the `cosine` function, 7 VCs are generated:

- 2 VCs to check representability of constants `1.0f` and `0.5f`.
- 3 VCs to check non-overflow of the addition and the two multiplications in the code.
- 1 VC for each postcondition.

For example, the VC for the non-overflow of `x*x` has the form

$$\begin{array}{l}\forall x : \mathsf{gen\_float}, |\mathsf{exact\_value}(x)| \leq 2^{-5} \wedge \\ \quad |\mathsf{float\_value}(x) - \mathsf{exact\_value}(x)| \leq 2^{-20} \Rightarrow \\ \quad \mathsf{no\_overflow}(\mathsf{Single}, \mathsf{NearestEven}, \mathsf{float\_value}(x) * \mathsf{float\_value}(x))\end{array}$$

We will see in Section 5 how to automatically discharge such a VC.

4.4 The Full Model of FP Computations

The *full* model allows FP computations to overflow, and make use of special values: NaNs, infinities and signed zeros.

To fully specify the result of FP operations, in case of overflow, we introduce another predicate

$$\begin{array}{l}\mathsf{overflow\_value}(f : \mathsf{float\_format}, m : \mathsf{rounding\_mode}, x : \mathsf{gen\_float}) := \\ \quad (m = \mathsf{Down} \Rightarrow \\ \qquad (\mathsf{float\_sign}(x) = \mathsf{Negative} \Rightarrow \mathsf{is\_infinite}(x)) \wedge \\ \qquad (\mathsf{float\_sign}(x) = \mathsf{Positive} \Rightarrow \mathsf{is\_finite}(x) \wedge \mathsf{float\_value}(x) = \mathsf{max\_float}(f))) \\ \wedge (m = \mathsf{Up} \Rightarrow \\ \qquad (\mathsf{float\_sign}(x) = \mathsf{Negative} \Rightarrow \mathsf{is\_finite}(x) \wedge \mathsf{float\_value}(x) = -\mathsf{max\_float}(f)) \wedge \\ \qquad (\mathsf{float\_sign}(x) = \mathsf{Positive} \Rightarrow \mathsf{is\_infinite}(x))) \\ \wedge (m = \mathsf{ToZero} \Rightarrow \mathsf{is\_finite}(x) \wedge \\ \qquad (\mathsf{float\_sign}(x) = \mathsf{Negative} \Rightarrow \mathsf{float\_value}(x) = -\mathsf{max\_float}(f)) \wedge \\ \qquad (\mathsf{float\_sign}(x) = \mathsf{Positive} \Rightarrow \mathsf{float\_value}(x) = \mathsf{max\_float}(f))) \\ \wedge (m = \mathsf{NearestAway} \ \vee \ m = \mathsf{NearestEven} \Rightarrow \mathsf{is\_infinite}(x))\end{array}$$

This predicate specifies the float class of an overflowing x depending on its sign, and the format and the rounding mode, as indicated by the IEEE standard.

Unlike for the defensive model, there are no preconditions on the unary and binary operations. Here we show the complete specification for the two operations of multiplication and less-than comparison. See [2] for the complete description.

Multiplication is specified in Figure 6 where

$$\begin{array}{l}\mathsf{is\_gen\_zero}(x : \mathsf{gen\_float}) := \mathsf{float\_class}(x) = \mathsf{Finite} \ \wedge \mathsf{float\_value}(x) = 0 \\ \mathsf{product\_sign}(z : \mathsf{gen\_float}, x : \mathsf{gen\_float}, y : \mathsf{gen\_float}) := \\ \qquad\qquad (\mathsf{same\_sign}(x, y) \Rightarrow \mathsf{float\_sign}(z) = \mathsf{Positive}) \wedge \\ \qquad\qquad (\mathsf{diff\_sign}(x, y) \Rightarrow \mathsf{float\_sign}(z) = \mathsf{Negative})\end{array}$$

```
mul_gen_float(f : float_format, m : rounding_mode, x : gen_float, y : gen_float) :
  { // no preconditions }
  gen_float
  { ((is_NaN(x) ∨ is_NaN(y)) ⇒ is_NaN(result))
          // NaNs arguments propagate to the result
  ∧ ((is_gen_zero(x) ∧ is_infinite(y)) ⇒ is_NaN(result))
  ∧ ((is_infinite(x) ∧ is_gen_zero(y)) ⇒ is_NaN(result))
          // zero times ∞ gives NaN
  ∧ ((is_finite(x) ∧ is_infinite(y) ∧ float_value(x) ≠ 0) ⇒ is_infinite(result))
  ∧ ((is_infinite(x) ∧ is_finite(y) ∧ float_value(y) ≠ 0) ⇒ is_infinite(result))
          // ∞ times non-zero finite gives ∞
  ∧ ((is_infinite(x) ∧ is_infinite(y)) ⇒ is_infinite(result))
          // ∞ times ∞ gives ∞
  ∧ ((is_finite(x) ∧ is_finite(y) ⇒
      if no_overflow(f, m, float_value(x) × float_value(y))) then
      (is_finite(result)∧
      float_value(result) = round_float(f, m, float_value(x) × float_value(y)))
          // finite times finite without overflow
      else (overflow_value(f, m, result)))
          // finite times finite with overflow
  ∧ product_sign(result, x, y)
          // in any case, sign of result is product of signs
  ∧ exact_value(result) = exact_value(x) × exact_value(y)
  }
```

Fig. 6. Multiplication in the full model

```
lt_gen_float(x : gen_float, y : gen_float) :
  {// no preconditions}
  bool
  {if result then not is_NaN(x) ∧ not is_NaN(y) ∧
       ( (is_finite(x) ∧ is_finite(y) ∧ float_value(x) < float_value(y))∨
         (is_minus_infinity(x) ∧ is_plus_infinity(y))∨
         (is_minus_infinity(x) ∧ is_finite(y))∨
         (is_finite(x) ∧ is_plus_infinity(y)))
   else is_NaN(x) ∨ is_NaN(y)∨
       (is_finite(x) ∧ is_finite(y) ∧ float_value(x) ≥ float_value(y)) ∨
       is_plus_infinity(x) ∨ is_minus_infinity(y)
  }
```

Fig. 7. abstract less-than in full model

the last predicate encoding the usual rule for sign of a product.

Other operations are specified similarly. Comparison operators are also significantly complicated to specify, this is illustrated by the specification of $<$ given in Figure 7.

5 Discharging proof obligations

Our aim is to use as many theorem provers as possible. However, we must consider provers that are able to understand first-order logic with integer and real arithmetic. Suitable automatic provers are those of the SMT-family (Satisfiability Modulo Theories) [26] which support first-order quantification, such as Z3 [14], CVC3 [4], Yices [16], Alt-Ergo [12]. Due to the high expressiveness of the logic, these provers are necessarily incomplete. Hence we may also use interactive theorem provers, such as Coq and PVS.

Additionally, recall that our modeling involves an uninterpreted rounding function round_float. The Gappa tool [25] is an automatic prover, which specifically handles formulas made of equalities and inequalities over expressions involving real constants, arithmetic operations, and the round_float operator.

All the provers mentioned above are available as back-ends for the Frama-C environment and its Jessie/Why plugin [17]. Our experiments are conducted with those.

Example 7 (Example 1 continued). The VCs for our Remez approximation of exponential are the following:

- 3 VCs for the representability of constants `0.9890365552`; `1.130258690` and `0.5540440796` in double format. These are proven by Gappa and by SMT solvers. SMT solvers make use of the axiom (1) on round_float.
- 5 VCs for checking that the three multiplications and the two additions do not overflow. These are automatically proven by Gappa. This demonstrates the power of Gappa to check non-overflow of FP computations in practice.
- 1 VC for the validity of the post-condition. This is also proven by Gappa, as a consequence of the assertion. In other words, whenever Gappa is given the method error, it is able to add the rounding error to deduce the total error.
- 1 VC for the validity of the assertion stating the method error. This is not proven by any automatic prover. It corresponds to the VC:

$$\begin{array}{l}\forall x : \mathsf{gen\_float}, |\mathsf{float\_value}(x)| \leq 1.0 \Rightarrow \\ \quad |0.9890365552 + 1.130258690 \times \mathsf{float\_value}(x) + 0.5540440796 \times \\ \qquad \mathsf{float\_value}(x) * \mathsf{float\_value}(x) - \exp(\mathsf{float\_value}(x))| \leq (1 - 2^{-16}) \times 2^{-4}\end{array}$$

  Indeed, $\mathsf{float\_value}(x)$ is just an arbitrary real number here, and that formula is a pure real arithmetic formula. It is expected that no automatic prover prove it since they do not know anything about the exp function. However, this VC can be proven valid using the Coq proof assistant, in a very simple way (2 lines of proof script to write) thanks to its `interval` tactic[6] [25], which is able to bound mathematical expressions using interval arithmetic:

```
intuition.
interval with (i_bisect_diff (float_value x),i_nocheck).
```

Example 8 (Cosine example continued).

[6] `http://www.msr-inria.inria.fr/~gmelquio/soft/coq-interval/`

```
/*@ lemma abs_triangle: \forall real x,y,z;
  @             \abs(x-y) <= \abs(x-z)+\abs(z-y);
  @ lemma cos_lipschitz: \forall real x,y;
  @             \abs(\cos(x)-\cos(y)) <= \abs(x-y);
  @*/

/*@ requires \abs(\exact(x)) <= 0x1p-5;
  @ requires \round_error(x) <= 0x1p-20;
  @ ensures \abs(\exact(\result) - \cos(\exact(x))) <= 0x1p-24;
  @ ensures \round_error(\result) <= \round_error(x) + 0x3p-24;
  @*/
float cosine(float x) {
  //@ assert a1: \abs(x) <= 0x1.0002p-5;
  float r = 1.0f - x * x * 0.5f;
  //@ assert a2: \abs(\exact(r) - \cos(\exact(x))) <= 0x1p-24;
  //@ assert a3: \abs(1.0 - x*x*0.5 - \cos(x)) <= 0x1p-24;
  //@ assert a4: \abs(r - \cos(x)) <= 0x1p-23;
  return r;
}
```

Fig. 8. Additional lemmas and code assertions for the cosine example

To acheive the proofs of Example 2, we had to provide some help to provers under the form of extra lemmas and also a few assertions in the code, see Figure 8. The verification conditions are now:

- Lemma `abs_triangle` is proved by most SMT provers.
- Lemma `cos_lipschitz` is proved in Coq.
- Assertion `a1` is proved by SMT provers from lemma `abs_triangle` and preconditions.
- Assertions `a2` and `a3` are proved by interval tactic in Coq.
- Assertion `a4` is proved by Gappa using `a3`.
- The 5 VCs for checking non-overflow are proved automatically by Gappa (using assertion `a1`)
- The first postcondition is proved by SMT provers (from `a2`).
- The second postcondition is proved by the Z3 prover, thanks to both lemmas and assertions `a4` and `a2`.

Example 9 (Interval example continued). Although the code for interval addition (Fig. 4) is very simple, its verification was not easy because it involves a large amount of different cases to distinguish, depending on whether interval bounds are finite or infinite, and whether an overflow occurs or not. Our first attempt was to perform the proof entirely within Coq: it was very long but indeed without particular difficulty so that clearly automatic provers should do most of it. We instead tried to find intermediate lemmas to give to SMT solvers to help them. We succeeded to prove all VCs automatically with SMT provers using the following lemmas: for all format f, modes m_1, m_2 and real x:

$$\begin{aligned}
\mathsf{round_float}(f, \mathsf{Up}, x) &\geq x \\
\mathsf{round_float}(f, \mathsf{Up}, -x) &= -\mathsf{round_float}(f, \mathsf{Down}, x) \\
\mathsf{round_float}(f, m_1, \mathsf{round_float}(f, m_2, x)) &= \mathsf{round_float}(f, m_2, x)
\end{aligned}$$

As before, these can be checked valid using our Coq realization.

Example 10 (Interval multiplication).

To go further, we proved also the multiplication of intervals. Its code is given in Fig. 9. Notice the large number of branches. This code calls some intermediate functions on intervals from Fig. 10. This was difficult to verify. First, we had to find proper contracts for the intermediate functions: see the preconditions we need about signs for `mul_up` and `mul_dn`. Second the number of cases is even more larger than for addition. We got a total of 143 VCs. Each of them has a complex propositional structure, leading to consider a large number of subcases. Like for addition, by investigating the VCs which were not proved automatically, we were able to discover a few lemmas that helped the automatic provers: for all reals x, y, z and t:

$$\begin{aligned}
(0 \leq x \leq z \wedge 0 \leq y \leq t) &\Rightarrow x * y \leq z * t \\
(0 \leq z \leq x \wedge y \leq t \wedge y < 0) &\Rightarrow x * y \leq z * t \\
(x \leq z \leq 0 \wedge y \leq t \wedge y < 0) &\Rightarrow z * t \leq x * y
\end{aligned}$$

With that all VCs are automatically proved. We clearly benefit from the ability of SMT provers to deal with all the cases of the complex propositional structures.

6 Related Works

There exist several formalizations of FP arithmetic in various proof environments: two variants in Coq [13, 24] and one in PVS[7] exclude special values; one in ACL2 [22] and one in HOL-light [21] also deal with special values. Compare to those, our purely first-order axiomatization has the clear disadvantage of being incomplete, but has the advantage of allowing use of off-the-shelf automatic theorem provers. Our approach allows to incorporate FP reasoning in environments for program verification for general-purpose programming languages like C or Java.

In 2006, Leavens [23] described some pitfalls when trying to incorporate FP special values and specifically NaN values in a BISL like JML for Java. In its approach, FP numbers, rounding and such also appear in annotations, which cause several issues and traps for specifiers. We argue that our approach, using true real numbers instead in annotations, solves these kind of problems.

In 2006, Reeber & Sawada [27] used the ACL2 system together with a automated tool to verify a FP multiplier unit. Although their goal is at a significantly different concern (hardware verification instead of software behavioral properties) it is interesting to remark that they came to a similar conclusion, that using interactive proving alone is not practicable, but incorporating an automatic tool is successful.

[7] `http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/float-details.html`

```
/*@ ensures \forall real a,b;
  @   in_interval(a,x) && in_interval(b,y) ==>
  @        in_interval(a*b,\result);
  @*/
interval mul(interval x, interval y) {
  interval z;
  if (x.l < 0.0)
    if (x.u > 0.0)
      if (y.l < 0.0)
        if (y.u > 0.0) {
          z.l = min(mul_dn(x.l, y.u), mul_dn(x.u, y.l));
          z.u = max(mul_up(x.l, y.l), mul_up(x.u, y.u)); }
        else { z.l = mul_dn(x.u, y.l); z.u = mul_up(x.l, y.l); }
      else
        if (y.u > 0.0)
          { z.l = mul_dn(x.l, y.u); z.u = mul_up(x.u, y.u); }
        else { z.l = 0.0; z.u = 0.0; }
    else
      if (y.l < 0.0)
        if (y.u > 0.0)
          { z.l = mul_dn(x.l, y.u); z.u = mul_up(x.l, y.l); }
        else { z.l = mul_dn(x.u, y.u); z.u = mul_up(x.l, y.l); }
      else
        if (y.u > 0.0)
          { z.l = mul_dn(x.l, y.u); z.u = mul_up(x.u, y.l); }
        else { z.l = 0.0; z.u = 0.0; }
  else
    if (x.u > 0.0)
      if (y.l < 0.0)
        if (y.u > 0.0)
          { z.l = mul_dn(x.u, y.l); z.u = mul_up(x.u, y.u); }
        else { z.l = mul_dn(x.u, y.l); z.u = mul_up(x.l, y.u); }
      else
        if (y.u > 0.0)
          { z.l = mul_dn(x.l, y.l); z.u = mul_up(x.u, y.u); }
        else { z.l = 0.0; z.u = 0.0; }
    else { z.l = 0.0; z.u = 0.0; }
  return z;
}
```

Fig. 9. Multiplication of intervals

7 Conclusions and further work

Our methodology for verification of properties of FP programs is based on a first-order logic modeling of FP numbers, and interpretations of FP computations as Hoare-style pre- and postconditions. This approach is suitable for deductive verification tools such

```
/*@ requires !\is_NaN(x) && !\is_NaN(y);
  @ ensures \le_float(\result,x) && \le_float(\result,y);
  @ ensures \eq_float(\result,x) || \eq_float(\result,y);
  @*/
double min(double x, double y) { return x < y ? x : y; }

/*@ requires !\is_NaN(x) && !\is_NaN(y);
  @ ensures \le_float(x,\result) && \le_float(y,\result);
  @ ensures \eq_float(\result,x) || \eq_float(\result,y);
  @*/
double max(double x, double y) { return x > y ? x : y; }

/*@ requires !\is_NaN(x) && !\is_NaN(y);
  @ requires (\is_infinite(x) || \is_infinite(y))
  @                              ==> \sign(x) != \sign(y);
  @ requires (\is_infinite(x) && \is_finite(y)) ==> y != 0.0;
  @ requires (\is_infinite(y) && \is_finite(x)) ==> x != 0.0;
  @ ensures double_le_real(\result,x*y);
  @ ensures (\is_infinite(x) || \is_infinite(y)) ==>
  @            \is_minus_infinity(\result);
  @*/
double mul_dn(double x, double y) { return x*y; }

/*@ requires !\is_NaN(x) && !\is_NaN(y);
  @ requires (\is_infinite(x) || \is_infinite(y))
  @                              ==> \sign(x) == \sign(y);
  @ requires (\is_infinite(x) && \is_finite(y)) ==> y != 0.0;
  @ requires (\is_infinite(y) && \is_finite(x)) ==> x != 0.0;
  @ ensures  real_le_double(x * y,\result);
  @ ensures (\is_infinite(x) || \is_infinite(y)) ==>
  @                        \is_plus_infinity(\result);
  @*/
double mul_up(double x, double y)
{ double a=-y;
  //@ assert \is_finite(y) ==> \is_finite(a) && a == - y ;
  double b=x*a;
  double z=-b;
  //@ assert \is_finite(b) ==> \is_finite(z) && z == - b ;
   return z;
}
```

Fig. 10. Intermediate functions on intervals

as Spec#/Boogie, VCC/Boogie, ESCJava2, KeY, Frama-C/Why. It allows to prove complex properties of FP programs, specified in a very expressive specification language.

In Section 5, we have seen that SMT provers are not very good at proving properties about rounding. Our attempt to add more axioms in our modeling did not succeed, and the Gappa tool was necessary to prove those properties. An interesting future work is

to turn the Gappa approach into some specific built-in theory for SMT solvers. More generally, as shown by our examples, the success of our approach relies on the ability to mix automatic and interactive theorem proving. Ideally, one would like to process a proof inside an interactive environment like Coq, where at any time, on every subgoal, one could call powerful dedicated provers like SMT provers, Gappa for rounding errors, or interval arithmetic.

Our approach provides two models of FP operations: a defensive one which requires to prove the absence of overflow, and a full model allowing overflows and presence of special values. Since the pure logical part of the modeling is shared, it is clearly possible to mix the two possible settings in a given program: most functions should be overflow-free whereas a few of them might produce overflows. This could be done in practice by providing a specific clause in the annotation language. However, this mixing should be done carefully, e.g., if a function allowing overflow calls a function supposed to work in defensive mode, appropriate preconditions of the form $\mathsf{is\_finite}(x)$ for each float parameter x should be added.

Acknowledgements

We gratefully thank Guillaume Melquiond for his help in the use of the Gappa tool, the FP-specific Coq tactics, and more generally for his suggestions about the approach presented here.

References

1. IEEE standard for floating-point arithmetic. Technical report, 2008. `http://dx.doi.org/10.1109/IEEESTD.2008.4610935`.
2. A. Ayad. On formal methods for certifying floating-point C programs. Technical report, INRIA, 2009. `http://www.lri.fr/~ayad/floats.pdf`.
3. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# Programming System: An Overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2004.
4. C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07), Berlin, Germany*, Lecture Notes in Computer Science. Springer, 2007.
5. P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL: ANSI/ISO C Specification Language*, 2008. `http://frama-c.cea.fr/acsl.html`.
6. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *Lecture Notes in Computer Science*. Springer, 2007.
7. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE static analyzer. `http://www.astree.ens.fr/`.
8. S. Boldo and J.-C. Filliâtre. Formal Verification of Floating-Point Programs. In *18th IEEE International Symposium on Computer Arithmetic*, pages 187–194, Montpellier, France, June 2007.
9. L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 2004.

10. P. Chalin. Reassessing JML's logical foundation. In *Proceedings of the 7th Workshop on Formal Techniques for Java-like Programs (FTfJP'05)*, Glasgow, Scotland, July 2005.
11. D. R. Cok and J. R. Kiniry. ESC/Java2 implementation notes. Technical report, may 2007. `http://secure.ucd.ie/products/opensource/ESCJava2/ESCTools/docs/Escjava2-ImplementationNotes.pdf`.
12. S. Conchon, E. Contejean, J. Kanig, and S. Lescuyer. CC(X): Semantical Combination of Congruence Closure with Solvable Theories. In *Proceedings of the 5th International Workshop on Satisfiability Modulo Theories (SMT 2007)*, volume 198-2 of *Electronic Notes in Computer Science*, pages 51–69. Elsevier Science Publishers, 2008.
13. M. Daumas, L. Rideau, and L. Théry. A generic library for floating-point numbers and its application to exact computing. In *Theorem Proving in Higher Order Logics (TPHOLs'01)*, volume 2152 of *Lecture Notes in Computer Science*, pages 169+, 2001.
14. L. de Moura and N. Bjørner. Z3, An Efficient SMT Solver. `http://research.microsoft.com/projects/z3/`.
15. E. W. Dijkstra. *A discipline of programming*. Series in Automatic Computation. Prentice Hall Int., 1976.
16. B. Dutertre and L. de Moura. The YICES SMT Solver. available at `http://yices.csl.sri.com/tool-paper.pdf`, 2006.
17. J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *Computer Aided Verification (CAV)*, volume 4590 of *LNCS*, pages 173–177. Springer, July 2007.
18. The Frama-C platform for static analysis of C programs, 2008. `http://www.frama-c.cea.fr/`.
19. D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.
20. E. Goubault and S. Putot. Static analysis of numerical algorithms. In K. Yi, editor, *Static Analysis Symposium 2006*, volume 4134 of *Lecture Notes in Computer Science*, pages 18–34. Springer, 2006.
21. J. Harrison. Floating point verification in hol light: The exponential function. *Form. Methods Syst. Des.*, 16(3):271–305, 2000.
22. T. L. J. Strother Moore and M. Kaufmann. A mechanically checked proof of the correctness of the kernel of the amd5k86 floating-point division algorithm. *IEEE Transactions on Computers*, 47(9):913–926, 1998.
23. G. Leavens. Not a number of floating point problems. *Journal of Object Technology*, 5(2):75–83, 2006.
24. G. Melquiond. Floating-point arithmetic in the Coq system. In *Proceedings of the 8th Conference on Real Numbers and Computers*, pages 93–102, Santiago de Compostela, Spain, 2008.
25. G. Melquiond. Proving bounds on real-valued functions with computations. In A. Armando, P. Baumgartner, and G. Dowek, editors, *Proceedings of the 4th International Joint Conference on Automated Reasoning*, volume 5195 of *Lecture Notes in Artificial Intelligence*, pages 2–17, Sydney, Australia, 2008.
26. S. Ranise and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `http://www.smtcomp.org`, 2006.
27. E. Reeber and J. Sawada. Combining acl2 and an automated verification tool to verify a multiplier. In *Sixth International Workshop on the ACL2 Theorem Prover and its Applications*, pages 63–70. ACM, 2006.
28. W. Schulte, S. Xia, J. Smans, and F. Piessens. A glimpse of a verifying C compiler. `http://www.cs.ru.nl/~tews/cv07/cv07-smans.pdf`.
29. The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.2*, 2008. `http://coq.inria.fr`.