An SMT-LIB Theory of Binary Floating-Point Arithmetic[*]

Philipp Rümmer and Thomas Wahl
Oxford University Computing Laboratory, United Kingdom

Abstract

Floating-point arithmetic is an essential ingredient of embedded systems, such as in the avionics and automotive industries. By nature, many of these applications are safety-critical, requiring rigorous mathematical methods such as model checking to verify the adherence to safety standards. One of the bottlenecks in comparing different approaches to the floating-point program verification problem is the lack of a standardised formal language to interface with SMT and constraint solvers. In this paper, we propose a theory, FPA, of floating-point arithmetic for the recently released SMT-LIB 2.0 standard. We rigorously define the semantics of FPA, following the IEEE binary floating-point standard 754-2008. We motivate our design decisions and deviations from the IEEE standard. The long-term goal is the development of SMT solvers with FPA support, as well as a set of FPA benchmarks in the SMT-LIB format that allow comparative studies of floating-point verification techniques.

1 Introduction

Many embedded systems are controlled by software that conceptually manipulates real-valued quantities, for instance measurements of environment data. Such quantities are stored in a computer as floating-point numbers. As computer memory is bounded, calculations with floating-point numbers can at best approximate real-valued calculations. The resulting floating-point calculus has become known as *floating-point arithmetic*.

Embedded software is common in mobile systems, such as in aircraft and automobiles. By nature, these systems are safety-critical; their design must abide by the highest reliability standards. Errors due to floating-point peculiarities such as rounding tend to be particularly unintuitive and are often system-dependent and hard to test for [1]. Techniques most promising for ensuring correctness are rigorous formal methods such as abstract interpretation, theorem proving, or model checking.

The automated verification of floating-point programs is, however, still in its infancy. We contribute this state of the art not only to the (unavoidable) computational complexity of this problem, but also to the lack of SMT solvers that can handle floating-point arithmetic, and the lack of a set of benchmarks that would enable the comparison of results obtained using various verification methods.

To address these problems, this paper proposes a theory, denoted FPA, of floating-point arithmetic for the (recently released) SMT-LIB 2.0 standard [2] that can serve as a reference point for benchmark design. To support the development of tools, the paper gives a self-contained and rigorous definition of the FPA syntax and semantics, largely following the IEEE standard 754-2008 [3]. Our proposal concentrates on arithmetic aspects, abstracting from more operational topics such as exception handling. Also, we only consider the case of binary (as opposed to decimal) floating-point arithmetic, which is most widely used in practice. As a result, our theory avoids the very problems that render the standard unsuitable as a direct reference in developing floating-point solvers or benchmarks, namely its complexity, and the occasional lack of rigour.

Our formalisation supports all aspects of floating-point arithmetic that are relevant for a logical theory: rounding modes, positive and negative infinity and zeroes, the not-a-number quantity, and arithmetic and comparison operations. Deviations from the IEEE standard are mentioned and motivated below.

[*]This research is supported by the EPSRC project EP/G026254/1, and by the EU ARTEMIS CESAR project.

Related work. The computational complexity of exhaustive floating-point program verification mentioned above has, in the past, mostly been addressed by resorting to interactive analysis using proof assistants [4], or to abstract interpretation [5, 1]. A problem common to these approaches is the lack of evidence returned in case the given property could not be proved, which may be due to it not being true, due to the imprecision of the abstraction, or due to the weakness of the proof strategy.

To counter these deficiencies, automated reasoning tools and model checkers are being developed [6, 7]. The work in [7], however, reduces the floating-point verification problem to a propositional SAT problem and does not employ an SMT solver. Among the many theorem proving approaches to program analysis that require modelling floating-point operations [8, 6] (see [9] for an overview), the authors of [6] propose an axiomatic formalisation of FPA. They model floating-point operations via rounding applied to the result of the corresponding real-arithmetic operations, similar to our approach. The major difference is that our formalisation follows the SMT-LIB standard and should thus be able to assist the development of benchmarks for automated provers and model checkers alike. We also believe our formalisation to be closer to the IEEE standard; for example, we directly support rounding functions, and represent non-real quantities like $+\infty$ as explicit constant symbols, rather than implicitly via predicates.

2 Syntax of FPA

The next pages define the FPA syntax. We assume that the reader has some familiarity with the SMT-LIB standard [2] and the basic concepts of floating-point arithmetic. The complete syntax of our FPA formalisation (sorts and operations) is given in Appendix A. In the following two subsections, we comment on these definitions and provide intuition for the symbols defined; for their precise semantics, see Section 3.

2.1 Sorts of FPA

The floating-point sorts. Floating-point sorts are represented as indexed nullary sort identifiers of the form `(_ FP` ⟨*ebits*⟩ ⟨*sbits*⟩`)`, where `FP` symbolises the floating-point sorts, and the indices ⟨*ebits*⟩ and ⟨*sbits*⟩ are positive integers. Intuitively, ⟨*ebits*⟩ and ⟨*sbits*⟩ define the number of available bits in the exponent and in the significand, respectively, of a particular floating-point sort. For example, `(_ FP 11 53)` represents the *binary64* format of IEEE 754-2008.

We introduce constants for distinguished floating-point numbers, for all positive integers ⟨*ebits*⟩ and ⟨*sbits*⟩:

```
(plusInfinity  (_ FP ⟨ebits⟩ ⟨sbits⟩))          (NaN (_ FP ⟨ebits⟩ ⟨sbits⟩))
(minusInfinity (_ FP ⟨ebits⟩ ⟨sbits⟩))
```

Given ⟨*ebits*⟩ and ⟨*sbits*⟩, i.e., one fixed floating-point sort, there are exactly one `FP` representing, intuitively, $+\infty$, and exactly one `FP` representing $-\infty$, in agreement with the IEEE standard. Similarly, given ⟨*ebits*⟩ and ⟨*sbits*⟩, there is exactly one `NaN`, representing the not-a-number quantity, such as obtained when adding $+\infty$ and $-\infty$. This simplifies the IEEE standard in that we do not distinguish *signalling* and *quiet* `NaN`s. The former are used in operations that may raise exceptions, which are not part of this FPA theory.

Rounding Modes. Our formalisation supports all five rounding modes introduced by the IEEE standard. In the standard, the rounding mode is an attribute associated with a program block. In our theory, it is a parameter to many (but not all) arithmetic operations:

```
:sorts ((RoundingMode 0))
:funs ((roundNearestTiesToEven RoundingMode)
       (roundNearestTiesToAway RoundingMode)
       (roundTowardPositive    RoundingMode)
       (roundTowardNegative    RoundingMode)
       (roundTowardZero        RoundingMode))
```

2.2 Operations of FPA

Binary arithmetic. For all positive integers ⟨*ebits*⟩ and ⟨*sbits*⟩, we define operations that take two FP numbers and return, intuitively, their sum, difference, etc.:

```
(+ RoundingMode (_ FP ⟨ebits⟩ ⟨sbits⟩) (_ FP ⟨ebits⟩ ⟨sbits⟩) (_ FP ⟨ebits⟩ ⟨sbits⟩))
(- RoundingMode (_ FP ⟨ebits⟩ ⟨sbits⟩) (_ FP ⟨ebits⟩ ⟨sbits⟩) (_ FP ⟨ebits⟩ ⟨sbits⟩))
(* RoundingMode (_ FP ⟨ebits⟩ ⟨sbits⟩) (_ FP ⟨ebits⟩ ⟨sbits⟩) (_ FP ⟨ebits⟩ ⟨sbits⟩))
(/ RoundingMode (_ FP ⟨ebits⟩ ⟨sbits⟩) (_ FP ⟨ebits⟩ ⟨sbits⟩) (_ FP ⟨ebits⟩ ⟨sbits⟩))
```

Here, `-` stands for the binary minus. The results of these operations depend on the rounding mode, which is passed as their first argument. Note that each of the symbols is overloaded and can receive FP numbers of arbitrary bit-widths.

Basic unary operations. The following operations accept and return a FP number:

```
(abs (_ FP ⟨ebits⟩ ⟨sbits⟩) (_ FP ⟨ebits⟩ ⟨sbits⟩))
(-   (_ FP ⟨ebits⟩ ⟨sbits⟩) (_ FP ⟨ebits⟩ ⟨sbits⟩))
```

Intuitively, these return the absolute value and the negation of a FP number. Because the operations do not require rounding, no rounding mode is given as an argument (similarly to comparison operators, zero-tests, and `min`/`max` operations defined below).

Remainder. The following operation takes two FP numbers x and y and returns $x - y * n$, where n is the integer nearest to x/y (using the rounding mode *ties to even* defined in Section 3.2):

```
(remainder (_ FP ⟨ebits⟩ ⟨sbits⟩) (_ FP ⟨ebits⟩ ⟨sbits⟩) (_ FP ⟨ebits⟩ ⟨sbits⟩))
```

Ternary arithmetic. The following operation takes three FP numbers x, y, z and computes $(x * y) + z$. This is known as *fused multiply-add*:

```
(fusedMA RoundingMode (_ FP ⟨ebits⟩ ⟨sbits⟩) (_ FP ⟨ebits⟩ ⟨sbits⟩) (_ FP ⟨ebits⟩ ⟨sbits⟩)
         (_ FP ⟨ebits⟩ ⟨sbits⟩))
```

Square root and explicit rounding. The following unary operations intuitively compute the square root and the nearest integral FP number of their argument and depend on the rounding mode:

```
(squareRoot      RoundingMode (_ FP ⟨ebits⟩ ⟨sbits⟩) (_ FP ⟨ebits⟩ ⟨sbits⟩))
(roundToIntegral RoundingMode (_ FP ⟨ebits⟩ ⟨sbits⟩) (_ FP ⟨ebits⟩ ⟨sbits⟩))
```

Binary comparison operations. The following operations compare two FP numbers. They return a Boolean and thus do not require rounding:

```
(==  (_ FP ⟨ebits⟩ ⟨sbits⟩) (_ FP ⟨ebits⟩ ⟨sbits⟩) Bool :chainable)
(<=  (_ FP ⟨ebits⟩ ⟨sbits⟩) (_ FP ⟨ebits⟩ ⟨sbits⟩) Bool :chainable)
(<   (_ FP ⟨ebits⟩ ⟨sbits⟩) (_ FP ⟨ebits⟩ ⟨sbits⟩) Bool :chainable)
(>=  (_ FP ⟨ebits⟩ ⟨sbits⟩) (_ FP ⟨ebits⟩ ⟨sbits⟩) Bool :chainable)
(>   (_ FP ⟨ebits⟩ ⟨sbits⟩) (_ FP ⟨ebits⟩ ⟨sbits⟩) Bool :chainable)
```

When comparing FPs for equality, the IEEE 754-2008 standard treats *NaN*s and signed zeros specially (see Section 3.3). We therefore introduce a second equality operator == in addition to the pre-defined SMT-LIB operator =. We declare the operations as `chainable`: they can be used with any number of arguments, as in `(==` x_1 x_2 x_3`)`, denoting a conjunction `(and (==` x_1 x_2`) (==` x_2 x_3`))`.

Unary comparison operations: zero-tests. The following operations return true exactly if, intuitively, their FP argument is zero-valued with either sign, has a negative sign, is a negative zero, or is a positive zero, respectively. For the precise semantics and meaning of "positive zero", etc., see Section 3 below. These operations return a Boolean and thus do not require rounding.

```
(isZero      (_ FP ⟨ebits⟩ ⟨sbits⟩) Bool)     (isNZero (_ FP ⟨ebits⟩ ⟨sbits⟩) Bool)
(isSignMinus (_ FP ⟨ebits⟩ ⟨sbits⟩) Bool)     (isPZero (_ FP ⟨ebits⟩ ⟨sbits⟩) Bool)
```

Minimum/Maximum operations. The following operations compare their arguments and return one of them; they thus do not require rounding:

```
(min (_ FP ⟨ebits⟩ ⟨sbits⟩) (_ FP ⟨ebits⟩ ⟨sbits⟩) (_ FP ⟨ebits⟩ ⟨sbits⟩))
(max (_ FP ⟨ebits⟩ ⟨sbits⟩) (_ FP ⟨ebits⟩ ⟨sbits⟩) (_ FP ⟨ebits⟩ ⟨sbits⟩))
```

2.3 Decimal Literals of FPA

We allow concrete (literal) floating-point numbers to be specified in decimal notation. To this end, we first define that decimal literals evaluate to elements of a sort `Rational` of rational numbers:

```
:sorts ((Rational 0))
:funs ((DECIMAL Rational))
```

Rational numbers can then be converted to floating-point numbers (which can involve rounding):

```
((_ asFloat ⟨ebits⟩ ⟨sbits⟩) RoundingMode Rational (_ FP ⟨ebits⟩ ⟨sbits⟩))
```

2.4 Example

The following formula expresses that every 64-bit FP number less than $+\infty$ is strictly less than some square:

```
(forall ((x (_ FP 11 53)))
        (=> (< x (as plusInfinity (_ FP 11 53)))
            (exists ((y (_ FP 11 53))) (< x (* roundTowardZero y y)))))
```

According to the SMT-LIB standard, the type of the constant `plusInfinity` in the inequality expression `(< x ...)` must be specified using the operator `as`, to avoid ambiguities.

3 Semantics of FPA

Following the SMT-LIB conventions, the FPA semantics is defined in terms of mappings $[\![\cdot]\!]$ from sorts to sets, and from function symbols to set-theoretic functions.

Preliminaries. In the following, $\mathbb{R}$ denotes the set of real numbers, $\mathbb{Z}$ the set of integers, and $\mathbb{Q}$ the set of rational numbers. We introduce an extension of real arithmetic to the set $\mathbb{R}^+ = \mathbb{R} \cup \{+\infty, -\infty, \mathit{NaN}\}$, where $+\infty, -\infty, \mathit{NaN}$ are individuals different from any real number. The set $\mathbb{R}^+$, together with the operations defined next, forms an algebraic structure that captures mathematical features distinguishing FPA from real arithmetic: the presence of infinities and the non-number *NaN*, modelling partiality of operations. In this sense, $\mathbb{R}^+$ can be considered as an idealised (continuous) version of FPA. However, $\mathbb{R}^+$ does not incorporate features of FPA that are artifacts of the way numbers are represented. For instance, since *every* number of FPA has a *sign*, FPA distinguishes between $+0$ and -0. This distinction is not realised in $\mathbb{R}^+$.

The arithmetic operations $+, -, \cdot, /, |\cdot|, <, >, \leq, \geq$ naturally extend from $\mathbb{R}$ to $\mathbb{R}^+$, except that division by 0 is defined in some cases. Whenever any of the arguments of the predicates $<, >, \leq, \geq$ is *NaN*, the result is *false*. Whenever any of the arguments of $+, -, \cdot, /, |\cdot|$ is *NaN*, the result is *NaN* (the *NaN* is "propagated"). All other operations, i.e. division by 0 and those involving $+\infty, -\infty$, are defined by the axioms in Fig. 1.

$$+\infty \not< x, \qquad x < +\infty \iff x \neq +\infty, \qquad -\infty < x \iff x \neq -\infty, \qquad x \not< -\infty$$

$$x \leq y \iff (x < y \vee x = y), \qquad x > y \iff y < x, \qquad x \geq y \iff y \leq x$$

$$(+\infty) + x = x + (+\infty) = \begin{cases} \mathit{NaN} & x = -\infty \\ +\infty & \text{otherwise} \end{cases}, \qquad |x| = \begin{cases} x & x \geq 0 \\ -x & \text{otherwise} \end{cases}$$

$$(-\infty) + x = x + (-\infty) = \begin{cases} \mathit{NaN} & x = +\infty \\ -\infty & \text{otherwise} \end{cases},$$

$$-(+\infty) = -\infty, \qquad -(-\infty) = +\infty, \qquad x - y = x + (-y)$$

$$(+\infty) \cdot x = x \cdot (+\infty) = \begin{cases} +\infty & x > 0 \\ -\infty & x < 0 \\ \mathit{NaN} & \text{otherwise} \end{cases}, \qquad (-\infty) \cdot x = x \cdot (-\infty) = -(x \cdot (+\infty))$$

$$(+\infty)/x = \begin{cases} \mathit{NaN} & x = +\infty \text{ or } x = -\infty \\ +\infty & x \in \mathbb{R} \text{ and } x \geq 0 \\ -\infty & \text{otherwise} \end{cases}, \qquad x/(+\infty) = \begin{cases} \mathit{NaN} & x = +\infty \text{ or } x = -\infty \\ 0 & x \in \mathbb{R} \end{cases}$$

$$(-\infty)/x = -((+\infty)/x), \qquad x/(-\infty) = -(x/(+\infty)), \qquad x/0 = \begin{cases} +\infty & x > 0 \\ -\infty & x < 0 \\ \mathit{NaN} & \text{otherwise} \end{cases}$$

Figure 1: Axioms defining the operations in $\mathbb{R}^+$. In all formulae, we assume $x, y \in \mathbb{R}^+ \setminus \{\mathit{NaN}\}$.

3.1 The Domain of Floating-Point Numbers

`FP` numbers are represented by triplets (s,m,e) of integers: the *sign s*, the *significand m*, and the *exponent e*. The domain of FPA is organised as an indexed family $\mathbb{F}_{E^-,E^+,M}$ of sets, where the integers E^- and E^+ (with $E^- \leq E^+$) define the range of the exponent e, and the integer $M \geq 0$ defines the range of the significand m:

$$\begin{aligned}\mathbb{F}_{E^-,E^+,M} \;=\; & \left\{(s,m,e) \in \mathbb{Z}^3 \mid s \in \{0,1\},\ 2^{M-1} \leq m < 2^M,\ E^- \leq e \leq E^+\right\} \\ & \cup \left\{(s,m,E^-) \in \mathbb{Z}^3 \mid s \in \{0,1\},\ 0 \leq m < 2^{M-1}\right\} \\ & \cup \{+\infty, -\infty, NaN\}\end{aligned}$$

Each set $\mathbb{F}_{E^-,E^+,M}$ is defined as the union of three (disjoint) sets: (i) the set of *normal numbers*, whose significand is in the range $[2^{M-1}, 2^M)$, (ii) the set of *subnormal numbers*, whose exponent is E^- and whose significand is in the range $[0, 2^{M-1})$, and (iii) distinguished individuals representing infinities and non-numbers.

We also define an embedding of `FP` numbers into the extended reals $\mathbb{R}^+$:

$$re : \Big(\bigcup_{E^-,E^+,M} \mathbb{F}_{E^-,E^+,M} \Big) \to \mathbb{R}^+, \qquad \begin{aligned} re(s,m,e) &= (-1)^s \cdot m \cdot 2^e \\ re(x) &= x \qquad (x \in \{+\infty, -\infty, NaN\}) \end{aligned}$$

Note that *re* is injective on each set $\mathbb{F}_{E^-,E^+,M}$, with the only exception that $re(0,0,E^-) = re(1,0,E^-) = 0$. This is unavoidable, because zero does not have a sign in $\mathbb{R}^+$.

Domain interpretation. For any numerals $\langle ebits \rangle > 0$ and $\langle sbits \rangle > 0$, we define the interpretation of the sort `(_ FP` $\langle ebits \rangle$ $\langle sbits \rangle$`)` as follows:

$$[\![\texttt{(\_ FP } \langle ebits \rangle\ \langle sbits \rangle\texttt{)}]\!] = \mathbb{F}_{E^-,E^+,\langle sbits \rangle} ,$$

where $E^- = 3 - 2^{\langle ebits \rangle - 1} - \langle sbits \rangle$ and $E^+ = 2^{\langle ebits \rangle - 1} - \langle sbits \rangle$.

The sort `Rational` is interpreted as the set $[\![\texttt{Rational}]\!] = \mathbb{Q}$ of rational numbers.

3.2 Rounding to Floating-Point Numbers

If the result of an arithmetic operation cannot be represented as a `FP` number, it is necessary to perform rounding to derive a `FP` number close to the precise result. We generally define the semantics of FPA operations as follows: (i) use the embedding *re* to convert the floating-point argument(s) to elements of $\mathbb{R}^+$, (ii) apply the mathematical function on $\mathbb{R}^+$ corresponding to the FPA function to compute a precise result, (iii) use a rounding operation $round_{E^-,E^+,M}$ to round the $\mathbb{R}^+$ result to the floating-point domain w.r.t. the chosen rounding mode, (iv) check whether the result is zero-valued and thus corresponds to one of the two signed FPA zeros. If so, identify the correct sign of the result, depending on the individual semantics of each function. The last step is necessary since floating-point arithmetic, unlike $\mathbb{R}^+$ and $\mathbb{R}$, distinguishes between different zeros.

IEEE defines five modes of rounding, which are formalised in this section. We define rounding modes as the elements of the set *RM*:

$$[\![\texttt{RoundingMode}]\!] = RM = \left\{ \begin{array}{c} tiesToEven, tiesToAway, \\ towardPositive, towardNegative, towardZero \end{array} \right\}$$

$$[\![\texttt{roundNearestTiesToEven}]\!] = tiesToEven, \qquad [\![\texttt{roundTowardPositive}]\!] = towardPositive$$

$$[\![\texttt{roundNearestTiesToAway}]\!] = tiesToAway, \qquad [\![\texttt{roundTowardNegative}]\!] = towardNegative$$

$$[\![\texttt{roundTowardZero}]\!] = towardZero$$

The rounding operation is introduced as a family $round_{E^-,E^+,M} : RM \times \mathbb{R}^+ \to \mathbb{F}_{E^-,E^+,M}$ of mappings from the set of rounding modes and $\mathbb{R}^+$ to the various floating-point domains. The result of rounding *NaN* is always *NaN*:

$$round_{E^-,E^+,M}(d, NaN) = NaN$$

In the following, we therefore assume that $r \in \mathbb{R}^+ \setminus \{NaN\}$. In general, we define that $round_{E^-,E^+,M}(d,r)$ does not return the value $(1,0,E^-)$ (negative zero) for any number $r \in \mathbb{R}^+$; the other cases are handled in the following sections.

3.2.1 Rounding Ties to Even

Numbers that are too large or too small to be represented are rounded to $+\infty$ or $-\infty$:

$$\begin{aligned} round_{E^-,E^+,M}(tiesToEven, r) &= +\infty && \text{if } r \geq B \\ round_{E^-,E^+,M}(tiesToEven, r) &= -\infty && \text{if } r \leq -B \end{aligned}$$

where $B = \left(2^M - \frac{1}{2}\right) \cdot 2^{E^+}$. In case of $r \in (-B, B)$, the result $round_{E^-,E^+,M}(tiesToEven, r)$ of rounding r is a finite FP number (s,m,e) such that:

$$|r - re(s,m,e)| \leq |r - re(s',m',e')| \quad \text{for all } (s',m',e') \in \mathbb{F}_{E^-,E^+,M}$$

If two FP numbers are equi-distant to r, the one with even significand is chosen:

$$\text{for all } (s',m',e') \in \mathbb{F}_{E^-,E^+,M} : \text{ if } |r - re(s,m,e)| = |r - re(s',m',e')| \text{ and } m' \text{ is even, then } m \text{ is even.} \quad (1)$$

3.2.2 Rounding Ties to Away

The result of $round_{E^-,E^+,M}(tiesToAway, r)$ is defined exactly like $round_{E^-,E^+,M}(tiesToEven, r)$ in the previous paragraph, only that (1) is replaced with the condition:

$$\text{for all } (s',m',e') \in \mathbb{F}_{E^-,E^+,M} : \text{ if } |r - re(s,m,e)| = |r - re(s',m',e')| \text{ then } |re(s,m,e)| \geq |re(s',m',e')|.$$

3.2.3 Rounding Towards Positive

In *towardPositive* mode, a number r is rounded to the least upper floating-point bound. More precisely, we define

$$round_{E^-,E^+,M}(towardPositive, r) = f \in \mathbb{F}_{E^-,E^+,M} \setminus \{NaN\}$$

such that $r \leq re(f)$ and for all $f' \in \mathbb{F}_{E^-,E^+,M}$: if $r \leq re(f')$ then $re(f) \leq re(f')$.

3.2.4 Rounding Towards Negative

Similarly, in *towardNegative* mode, a number r is rounded downwards:

$$round_{E^-,E^+,M}(towardNegative, r) = f \in \mathbb{F}_{E^-,E^+,M} \setminus \{NaN\}$$

such that $r \geq re(f)$ and for all $f' \in \mathbb{F}_{E^-,E^+,M}$: if $r \geq re(f')$ then $re(f) \geq re(f')$.

3.2.5 Rounding Towards Zero

In the mode *towardZero*, numbers r are rounded to the closest FP number whose absolute value is at most r:

$$round_{E^-,E^+,M}(towardZero, r) = f \in \mathbb{F}_{E^-,E^+,M} \setminus \{NaN\}$$

such that $|r| \geq |re(f)|$ and for all $f' \in \mathbb{F}_{E^-,E^+,M}$: if $|r| \geq |re(f')|$ then $|re(f)| \geq |re(f')|$.

3.3 Semantics of Floating-Point Arithmetic Operations

The following pages define the semantics of the FPA operations, following the schema described in the beginning of Section 3.2. With the exception of the `min` and `max` symbols, the semantics of all function symbols is to return *NaN* (or *false* in case of Boolean-valued functions) whenever one of their arguments is *NaN*. Following IEEE 754-2008, the functions `min` and `max` are treated specially since a non-*NaN*, when compared against a *NaN*, is propagated.

Floating-point predicates such as the zero test `isZero` are defined directly as functions over a single floating-point argument, returning a Boolean.

As a convention, when we define the semantics of an operation using a case-expression

$$[\![\circ]\!](\dots) = \begin{cases} t_1 & \text{if } C_1 \\ t_2 & \text{if } C_2 \\ \dots & \\ t_n & \text{otherwise} \end{cases}$$

we stipulate that any C_i is implicitly strengthened by $\bigwedge_{j<i} \neg C_j$. In other words, cases are to be read and evaluated from top to bottom, choosing the first case whose condition is satisfied. Furthermore, when defining the semantics of Boolean-valued FPA functions in terms of (meta-level) formulae, we use the notation $[\phi]$ to express:

$$[\phi] \in \mathbb{B}, \qquad [\phi] = \begin{cases} \textit{true} & \text{if } \phi \\ \textit{false} & \text{otherwise} \end{cases}$$

3.3.1 Sign Operations

The operations for changing the sign of a FP number are defined by a simple case analysis:

$$[\![\mathsf{abs}]\!] : \mathbb{F}_{E^-,E^+,M} \to \mathbb{F}_{E^-,E^+,M}, \qquad \begin{aligned} [\![\mathsf{abs}]\!](s,m,e) &= (0,m,e) \\ [\![\mathsf{abs}]\!](+\infty) &= +\infty \\ [\![\mathsf{abs}]\!](-\infty) &= +\infty \end{aligned}$$

$$[\![-]\!] : \mathbb{F}_{E^-,E^+,M} \to \mathbb{F}_{E^-,E^+,M}, \qquad \begin{aligned} [\![-]\!](s,m,e) &= (1-s,m,e) \\ [\![-]\!](-\infty) &= +\infty \\ [\![-]\!](+\infty) &= -\infty \end{aligned}$$

3.3.2 Tests

Sign and zero tests do not depend on the rounding mode:

$$[\![\texttt{isZero}]\!], [\![\texttt{isSignMinus}]\!], [\![\texttt{isNZero}]\!], [\![\texttt{isPZero}]\!] : \mathbb{F}_{E^-,E^+,M} \to \mathbb{B}$$

$$\begin{aligned} [\![\texttt{isZero}]\!](s,m,e) &= [m=0] \\ [\![\texttt{isPZero}]\!](s,m,e) &= [m=0 \text{ and } s=0] \\ [\![\texttt{isZero}]\!](f) &= [\![\texttt{isPZero}]\!](f) = \textit{false} \qquad (\text{for } f \in \{\textit{NaN}, +\infty, -\infty\}) \\ [\![\texttt{isNZero}]\!](f) &= [\![\texttt{isPZero}]\!]([\![-]\!](f)) \end{aligned}$$

$$\begin{aligned} [\![\texttt{isSignMinus}]\!](s,m,e) &= [s=1] \\ [\![\texttt{isSignMinus}]\!](-\infty) &= \textit{true} \\ [\![\texttt{isSignMinus}]\!](f) &= \textit{false} \qquad (\text{for } f \in \{\textit{NaN}, +\infty\}) \end{aligned}$$

3.3.3 Comparisons

The binary comparison operators are defined like the corresponding operations on $\mathbb{R}^+$. Besides the standard equality predicate = already provided by the SMT-LIB core theory, we introduce a predicate == denoting equality as specified by IEEE; = and == will give different results when comparing *NaN* or the two floating-point zeroes.

$$[\![\texttt{==}]\!], [\![\texttt{<=}]\!], [\![\texttt{<}]\!], [\![\texttt{>=}]\!], [\![\texttt{>}]\!] : \mathbb{F}_{E^-,E^+,M} \times \mathbb{F}_{E^-,E^+,M} \to \mathbb{B}, \qquad [\![\circ_1]\!](f,f') = [re(f) \circ_2 re(f')]$$

where $(\circ_1, \circ_2) \in \{(\texttt{==}, =), (\texttt{<=}, \leq), (\texttt{<}, <), (\texttt{>=}, \geq), (\texttt{>}, >)\}$.

3.3.4 Field Operations

Arithmetic operations receive the rounding mode as first argument:

$$[\![\texttt{+}]\!], [\![\texttt{-}]\!], [\![\texttt{*}]\!], [\![\texttt{/}]\!] : RM \times \mathbb{F}_{E^-,E^+,M} \times \mathbb{F}_{E^-,E^+,M} \to \mathbb{F}_{E^-,E^+,M}$$

$$[\![\texttt{+}]\!](d,f,f') = \begin{cases} z & \text{if not } [\![\texttt{isZero}]\!](z) \\ (1,0,E^-) & \text{if } [\![\texttt{isNZero}]\!](f) \text{ and } [\![\texttt{isNZero}]\!](f') \\ (1,0,E^-) & \text{if } d = \texttt{roundTowardNegative} \\ (0,0,E^-) & \text{otherwise} \end{cases},$$

where $z = round_{E^-,E^+,M}(d, re(f) + re(f')) \in \mathbb{F}_{E^-,E^+,M}$.

$$[\![\texttt{-}]\!](d,f,f') = [\![\texttt{+}]\!](d,f,[\![-]\!](f')).$$

$$[\![\texttt{*}]\!](d,f,f') = \begin{cases} z & \text{if not } [\![\texttt{isZero}]\!](z) \\ (1,0,E^-) & \text{if } [\![\texttt{isSignMinus}]\!](f) \neq [\![\texttt{isSignMinus}]\!](f') \\ (0,0,E^-) & \text{otherwise} \end{cases},$$

where $z = round_{E^-,E^+,M}(d, re(f) \cdot re(f')) \in \mathbb{F}_{E^-,E^+,M}$.

$$[\![\texttt{/}]\!](d,f,f') = \begin{cases} -z & \text{if } [\![\texttt{isNZero}]\!](f') \text{ and not } [\![\texttt{isZero}]\!](f) \\ z & \text{if not } [\![\texttt{isZero}]\!](z) \\ (1,0,E^-) & \text{if } [\![\texttt{isSignMinus}]\!](f) \neq [\![\texttt{isSignMinus}]\!](f') \\ (0,0,E^-) & \text{otherwise} \end{cases},$$

where $z = round_{E^-,E^+,M}(d, re(f)/re(f')) \in \mathbb{F}_{E^-,E^+,M}$.

3.3.5 Fused Multiply-Add

The operation `fusedMA` receives a rounding mode and three `FP` numbers as argument. As for the other operations, the result of `fusedMA` is determined by precise calculation on $\mathbb{R}^+$ and rounding; the sign of the result is computed from the sign of the precise result and the sign of the corresponding computation $[\![\texttt{+}]\!](d, [\![\texttt{*}]\!](d, f, f'), f'')$ in terms of ordinary multiplication and addition:

$$[\![\texttt{fusedMA}]\!] : RM \times \mathbb{F}_{E^-,E^+,M} \times \mathbb{F}_{E^-,E^+,M} \times \mathbb{F}_{E^-,E^+,M} \to \mathbb{F}_{E^-,E^+,M}$$

$$[\![\texttt{fusedMA}]\!](d, f, f', f'') = \begin{cases} z & \text{if not } [\![\texttt{isZero}]\!](z) \\ (1, 0, E^-) & \text{if } x < 0 \\ (0, 0, E^-) & \text{if } x > 0 \\ (1, 0, E^-) & \text{if } [\![\texttt{isSignMinus}]\!](y) \\ (0, 0, E^-) & \text{otherwise} \end{cases},$$

where $x = (re(f) \cdot re(f')) + re(f'') \in \mathbb{R}^+$, $y = [\![\texttt{+}]\!](d, [\![\texttt{*}]\!](d, f, f'), f'')$, and $z = round_{E^-,E^+,M}(d, x) \in \mathbb{F}_{E^-,E^+,M}$.

3.3.6 Square Root

We define the square root operation implicitly via multiplication on reals. Note that $round_{E^-,E^+,M}(d, x)$ in the following equations is always a number with positive sign.

$$[\![\texttt{squareRoot}]\!] : RM \times \mathbb{F}_{E^-,E^+,M} \to \mathbb{F}_{E^-,E^+,M}$$

$$\begin{aligned} [\![\texttt{squareRoot}]\!](d, (0, m, e)) &= round_{E^-,E^+,M}(d, x) \\ [\![\texttt{squareRoot}]\!](d, (1, 0, E^-)) &= (1, 0, E^-) \\ [\![\texttt{squareRoot}]\!](d, (1, m, e)) &= NaN & \text{(for } m > 0) \\ [\![\texttt{squareRoot}]\!](d, +\infty) &= +\infty \\ [\![\texttt{squareRoot}]\!](d, f) &= NaN & \text{(for } f \in \{NaN, -\infty\}) \end{aligned}$$

where $x \in \mathbb{R}, x \geq 0$ such that $x \cdot x = re(0, m, e)$.

3.3.7 Rounding

The function `roundToIntegral` is defined in terms of the general rounding operation $round_{E^-,E^+,M}$ by choosing $E^- = 0$, ensuring an integral result:

$$[\![\texttt{roundToIntegral}]\!] : RM \times \mathbb{F}_{E^-,E^+,M} \to \mathbb{F}_{E^-,E^+,M}$$

$$[\![\texttt{roundToIntegral}]\!](d, f) = \begin{cases} z & \text{if not } [\![\texttt{isZero}]\!](z) \\ (1, 0, E^-) & \text{if } [\![\texttt{isSignMinus}]\!](f) \\ (0, 0, E^-) & \text{otherwise} \end{cases}$$

where

$$x = \begin{cases} round_{0,E^+,M}(d, re(f)) & \text{if } E^+ \geq 0 \\ round_{0,0,M+E^+}(d, re(f)) & \text{otherwise} \end{cases}$$

and $z \in \mathbb{F}_{E^-,E^+,M}$ such that $re(z) = re(x)$.

3.3.8 Remainder

The `remainder` operation is defined in terms of division and rounding to integers. When rounding the quotient $re(f)/re(f')$, the operation $round_{0,0,R}$ with $R = M + E^+ - E^-$ is used so that no overflows occur and the result is guaranteed to be finite:

$$[\![\texttt{remainder}]\!] : \mathbb{F}_{E^-,E^+,M} \times \mathbb{F}_{E^-,E^+,M} \to \mathbb{F}_{E^-,E^+,M}$$

$$[\![\texttt{remainder}]\!](f, f') \;=\; \begin{cases} f & \text{if } f' \in \{+\infty, -\infty\} \\ NaN & \text{if } f \in \{+\infty, -\infty\} \text{ or } f' = 0 \\ z & \text{if not } [\![\texttt{isZero}]\!](z) \\ (1, 0, E^-) & \text{if } [\![\texttt{isSignMinus}]\!](f) \\ (0, 0, E^-) & \text{otherwise} \end{cases}$$

where $n = round_{0,0,R}(tiesToEven, re(f)/re(f'))$ and $z \in \mathbb{F}_{E^-,E^+,M}$ such that $re(z) = re(f) - re(f') \cdot re(n)$.

3.3.9 Minimum and Maximum

The following two operations are the only ones where a *NaN* is not generally propagated from arguments to results:

$$[\![\texttt{max}]\!], [\![\texttt{min}]\!] : \mathbb{F}_{E^-,E^+,M} \times \mathbb{F}_{E^-,E^+,M} \to \mathbb{F}_{E^-,E^+,M}$$

$$\begin{aligned} [\![\texttt{max}]\!](f, f') \;&=\; \begin{cases} f' & \text{if } f = NaN \\ f & \text{if } f' = NaN \\ f & \text{if } re(f) > re(f') \text{ or } \big(re(f) = re(f') \text{ and } [\![\texttt{isSignMinus}]\!](f')\big) \\ f' & \text{otherwise} \end{cases} \\ [\![\texttt{min}]\!](f, f') \;&=\; [\![\texttt{-}]\!]([\![\texttt{max}]\!]([\![\texttt{-}]\!](f), [\![\texttt{-}]\!](f'))) \end{aligned}$$

3.3.10 Decimal Literals

The denotation $[\![\texttt{n}]\!]$ of a number `n` in decimal notation is the corresponding rational $r \in \mathbb{Q}$. Converting a rational $r \in \mathbb{Q} \subset \mathbb{R}^+$ to a floating-point number reduces to rounding:

$$[\![\texttt{(\_ asFloat } \langle ebits \rangle\ \langle sbits \rangle\texttt{)}]\!] : RM \times \mathbb{Q} \to \mathbb{F}_{E^-,E^+,\langle sbits \rangle}$$

$$[\![\texttt{(\_ asFloat } \langle ebits \rangle\ \langle sbits \rangle\texttt{)}]\!](d, r) \;=\; round_{E^-,E^+,\langle sbits \rangle}(d, r)$$

where $E^- = 1 - 2^{\langle ebits \rangle - 1} - \langle sbits \rangle$ and $E^+ = 2^{\langle ebits \rangle - 1} - \langle sbits \rangle$ as in Sect. 3.1.

4 Conclusion

The lack of reliable and scalable floating-point arithmetic decision procedures is a serious handicap for the automatic verification of embedded software. In order to encourage and support the development of such procedures, we have proposed an SMT-LIB format for encoding floating-point numbers and operations on them. The encoding largely emulates the stipulations of the IEEE floating-point standard 754 and is thus very close to what is implemented on most computers today. We have mentioned and motivated deviations from this standard where applicable.

There are certain operators that have not been added to the FPA theory yet, such as transcendental operations and casts between the various `FP` and other datatypes. In addition to filling these gaps, immediate future work includes the generation and assembly of a benchmark suite for floating-point arithmetic decision problems, such as extracted from pertinent embedded software. The long-term goal is a FPA testbench that helps improve verification tools for programs with floating-point arithmetic, an essential endeavour given the wide-spread use of floating-point calculations in safety-critical embedded systems.

References

[1] Monniaux, D.: The pitfalls of verifying floating-point computations. ACM Trans. Program. Lang. Syst. **30** (2008)

[2] Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB standard version 2.0 (2010) `http://www.smt-lib.org`.

[3] IEEE Standards Association: IEEE Standard for Floating-Point Arithmetic, `http://grouper.ieee.org/groups/754/`. (2008)

[4] Carreño, V.A.: Interpretation of IEEE-854 floating-point standard and definition in the HOL system. Technical report, NASA Langley Research Center (1995)

[5] Goubault, E.: Static analyses of the precision of floating-point operations. In: Static Analysis Symposium (SAS). (2001) 234–259

[6] Ayad, A., Marché, C.: Multi-prover verification of floating-point programs. In: Proceedings, International Joint Conference on Automated Reasoning, Edinburgh, Scotland. (2010) to appear.

[7] Brillout, A., Kroening, D., Wahl, T.: Mixed abstractions for floating-point arithmetic. In: Formal Methods in Computer-Aided Design (FMCAD). (2009)

[8] Harrison, J.: Floating-point verification using theorem proving. In: SFM. (2006) 211–242

[9] Harrison, J.: Floating-point verification. J. UCS **13** (2007) 629–638

A Complete Syntax of FPA

```
;; Rounding modes

 :sorts ((RoundingMode 0))

 :funs ((roundNearestTiesToEven RoundingMode)
        (roundNearestTiesToAway RoundingMode)
        (roundTowardPositive    RoundingMode)
        (roundTowardNegative    RoundingMode)
        (roundTowardZero        RoundingMode)
        )


;; The actual floating-point arithmetic

 :sorts_description
 "All nullary sort symbols of the form (_ FP ebits sbits),
  where ebits and sbits are numerals > 0."

;; ebits defines the number of bits in the exponent,
;; sbits the number of bits in the significand

 :funs_description
 "All function symbols with declaration of the form
        (plusInfinity  (_ FP ebits sbits))
        (minusInfinity (_ FP ebits sbits))
  where ebits and sbits are numerals > 0."

;; Given ebits and sbits, there are exactly one FP representing +infty
;; and exactly one FP representing -infty, in agreement with IEEE.

 :funs_description
 "All function symbols with declaration of the form
        (NaN (_ FP ebits sbits))
  where ebits and sbits are numerals > 0."

;; Given ebits and sbits, there is exactly one NaN (not-a-number).
;; This deviates from (simplifies) the IEEE.

 :funs_description
 "All function symbols with declarations of the form
      ; absolute value
        (abs (_ FP ebits sbits) (_ FP ebits sbits)

      ; unary minus (no rounding)
        (- (_ FP ebits sbits) (_ FP ebits sbits))

        (+ RoundingMode (_ FP ebits sbits) (_ FP ebits sbits) (_ FP ebits sbits))
      ; binary minus
        (- RoundingMode (_ FP ebits sbits) (_ FP ebits sbits) (_ FP ebits sbits))
        (* RoundingMode (_ FP ebits sbits) (_ FP ebits sbits) (_ FP ebits sbits))
        (/ RoundingMode (_ FP ebits sbits) (_ FP ebits sbits) (_ FP ebits sbits))
      ; fused multiply-add: (x * y) + z
        (fusedMA RoundingMode (_ FP ebits sbits) (_ FP ebits sbits)
                              (_ FP ebits sbits) (_ FP ebits sbits))

      ; square root
        (squareRoot RoundingMode (_ FP ebits sbits) (_ FP ebits sbits))

      ; round to nearest integral FP
        (roundToIntegral RoundingMode (_ FP ebits sbits) (_ FP ebits sbits))

      ; x - y * n, where n in Z is nearest to x/y
        (remainder (_ FP ebits sbits) (_ FP ebits sbits) (_ FP ebits sbits))

        (==  (_ FP ebits sbits) (_ FP ebits sbits) Bool :chainable)
        (<=  (_ FP ebits sbits) (_ FP ebits sbits) Bool :chainable)
        (<   (_ FP ebits sbits) (_ FP ebits sbits) Bool :chainable)
        (>=  (_ FP ebits sbits) (_ FP ebits sbits) Bool :chainable)
        (>   (_ FP ebits sbits) (_ FP ebits sbits) Bool :chainable)

        (isZero      (_ FP ebits sbits) Bool) ; test for +0 or -0
        (isSignMinus (_ FP ebits sbits) Bool) ; test for <0
        (isNZero     (_ FP ebits sbits) Bool) ; test for -0
        (isPZero     (_ FP ebits sbits) Bool) ; test for +0

      ; minimum
        (min (_ FP ebits sbits) (_ FP ebits sbits) (_ FP ebits sbits))
      ; maximum
        (max (_ FP ebits sbits) (_ FP ebits sbits) (_ FP ebits sbits))

  where ebits and sbits are numerals > 0."


;; Literals, using the syntactic category DECIMAL

 :sorts ((Rational 0))
 :funs ((DECIMAL Rational))

 :funs_description
 "All function symbols with declaration of the form
         ((_ asFloat ebits sbits) RoundingMode Rational (_ FP ebits sbits))
  where ebits and sbits are numerals > 0."
```