An Automatable Formal Semantics for IEEE-754 Floating-Point Arithmetic

Last revised: Friday 30th October, 2015

Martin Brain*, Cesare Tinelli†, Philipp Rümmer‡ and Thomas Wahl§ * Department of Computer Science, University of Oxford, Oxford, UK † Department of Computer Science, The University of Iowa, Iowa City, IA, USA ‡ Department of Information Technology, Uppsala University, Uppsala, Sweden § College of Computer and Information Science, Northeastern University, Boston, MA, USA

Abstract

Automated reasoning tools often provide little or no support to reason accurately and efficiently about floating-point arithmetic. As a consequence, software verification systems that use these tools are unable to reason reliably about programs containing floating-point calculations or may give unsound results. These deficiencies are in stark contrast to the increasing awareness that the improper use of floating-point arithmetic in programs can lead to unintuitive and harmful defects in software. To promote coordinated efforts towards building efficient and accurate floating-point reasoning engines, this paper presents a formalization of the IEEE-754 standard for floating-point arithmetic as a theory in many-sorted first-order logic. Benefits include a standardized syntax and unambiguous semantics, allowing tool interoperability and sharing of benchmarks, and providing a basis for automated, formal analysis of programs that process floating-point data.

I. Introduction

Real values can be represented in a computer in many ways, with various level of precision: as fixed-point numbers, binary or decimal floating-point numbers, rationals, arbitrary precision reals, etc. Due to the wide availability of high-performance hardware and support in most programming languages, binary floating-point has become the dominant representation system. Aside from some industry-specific exceptions, most programs that interact with or model the real world, not least those employed in safety critical applications, rely on binary floating-point arithmetic. This creates a significant challenge for program analysis tools: accurate reasoning about the behavior of (numerical) programs is only possible with bit-accurate reasoning about floating-point arithmetic. Many automated verification tools, such as software model checkers, rely on solvers for Satisfiability Modulo Theories (SMT) [4] as their reasoning engines. These solvers use specialized, built-in methods to check the satisfiability of formulas in *background theories* of interest, such as for instance the theories of integer numbers, arrays, bit vectors and so on. Reasoning about floating-point numbers accurately in SMT then requires the identification of a suitable theory of floating-point arithmetic. Due to significant semantic differences, reasoning in substitute theories such as the reals would generally render the solvers unsound and is thus not a satisfactory option.

In the past, designing a formal theory of floating-point arithmetic would have been prohibitively complex, as different manufacturers used different floating-point formats which varied from the others in significant, structural aspects. The introduction, in 1985, and subsequent near universal adoption of the IEEE-754 standard has considerably improved the situation. However, the standard is unsatisfactory as a formal theory definition for a number of reasons: it is written in natural language; it covers various aspects that are not relevant to developing a theory of floating-point (see Section VIII-B); it is lengthy (the 2008

Work partially supported by the Toyota Motor Corporation, ERC project 280053, EPSRC project EP/H017585/1, DSTL under CDE Project 30713, NSF grant CCF-1218075, and the VR grant 2011-6310.

revision has 70 pages) and, most critically for automated reasoning purposes, it does not describe a logical signature and interpretations.

This paper presents the syntax and semantics of a logical theory that formalizes floating-point arithmetic based on the IEEE-754 standard, with the goal of facilitating the construction of automated and precise reasoning tools for it. The models of our theory correspond exactly to conforming implementations of IEEE-754. While rather general, our formalization was developed with inputs from the SMT community to be a reference theory for SMT solver implementors and users, and was recently incorporated in the SMT-LIB standard [5], a widely used input/output language for SMT solvers.

This paper makes three specific contributions:

1) Discusses the challenges of reasoning about floating-point numbers and illustrates its critical uses. [Section II]
2) Presents mathematical structures intended to formally model binary floating-point arithmetic. [Section V]
3) Provides a signature for a theory of floating-point arithmetic and an interpretation of its operators in terms of the mathematical structures defined earlier. [Section VI]

II. Floating-Point Arithmetic

Floating-point refers to a way of encoding subsets of the rational numbers using bit vectors of fixed width. A floating point number consists of three such bit vectors: one for the fractional part of the number, called the *significand*, one for the *exponent* of an integer power by which the fractional part is multiplied, and a single bit for the *sign*. For example:

$$\begin{aligned} 7.28125 &= 7 + 28125 \cdot 10^{-5} = (2^2 + 2^1 + 2^0) + (2^{-2} + 2^{-5}) \\ &= 111.01001_2 = 1.1101001_2 \cdot 2^2 \end{aligned}$$

can be represented as a *binary* floating point number with a sign bit of `0`, an exponent of `2` and a significand of `1101001`: the leading `1` before the binary point is omitted (for so-called *normal* numbers) and hence known as the *hidden bit*.

Arithmetic on floating point numbers is defined as performing the operation *as if the numbers were reals*, followed by rounding the result to the nearest value representable as a floating point number; "nearest" is defined by a set of rules called a *rounding mode*. Many floating-point systems implemented in computer processors have included special values such as infinities and not-a-number (NaN). When an underflow, overflow or other exceptional condition occurs, these special values can be returned instead of triggering an interrupt. This can simplify the control circuitry and results in faster computation but at the cost of making the floating-point number system more complex. IEEE-754 standardizes, in different floating-point formats, the sizes of the bit vectors used for number presentation, as well as the various rounding modes, and the meaning and use of special values.

Formal reasoning about algorithms and programs that use floating-point numbers has acquired a reputation of being difficult, complex and error prone [25]. There are at least two causes for this. One is the combinatorial explosion in operations due to the existence of special computational values. More precisely, IEEE-754 defines five classes of numbers: normal, subnormal (which cannot be expressed in normal form, due to limits on the exponent range), zeros (a positive and a negative one, so that approximation from above and below can be distinguished), infinities (one positive and one negative), and NaN. The behavior of each operation depends on the classes of its arguments, potentially requiring a five-way case distinction for every operation considered. The other cause for the intricacies is rounding. Rounding is vital as it permits a fixed bound on the amount of memory required to store and compute with floating-point numbers. However, it also causes floating-point arithmetic to defy some fundamental laws of traditional arithmetics, such as the associativity of addition[1] As a result, many algebraic and automated reasoning

[1]Since rounding is performed after every operation, the expressions $(a + b) + c$ and $a + (b + c)$ can evaluate to different values.

techniques based on these laws are inapplicable to it. While challenging, reasoning formally and possibly automatically about floating-point numbers is crucial for many purposes.

Identifying the Generation of Special-Values It is generally useful to know whether a program may generate the special floating-point values infinity and NaN. While their presence is not necessarily a bug, they fall outside classical real-number arithmetic. In some cases, the detection of subnormal numbers is valuable as they have limited precision and can thus amplify errors.

Detecting Numerical Instabilities As a consequence of rounding, there are non-zero, normal numbers x and y such that $x + y = x$. Although again this is not necessarily a bug, it is counter-intuitive for many programmers and may be a symptom of deeper algorithmic problems. Similarly, *catastrophic cancellation* occurs in $x+y$ when the exponents of x and y are (near-)equal and their signs differ. In this case the most significant bits can cancel, effectively amplifying the errors introduced by rounding.

Exposing Undefined Behavior With few exceptions, IEEE-754 fully defines the results of all operations on floating-point numbers but does not cover corner cases of conversions to other formats. Although this may seem like a trivial issue, it was the immediate cause of the loss of Ariane 5 Flight 501.

Generating Test-Cases Many safety standards for embedded software mandate testing coverage criteria. Automatic test case generation is vital as system complexity rises, and is particularly important for showing which paths are infeasible, as developers often miss floating-point related corner cases. These applications demand model generating and bit-precise reasoning engines.

Proving Functional Correctness Formal correctness specifications for numerical programs often include bounds on the output. Checking those bounds is a suitable task for a reasoning tool. Another common correctness measure is the difference between the result of the computation performed in floating-point and its real-arithmetic result. Care must be taken because most non-trivial numerical algorithms are not simple transcriptions of mathematical expressions into software; comparing the same algorithm over the two domains may be misleading.

All these scenarios have in common that classical mathematical reasoning, regardless of whether it is manual or automated, is ineffective when dealing with floating-point.

III. Formalization in Automated Reasoning

In general, to automate reasoning in a domain $\mathbf{D}$ of interest using a formal logic one has to restrict the set of possible interpretations assigned to the function and predicate symbols used to build formulas representing statements over $\mathbf{D}$. There are normally two approaches to achieve this, one *axiomatic* and one *algebraic*.

In the axiomatic approach, typical of interactive theorem proving tools, one first constructs a formula ψ that *axiomatizes* $\mathbf{D}$, i.e., formally describes (some of) the properties of the chosen function and predicate symbols. Then, to check that a particular formula ϕ is valid in $\mathbf{D}$ one asks the prover, in essence, to check the logical validity of the implication $\psi \Rightarrow \phi$. Previous formalizations of floating-point (see Section VII) have followed this approach. Since the axiomatization ψ is part of the *input*, no specific support for the domain $\mathbf{D}$ on the prover's side is needed. Formalizations of this kind are flexible and easily extensible, however, the axioms ψ tend to be voluminous and intricate, which can limit the performance of many automated techniques. Another limitation is that certain domains cannot be captured accurately by a relatively small, or even finite, axiomatization in the prover's logic. In that case, different, approximate axiomatizations may have to be considered and compared with respect the trade-offs they offer.

In the algebraic approach, typical of SMT solvers, a domain $\mathbf{D}$ is formalized instead by a set of algebraic structures (i.e., models in the chosen logic) that interpret the various functions and predicate symbols. The formalization is used as a *specification* for the prover, and the knowledge of what the symbols mean is pre-built into the prover. An advantage of this is that fast, domain specific procedures can be used to reason about $\mathbf{D}$. Moreover, in addition to checking for validity in $\mathbf{D}$, such procedure are usually also able to generate counter-examples for invalid formulas. Since these formalizations are used as specifications, the key issues are whether they can be implemented easily and efficiently and how well the interpretations they describe capture relevant properties of the domain.

TABLE I
FORMALIZATION OF NaN'S BEHAVIOR, WITH $u \in S^*$

$$\begin{array}{rclcl} u + \mathrm{NaN} & = & \mathrm{NaN} + u & = & \mathrm{NaN} \\ u \cdot \mathrm{NaN} & = & \mathrm{NaN} \cdot u & = & \mathrm{NaN} \end{array} \qquad \begin{array}{rcl} -\mathrm{NaN} & = & \mathrm{NaN} \\ \mathrm{NaN}^{-1} & = & \mathrm{NaN} \end{array}$$

$$\mathrm{NaN} \leqslant u \;\Leftrightarrow\; u = \mathrm{NaN} \qquad u \leqslant \mathrm{NaN} \;\Leftrightarrow\; u = \mathrm{NaN}$$

We present a formalization of floating-point arithmetic in the algebraic style, intended as a specification for SMT solvers, and make the case that this accurately captures the semantics of the IEEE-754 standard. We concentrate on arithmetic aspects, abstracting away more operational ones, such as exception handling. Also, we only consider the case of binary (as opposed to decimal) floating-point arithmetic, as it is more widely used in practice.

IV. Formal Foundations

IEEE-754 gives an informal definition of the semantics of floating-point operations:

> 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.

To formalize this it is helpful to define an extension of the real numbers so that we can treat floating-point values as if they had "infinite precision and unbounded range," and to define a notion of rounding. Note that the extended reals are used here simply to aid the formal definition of floating-point operations. They are not the domain of interpretation of floating-point numbers. For that we will define a family of algebras over bit vector triples.

A. Extended Reals

We extend the set $\mathbb{R}$ of real numbers with three new elements: $+\infty$, $-\infty$ and NaN, which represent respectively positive and negative infinity and a special *not a number* value used to obtain an arithmetically closed system. For each set $S \subseteq \mathbb{R}$ we can define the following sets:

$$S^\dagger \;=\; S \cup \{+\infty, -\infty\} \qquad S^* \;=\; S^\dagger \cup \{\mathrm{NaN}\}$$

When S is the base of an ordered additive or multiplicative monoid or group, we extend $\mathbb{R}$'s binary operations $+$ and $\cdot$, unary operations $-$ and $(\_)^{-1}$ and order relation $\leq$. Table I gives axioms defining these operations when one argument is NaN. Table III gives axioms defining these operations when one argument is $+\infty$ or $-\infty$, as well as the axiom for the inverse of zero. We remark that, although these definitions extend the operations of ordered fields and rings and can be applied to $\mathbb{R}$ and $\mathbb{Z}$, the extended reals $\mathbb{R}^*$ and extended integers $\mathbb{Z}^*$ do not have all of the structure of $\mathbb{R}$ and $\mathbb{Z}$.[2]

Note that $(\mathbb{R}^*, \leqslant)$ is a partial order since NaN is comparable only with itself. In contrast $(\mathbb{R}^\dagger, \leqslant)$ is a total order. The extended reals operate as three largely algebraically independent subsets: $\{\mathrm{NaN}\}$, $\{+\infty, -\infty\}$ and $\mathbb{R}$. If a sub-expression of an expression e evaluates to NaN, then the whole e evaluates to NaN — the set is closed under the basic operations. Infinities generate infinities or NaN, although the reciprocal operator maps an infinity back to a real value. Reals are of course closed under all operations except reciprocal of zero. For convenience, we will also use the usual additional symbols in Table II, which are definable in terms of the basic operations.

[2] $\mathbb{R}^*$ is neither an additive or multiplicative group as $(-\mathrm{NaN}) + \mathrm{NaN} \neq 0$ and $\mathrm{NaN}^{-1} \cdot \mathrm{NaN} \neq 1$, nor does it have an annihilating 0 as $\mathrm{NaN} \cdot 0 \neq 0$. However it is an additive and multiplicative commutative monoid with the distributivity property, a structure some authors refer to as a semi-ring.

TABLE II
DEFINED SYMBOLS, WITH $x, y \in S^*$

$$\begin{array}{rcl} x - y & := & x + (-y) \\ x/y & := & x \cdot y^{-1} \end{array} \qquad \begin{array}{rcl} x \geqslant y & := & y \leqslant x \\ x < y & := & (x \leqslant y) \wedge \neg(x = y) \\ x > y & := & (x \geqslant y) \wedge \neg(x = y) \end{array}$$

TABLE III
FORMALIZATION OF $\pm\infty$'S AND INVERSE OF ZERO'S BEHAVIOR, $w \in S^\dagger$

$$+\infty \leqslant w \;\Leftrightarrow\; w = +\infty \qquad w \leqslant -\infty \;\Leftrightarrow\; w = -\infty$$

$$\begin{array}{llll} w \leqslant +\infty & -(+\infty) = -\infty & +\infty^{-1} = 0 \\ -\infty \leqslant w & -(-\infty) = +\infty & -\infty^{-1} = 0 \end{array}$$

$$w + (+\infty) = (+\infty) + w = \begin{cases} \mathrm{NaN} & \text{if } w = -\infty \\ +\infty & \text{if } w \neq -\infty \end{cases}$$

$$w + (-\infty) = (-\infty) + w = \begin{cases} \mathrm{NaN} & \text{if } w = +\infty \\ -\infty & \text{if } w \neq -\infty \end{cases}$$

$$w \cdot +\infty = +\infty \cdot w = \begin{cases} +\infty & \text{if } 0 < w \\ -\infty & \text{if } w < 0 \\ \mathrm{NaN} & \text{if } w = 0 \end{cases}$$

$$w \cdot -\infty = -\infty \cdot w = \begin{cases} -\infty & \text{if } 0 < w \\ +\infty & \text{if } w < 0 \\ \mathrm{NaN} & \text{if } w = 0 \end{cases}$$

$$0^{-1} = +\infty$$

B. Rounding

The second concept needed to formalize the IEEE-754 definition of operations is that of rounding; a map that will take the intermediate result in $\mathbb{R}^*$ back into the set of floating-point numbers. We define it as a function that selects between the two adjoints of the corresponding map into $\mathbb{R}^*$.

More generally, let $(X, \sqsubseteq)$ be a partially ordered set that consists of one or more disjoint lattices, and let $v : X \to \mathbb{R}^*$ be an order-embedding function from X into the extended reals such that $\{+\infty, -\infty, \mathrm{NaN}\} \subset v(X)$. Then, the *upper adjoint* and *lower adjoint* of v are respectively the unique functions $\overline{v} : \mathbb{R}^* \to X$ and $\underline{v} : \mathbb{R}^* \to X$ such that for all $r \in \mathbb{R}^*$.

- $r \leqslant v(\overline{v}(r))$ and $\overline{v}(r) \sqsubseteq x$ for all $x \in X$ with $r \leqslant v(x)$;
- $v(\underline{v}(r)) \leqslant r$ and $x \sqsubseteq \underline{v}(r)$ for all $x \in X$ with $v(x) \leqslant r$.

The function $\overline{v}$ maps an element r to the smallest element of X that projects above r (rounding up) while $\underline{v}$ maps r to the largest element of X that projects below r (rounding down). Let $\mathbb{B} = \{\top, \bot\}$ be the Booleans, with $\top$ being the true value. We define a family of (higher-order) rounding functions:

$$\mathrm{round} : \mathrm{RM} \times \mathbb{B} \times \mathbb{R}^* \to (X \to \mathbb{R}^*) \to (\mathbb{R}^* \to X)$$

parametrized by the partially ordered set X, which provides a systematic way of selecting between rounding up and rounding down. Given a map $v : X \to \mathbb{R}^*$, the rounding function returns one of v's two adjoints, based on three previous inputs. The first is the *rounding mode*, chosen from the set:

$$\mathrm{RM} = \{\mathrm{rne}, \mathrm{rna}, \mathrm{rtp}, \mathrm{rtn}, \mathrm{rtz}\}$$

which represents the five rounding modes defined by IEEE-754, namely, round to nearest with ties picking even value (rne), round to nearest with ties away from zero (rna), round towards $+\infty$ (rtp), round towards $-\infty$ (rtn), and round towards zero (rtz). The second input of round is a Boolean value determining the sign of zero when X contains signed zeros (which is the case when X is a set of floating-point numbers). The third input is the value to be rounded, needed because the rounding direction may depend on it (for example, when rounding to the nearest element of X).

TABLE IV
DEFINITION OF round

$$\mathrm{round}(\mathrm{rne},s,r)(v) = \begin{cases} \overline{v} & r \neq 0,\ \neg \mathrm{lh}_X(r,v),\ \neg \mathrm{tb}_X(r,v) \\ \overline{v} & r \neq 0,\ \mathrm{tb}_X(r,v),\ \mathrm{ev}_X(\overline{v}(r)) \\ \underline{v} & r \neq 0,\ \mathrm{tb}_X(r,v),\ \mathrm{ev}_X(\underline{v}(r)) \\ \underline{v} & r \neq 0,\ \mathrm{lh}_X(r,v) \\ \mathrm{rsz}(s)(v) & r = 0 \\ \underline{v} & r = \mathrm{NaN} \end{cases}$$

$$\mathrm{round}(\mathrm{rna},s,r)(v) = \begin{cases} \overline{v} & r > 0,\ \neg \mathrm{lh}_X(r,v) \\ \underline{v} & r > 0,\ \mathrm{lh}_X(r,v) \\ \mathrm{rsz}(s)(v) & r = 0 \\ \overline{v} & r < 0,\ \neg \mathrm{lh}_X(r,v),\ \neg \mathrm{tb}_X(r,v) \\ \underline{v} & r < 0,\ \mathrm{lh}_X(r,v) \vee \mathrm{tb}_X(r,v) \\ \underline{v} & r = \mathrm{NaN} \end{cases} \quad \text{where}$$

$$\mathrm{round}(\mathrm{rtp},s,r)(v) = \begin{cases} \mathrm{rsz}(s)(v) & r = 0 \\ \overline{v} & \text{otherwise} \end{cases}$$

$$\mathrm{round}(\mathrm{rtn},s,r)(v) = \begin{cases} \mathrm{rsz}(s)(v) & r = 0 \\ \underline{v} & \text{otherwise} \end{cases}$$

$$\mathrm{round}(\mathrm{rtz},s,r)(v) = \begin{cases} \underline{v} & r > 0 \\ \mathrm{rsz}(s)(v) & r = 0 \\ \overline{v} & \text{otherwise} \end{cases}$$

$$\mathrm{rsz}(\top)(v) = \overline{v} \qquad \mathrm{rsz}(\bot)(v) = \underline{v}$$

TABLE V
AUXILIARY PREDICATES FOR ROUND IN THE CASE $X = \mathbb{Z}^*$

$$\begin{array}{rcl} \mathrm{lh}_{\mathbb{Z}^*}(r,v) & := & r - v(\underline{v}(r)) < v(\overline{v}(r)) - r \\ \mathrm{tb}_{\mathbb{Z}^*}(r,v) & := & r - v(\underline{v}(r)) = v(\overline{v}(r)) - r \\ \mathrm{ev}_{\mathbb{Z}^*}(x) & := & \exists z \in \mathbb{Z} \centerdot x = 2 * z \end{array}$$

The function round is defined in Table IV. The definition relies on three auxiliary predicates lh_X, tb_X and ev_X whose own definition depend on the particular domain X. These express: when the value is in the *lower half* of the interval between two representations in X (i.e. closer to $\underline{v}(r)$ than $\overline{v}(r)$); the *tie-break* condition when it is equal distance from both; and whether a representation is even. For illustration purposes, we provide a definition of those predicates in Table V for when $X = \mathbb{Z}$, the set of integers with the usual ordering. A definition of those predicates for sets of floating-point numbers is given later, after we formalize such sets.

The fairly elaborate definition of round is motivated by our goal to provide an accurate model of rounding as defined in IEEE-754. In particular, there is no mathematical reason for not using exclusively $\overline{v}$ or $\underline{v}$ in it. However, doing so would fail to reflect some properties of IEEE-754 floating-point numbers, for example "the sign of a sum [...] differs from at most one of the addends' signs" [1]. For brevity, we will write $\mathrm{rnd}(v,m,s,r)$ for the application $\mathrm{round}(m,s,r)(v)(r)$ which returns the value of X that the real number r is rounded to by using v.

V. MODELS OF FLOATING-POINT ARITHMETIC

In this section we specify a set of (many-sorted) structures in the sense of model theory. These are the intended models of a logical theory of floating-point numbers that reflects IEEE-754. In the next section we will specify a signature for such a theory and show how each sort, function and predicate symbol in the signature is interpreted over this set of structures.

A. Universe

The universe of each of our models consists of multiple sets: one for the rounding modes and one for each of the different floating-point precisions. Floating-point numbers other than NaN are triples of bit vectors modelling the three components (sign, exponent and significand) of the representations in IEEE-754. We identify bit vectors of length $\nu > 0$ with elements of the function space $\mathbb{BV}_\nu = \mathbb{N}_\nu \to \{0,1\}$ where $\mathbb{N}_\nu = \{0,\ldots,\nu-1\}$. We write $\mathbf{1}_\nu$ for the unique function in $\mathbb{N}_\nu \to \{1\}$ and $\mathbf{0}_\nu$ for the unique function in $\mathbb{N}_\nu \to \{0\}$, which represent respectively the bit vector of length ν containing all ones and that containing all zeros. Let $\mathrm{ub}_\nu : \mathbb{BV}_\nu \to \mathbb{N}$ and $\mathrm{sb}_\nu : \mathbb{BV}_\nu \to \mathbb{Z}$ denote the usual unsigned and 2's complement encodings of bit vectors into integers. Let $B_{\mu,\nu}$ denote the set $\mathbb{BV}_1 \times \mathbb{BV}_\mu \times \mathbb{BV}_{\nu-1}$. For all integers $\varepsilon, \sigma > 1$, we define the set of floating-point numbers with ε exponent bits and σ significand bits[3] as the set $\mathbb{F}_{\varepsilon,\sigma} = \mathrm{F}_{\varepsilon,\sigma} \cup \{\mathrm{NaN}\}$ where

$$\begin{aligned}
\mathrm{F}_{\varepsilon,\sigma} &= \mathrm{FZ}_{\varepsilon,\sigma} \cup \mathrm{FS}_{\varepsilon,\sigma} \cup \mathrm{FN}_{\varepsilon,\sigma} \cup \mathrm{FI}_{\varepsilon,\sigma} \\
\mathrm{FZ}_{\varepsilon,\sigma} &= \{(s,e,m) \in B_{\varepsilon,\sigma} \mid e = \mathbf{0}_\varepsilon,\ m = \mathbf{0}_{\sigma-1}\} \\
\mathrm{FS}_{\varepsilon,\sigma} &= \{(s,e,m) \in B_{\varepsilon,\sigma} \mid e = \mathbf{0}_\varepsilon,\ m \neq \mathbf{0}_{\sigma-1}\} \\
\mathrm{FN}_{\varepsilon,\sigma} &= \{(s,e,m) \in B_{\varepsilon,\sigma} \mid e \neq \mathbf{1}_\varepsilon,\ e \neq \mathbf{0}_\varepsilon\} \\
\mathrm{FI}_{\varepsilon,\sigma} &= \{(s,e,m) \in B_{\varepsilon,\sigma} \mid e = \mathbf{1}_\varepsilon,\ m = \mathbf{0}_{\sigma-1}\}
\end{aligned}$$

The last four sets above correspond respectively to the bit vector triples used to represent zeros, subnormal numbers, normal numbers and infinities in IEEE-754, with the three components storing respectively sign, exponent and significand of the floating-point number.[4] We will write informally -0 and $+0$ to refer to the two elements of $\mathrm{FZ}_{\varepsilon,\sigma}$.

We fix a total order $\sqsubseteq$ over $\mathrm{F}_{\varepsilon,\sigma}$ such that $(s_1,e_1,m_1) \sqsubseteq (s_2,e_2,m_2)$ if one of the following holds:

- $s_1 = 1, s_2 = 0$
- $s_1 = 0, s_2 = 0,\ \mathrm{ub}_\varepsilon(e_1) < \mathrm{ub}_\varepsilon(e_2)$
- $s_1 = 0, s_2 = 0,\ \mathrm{ub}_\varepsilon(e_1) = \mathrm{ub}_\varepsilon(e_2),\ \mathrm{ub}_\sigma(m_1) \leqslant \mathrm{ub}_\sigma(m_2)$
- $s_1 = 1, s_2 = 1,\ \mathrm{ub}_\varepsilon(e_2) < \mathrm{ub}_\varepsilon(e_1)$
- $s_1 = 1, s_2 = 1,\ \mathrm{ub}_\varepsilon(e_1) = \mathrm{ub}_\varepsilon(e_2),\ \mathrm{ub}_\sigma(m_2) \leqslant \mathrm{ub}_\sigma(m_1)$

We extend $\sqsubseteq$ to a partial order on $\mathbb{F}_{\varepsilon,\sigma}$ by $\mathrm{NaN} \sqsubseteq \mathrm{NaN}$.

As discussed in Section IV, we define operations over $\mathbb{F}_{\varepsilon,\sigma}$ analogously to those defined over $\mathbb{R}^*$ by mapping floating-point values to extended reals, performing the corresponding extended reals operation and then rounding the result back to a floating-point value. To formalize this we define a function $\mathrm{v}_{\varepsilon,\sigma} : \mathbb{F}_{\varepsilon,\sigma} \to \mathbb{R}^*$ which maps each floating-point number to the extended real it represents. Let $\mathrm{bias}(\varepsilon) = 2^{\varepsilon-1} - 1$.

$$\mathrm{v}_{\varepsilon,\sigma}(f) = \mathrm{v}_{\varepsilon,\sigma}((s,e,m)) =$$
$$\begin{cases}
0 & f \in \mathrm{FZ}_{\varepsilon,\sigma} \\
(-1)^{\mathrm{ub}_1(s)} \cdot 2^{1-\mathrm{bias}(\varepsilon)} \cdot (0 + \frac{\mathrm{ub}_{\sigma-1}(m)}{2^{\sigma-1}}) & f \in \mathrm{FS}_{\varepsilon,\sigma} \\
(-1)^{\mathrm{ub}_1(s)} \cdot 2^{\mathrm{ub}_\varepsilon(e)-\mathrm{bias}(\varepsilon)} \cdot (1 + \frac{\mathrm{ub}_{\sigma-1}(m)}{2^{\sigma-1}}) & f \in \mathrm{FN}_{\varepsilon,\sigma} \\
(-1)^{\mathrm{ub}_1(s)} \cdot (+\infty) & f \in \mathrm{FI}_{\varepsilon,\sigma}
\end{cases}$$
$$\mathrm{v}_{\varepsilon,\sigma}(\mathrm{NaN}) = \mathrm{NaN}$$

For brevity we will write just v in place of $\mathrm{v}_{\varepsilon,\sigma}$ when the values ε and σ are clear from context or not important. One can show that v is injective over $\mathbb{F}_{\varepsilon,\sigma} \setminus \mathrm{FZ}_{\varepsilon,\sigma}$ and monotonic. Thanks to the latter we have that both $\overline{\mathrm{v}}$ and $\underline{\mathrm{v}}$ are well defined[5] and so the function round can be used to map back from $\mathbb{R}^*$ to $\mathbb{F}_{\varepsilon,\sigma}$. The auxiliary predicates used in the definition of rounding in the case of $X = \mathbb{F}_{\varepsilon,\sigma}$ are defined in Table VI. Both $\mathrm{lh}_{\mathbb{F}_{\varepsilon,\sigma}}$ and $\mathrm{tb}_{\mathbb{F}_{\varepsilon,\sigma}}$ use a set of floating-point numbers with one extra significand bit. This is

[3]Allowing arbitrary values for ε and σ is strictly a generalization of IEEE-754, which only defines a handful of precisions. However, doing so supports a wider range of applications with little additional notation and effort.

[4]The significand component has length $\sigma - 1$ because the hidden bit, which is 1 for normal numbers, is not explicitly represented.

[5]These are surjections for all points except $\mathrm{FZ}_{\varepsilon,\sigma}$ which has the curious property that $-0 = \overline{\mathrm{v}}(0) \sqsubseteq \underline{\mathrm{v}}(0) = +0$.

TABLE VI
AUXILIARY PREDICATES FOR ROUND IN THE CASE $X = \mathbb{F}_{\varepsilon,\sigma}$

$$\begin{array}{rcl}
\mathrm{ev}_{\mathbb{F}_{\varepsilon,\sigma}}(f) & := & f = (s,e,m) \in \mathrm{F}_{\varepsilon,\sigma} \wedge \mathrm{ev}_{\mathbb{Z}^*}(\mathrm{ub}_{\sigma-1}(m)) \\
\mathrm{lh}_{\mathbb{F}_{\varepsilon,\sigma}}(r,\mathrm{v}) & := & \sigma' = \sigma+1 \wedge \mathrm{v}(\underline{\mathrm{v}}(r)) = \mathrm{v}_{\varepsilon,\sigma'}(\underline{\mathrm{v}_{\varepsilon,\sigma'}}(r)) \\
\mathrm{tb}_{\mathbb{F}_{\varepsilon,\sigma}}(r,\mathrm{v}) & := & \sigma' = \sigma+1 \wedge \mathrm{v}(\underline{\mathrm{v}}(r)) < \mathrm{v}_{\varepsilon,\sigma'}(\underline{\mathrm{v}_{\varepsilon,\sigma'}}(r)) = \\
& & \mathrm{v}_{\varepsilon,\sigma'}(\overline{\mathrm{v}_{\varepsilon,\sigma'}}(r)) < \mathrm{v}_{\varepsilon,\sigma'}(\overline{\mathrm{v}}(r))
\end{array}$$

equivalent to the *guard bit* used in hardware implementations, giving a point mid-way between $\overline{\mathrm{v}}$ and $\underline{\mathrm{v}}$. The predicate $\mathrm{tb}_{\mathbb{F}_{\varepsilon,\sigma}}$ captures the property of r being equidistant from $\underline{\mathrm{v}}(r)$ and $\overline{\mathrm{v}}(r)$, which means that any further significand bits would be 0. This is equivalent to the *sticky bit* used in hardware being equal to 0.

B. Relations

Having defined a universe for the models, we next define various relations which will be used as the interpretation of predicates in the theory of floating-point. Every relation is parameterized by a floating-point domain, so each definition here actually describes a whole *family* of relations.

1) Unary Relations: We consider the following unary relations (subsets) for classifying floating-point numbers as well as determining their sign, if applicable (see Section VIII-B for further discussion of sign) .[6]

$$\begin{array}{rcl}
\mathrm{isNeg}_{\varepsilon,\sigma} & = & \{f \in \mathrm{F}_{\varepsilon,\sigma} \mid f = (1,e,m)\} \\
\mathrm{isPos}_{\varepsilon,\sigma} & = & \{f \in \mathrm{F}_{\varepsilon,\sigma} \mid f = (0,e,m)\}
\end{array}$$

2) Binary Relations: We define a number of binary relations for comparing floating-point numbers. These are different from the equality and ordering relations on $\mathbb{F}_{\varepsilon,\sigma}$ (i.e., $=$ and $\sqsubseteq$) and those on $\mathbb{R}^*$ (i.e., $=$ and $\leqslant$). Despite their names, they are not actually equality or ordering relations as they do not contain $(\mathrm{NaN},\mathrm{NaN})$ and eq, leq and geq contain both $(+0,-0)$ and $(-0,+0)$.

$$\begin{array}{rcl}
\mathrm{eq}_{\varepsilon,\sigma} & = & \{(f,g) \in \mathrm{F}_{\varepsilon,\sigma} \times \mathrm{F}_{\varepsilon,\sigma} \mid \mathrm{v}(f) = \mathrm{v}(g)\} \\
\mathrm{leq}_{\varepsilon,\sigma} & = & \{(f,g) \in \mathrm{F}_{\varepsilon,\sigma} \times \mathrm{F}_{\varepsilon,\sigma} \mid \mathrm{v}(f) \leqslant \mathrm{v}(g)\} \\
\mathrm{lt}_{\varepsilon,\sigma} & = & \{(f,g) \in \mathrm{F}_{\varepsilon,\sigma} \times \mathrm{F}_{\varepsilon,\sigma} \mid \mathrm{v}(f) < \mathrm{v}(g)\} \\
\mathrm{geq}_{\varepsilon,\sigma} & = & \{(f,g) \in \mathrm{F}_{\varepsilon,\sigma} \times \mathrm{F}_{\varepsilon,\sigma} \mid \mathrm{v}(f) \geqslant \mathrm{v}(g)\} \\
\mathrm{gt}_{\varepsilon,\sigma} & = & \{(f,g) \in \mathrm{F}_{\varepsilon,\sigma} \times \mathrm{F}_{\varepsilon,\sigma} \mid \mathrm{v}(f) > \mathrm{v}(g)\}
\end{array}$$

C. Operations

Similarly to relations, we define families (parameterized by domains) of functions which will serve as the interpretation of various operations in the theory of floating-point.

[6]These definitions imply $f = \mathrm{NaN} \Leftrightarrow \neg(\mathrm{isNeg}_{\varepsilon,\sigma}(f) \vee \mathrm{isPos}_{\varepsilon,\sigma}(f))$.

a) Sign Operations: Two operations, negation and absolute value, manipulate the sign of the number. Since the domains of floating-point numbers are symmetric around 0, there is no need for rounding and the operations can be defined directly on the floating-point bit vectors without using $\mathbb{R}^*$.

$$\begin{aligned}
&\mathrm{neg}_{\varepsilon,\sigma} : \mathbb{F}_{\varepsilon,\sigma} \to \mathbb{F}_{\varepsilon,\sigma} \\
&\mathrm{neg}_{\varepsilon,\sigma}(f) \;=\; \begin{cases} (\neg s, e, m) & f = (s,e,m) \in \mathrm{F}_{\varepsilon,\sigma} \\ \mathrm{NaN} & f = \mathrm{NaN} \end{cases}
\end{aligned}$$

$$\begin{aligned}
&\mathrm{abs}_{\varepsilon,\sigma} : \mathbb{F}_{\varepsilon,\sigma} \to \mathbb{F}_{\varepsilon,\sigma} \\
&\mathrm{abs}_{\varepsilon,\sigma}(f) \;=\; \begin{cases} (0, e, m) & f = (s,e,m) \in \mathrm{F}_{\varepsilon,\sigma} \\ \mathrm{NaN} & f = \mathrm{NaN} \end{cases}
\end{aligned}$$

b) Arithmetic Operations: The main operations on floating-point numbers are those that correspond to the operations on an ordered field. They are defined by mapping arguments to $\mathbb{R}^*$ with v, performing the corresponding operation in $\mathbb{R}^*$, and finally mapping the result back with round.

$$\begin{aligned}
&\mathrm{add}_{\varepsilon,\sigma} : \mathrm{RM} \times \mathbb{F}_{\varepsilon,\sigma} \times \mathbb{F}_{\varepsilon,\sigma} \to \mathbb{F}_{\varepsilon,\sigma} \\
&\mathrm{add}_{\varepsilon,\sigma}(rm, f, g) \;=\; \mathrm{rnd}(\mathrm{v}, rm, \mathrm{addSign}(rm, f, g), \mathrm{v}(f) + \mathrm{v}(g))
\end{aligned}$$

$$\begin{aligned}
&\mathrm{sub}_{\varepsilon,\sigma} : \mathrm{RM} \times \mathbb{F}_{\varepsilon,\sigma} \times \mathbb{F}_{\varepsilon,\sigma} \to \mathbb{F}_{\varepsilon,\sigma} \\
&\mathrm{sub}_{\varepsilon,\sigma}(rm, f, g) \;=\; \mathrm{rnd}(\mathrm{v}, rm, \mathrm{subSign}(rm, f, g), \mathrm{v}(f) - \mathrm{v}(g))
\end{aligned}$$

$$\begin{aligned}
&\mathrm{mul}_{\varepsilon,\sigma} : \mathrm{RM} \times \mathbb{F}_{\varepsilon,\sigma} \times \mathbb{F}_{\varepsilon,\sigma} \to \mathbb{F}_{\varepsilon,\sigma} \\
&\mathrm{mul}_{\varepsilon,\sigma}(rm, f, g) \;=\; \mathrm{rnd}(\mathrm{v}, rm, \mathrm{xorSign}(f, g), \mathrm{v}(f) * \mathrm{v}(g))
\end{aligned}$$

$$\begin{aligned}
&\mathrm{div}_{\varepsilon,\sigma} : \mathrm{RM} \times \mathbb{F}_{\varepsilon,\sigma} \times \mathbb{F}_{\varepsilon,\sigma} \to \mathbb{F}_{\varepsilon,\sigma} \\
&\mathrm{div}_{\epsilon,\sigma}(rm, f, g) \;= \\
&\quad \begin{cases} \mathrm{neg}_{\varepsilon,\sigma}(\mathrm{rnd}(v, rm, \top, -(\mathrm{v}(f)/\mathrm{v}(g)))) & \mathrm{xorSign}(f, g) \\ \mathrm{rnd}(v, rm, \bot, \mathrm{v}(f)/\mathrm{v}(g)) & \neg\mathrm{xorSign}(f, g) \end{cases}
\end{aligned}$$

$$\begin{aligned}
&\mathrm{fma}_{\varepsilon,\sigma} : \mathrm{RM} \times \mathbb{F}_{\varepsilon,\sigma} \times \mathbb{F}_{\varepsilon,\sigma} \times \mathbb{F}_{\varepsilon,\sigma} \to \mathbb{F}_{\varepsilon,\sigma} \\
&\mathrm{fma}_{\varepsilon,\sigma}(rm, f, g, h) \;= \\
&\quad \mathrm{rnd}(\mathrm{v}, rm, \mathrm{fmaSign}(rm, f, g, h), (\mathrm{v}(f) * \mathrm{v}(g)) + \mathrm{v}(h))
\end{aligned}$$

The $\mathrm{addSign}$, $\mathrm{subSign}$, $\mathrm{xorSign}$ and $\mathrm{fmaSign}$ predicates above are defined as follows ($\oplus$ denotes exclusive or):

$$\mathrm{addSign}(rm, f, g) \;:=\; \begin{cases} \mathrm{isNeg}(f) \wedge \mathrm{isNeg}(g) & rm \neq \mathrm{rtn} \\ \mathrm{isNeg}(f) \vee \mathrm{isNeg}(g) & rm = \mathrm{rtn} \end{cases}$$

$$\mathrm{xorSign}(f, g) \;:=\; \mathrm{isNeg}(f) \oplus \mathrm{isNeg}(g)$$

$$\mathrm{fmaSign}(rm, f, g, h) \;:=\; \mathrm{addSign}(rm, \mathrm{mul}_{\varepsilon,\sigma}(rm, f, g), h)$$

$$\mathrm{subSign}(rm, f, g) \;:=\; \mathrm{addSign}(rm, f, \mathrm{neg}_{\varepsilon,\sigma}(g))$$

Note that since $\mathrm{fma}_{\varepsilon,\sigma}$ only calls round once, it is not the same as $\mathrm{add}_{\varepsilon,\sigma}(rm, \mathrm{mul}_{\varepsilon,\sigma}(rm, a, b), c)$. Also, $\mathrm{div}_{\epsilon,\sigma}$ is equal to $\mathrm{rnd}(\mathrm{v}, rm, \mathrm{xorSign}(f, g), \mathrm{v}(f)/\mathrm{v}(g))$ at all points except positive numbers divided by -0, where the "obvious" definition gives positive infinity while the definition given above gives the correct result of minus infinity.

c) Additional operations: IEEE-754 defines a square root function which returns the floating-point number nearest to the square root of the real represented by the input number. Also $\mathrm{sqrt}(rm, -0) = -0$ since -0 represents $0 \in \mathbb{R}^*$, whose square root is $0 \in \mathbb{R}^*$ and the sign is inherited when rounding is performed.

$$\mathrm{sqrt}_{\varepsilon,\sigma} : \mathrm{RM} \times \mathbb{F}_{\varepsilon,\sigma} \to \mathbb{F}_{\varepsilon,\sigma}$$
$$\mathrm{sqrt}_{\varepsilon,\sigma}(rm, f) = \begin{cases} \mathrm{rnd}(\mathrm{v}, rm, \mathrm{isNeg}(f), z) & 0 \leqslant x = \mathrm{v}(f), \\ & z \cdot z = x,\ 0 \leqslant z \\ \mathrm{NaN} & \text{otherwise} \end{cases}$$

While this is the natural definition of square root, it results in a few unexpected consequences. For example $\mathrm{mul}(rm, \mathrm{sqrt}(rm_1, x), \mathrm{sqrt}(rm_2, x))$ is not guaranteed to be equal to x and, depending on the three rounding modes may give an infinity, even when x is a normal number.[7]

Let $\mathrm{w}_{\varepsilon,\sigma}$ be the restriction of $\mathrm{v}_{\varepsilon,\sigma}$ to $\mathrm{FI}_{\varepsilon,\sigma} \cup \{\mathrm{NaN}\} \cup \{f \in \mathbb{F}_{\varepsilon,\sigma} \mid \mathrm{v}_{\varepsilon,\sigma}(f) \in \mathbb{Z}\}$. The following operation rounds back to a subset of $\mathbb{F}_{\varepsilon,\sigma}$, effectively rounding the value to a whole number representable in the given floating-point format:

$$\mathrm{rti}_{\varepsilon,\sigma} : \mathrm{RM} \times \mathbb{F}_{\varepsilon,\sigma} \to \mathbb{F}_{\varepsilon,\sigma}$$
$$\mathrm{rti}_{\varepsilon,\sigma}(rm, f) = \mathrm{rnd}(\mathrm{w}_{\varepsilon,\sigma}, rm, \mathrm{isNeg}(f), \mathrm{v}(r))$$

Let $\mathrm{in} : \mathbb{Z}^* \to \mathbb{R}^*$ with $\mathrm{in}(z) = z$.[8] Similarly to $\mathrm{rti}_{\varepsilon,\sigma}$, the remainder operation requires rounding an intermediate value to an integer:

$$\mathrm{rem}_{\varepsilon,\sigma} : \mathrm{RM} \times \mathbb{F}_{\varepsilon,\sigma} \times \mathbb{F}_{\varepsilon,\sigma} \to \mathbb{F}_{\varepsilon,\sigma}$$
$$\mathrm{rem}_{\varepsilon,\sigma}(rm, f, g) =$$
$$\begin{cases} f & f \notin \mathrm{FI}_{\varepsilon,\sigma} \cup \{\mathrm{NaN}\}, g \in \mathrm{FI}_{\varepsilon,\sigma} \\ \mathrm{NaN} & f \in \mathrm{FI}_{\varepsilon,\sigma} \cup \{\mathrm{NaN}\} \\ \mathrm{NaN} & g \in \mathrm{FZ}_{\varepsilon,\sigma} \cup \{\mathrm{NaN}\} \\ \mathrm{rnd}(\mathrm{v}, rm, \mathrm{isNeg}(f), x) & x = \mathrm{v}(f) - (\mathrm{v}(g) * y), \\ & y = \mathrm{rnd}(\mathrm{in}, rm, \perp, \mathrm{v}(f)/\mathrm{v}(g)) \end{cases}$$
$$\mathrm{remrne}_{\varepsilon,\sigma} : \mathbb{F}_{\varepsilon,\sigma} \times \mathbb{F}_{\varepsilon,\sigma} \to \mathbb{F}_{\varepsilon,\sigma}$$
$$\mathrm{remrne}_{\varepsilon,\sigma}(f, g) = \mathrm{rem}_{\varepsilon,\sigma}(\mathrm{rne}, f, g)$$

Note that the remainder computed as above is always exact when rne is used. This remainder function is the one used by the C standard library but is not necessarily the same as the intuitive idea of remainder which can be computed via: $\mathrm{fma}_{\varepsilon,\sigma}(rm, \mathrm{neg}_{\varepsilon,\sigma}(\mathrm{div}_{\varepsilon,\sigma}(rm, f, g)), g, f)$. The next two operations, the maximum and minimum of two floating-point numbers, are specified only partially: when the two arguments have the same value in $\mathbb{R}^*$, either one of the arguments can be returned.

$$\mathrm{max}_{\varepsilon,\sigma} : \mathbb{F}_{\varepsilon,\sigma} \times \mathbb{F}_{\varepsilon,\sigma} \to \mathbb{F}_{\varepsilon,\sigma}$$
$$\mathrm{max}_{\varepsilon,\sigma}(f, g) = \begin{cases} f & \mathrm{gt}_{\varepsilon,\sigma}(f, g) \text{ or } g = \mathrm{NaN} \\ g & \mathrm{gt}_{\varepsilon,\sigma}(g, f) \text{ or } f = \mathrm{NaN} \\ h & h \in \{f, g\},\ \mathrm{eq}_{\varepsilon,\sigma}(f, g) \end{cases}$$

$$\mathrm{min}_{\varepsilon,\sigma} : \mathbb{F}_{\varepsilon,\sigma} \times \mathbb{F}_{\varepsilon,\sigma} \to \mathbb{F}_{\varepsilon,\sigma}$$
$$\mathrm{min}_{\varepsilon,\sigma}(f, g) = \begin{cases} f & \mathrm{lt}_{\varepsilon,\sigma}(f, g) \text{ or } g = \mathrm{NaN} \\ g & \mathrm{lt}_{\varepsilon,\sigma}(g, f) \text{ or } f = \mathrm{NaN} \\ h & h \in \{f, g\},\ \mathrm{eq}_{\varepsilon,\sigma}(f, g) \end{cases}$$

[7] $x \in \mathbb{R}^*$ and thus z is uniquely defined. Given $f \in \mathbb{F}_{\varepsilon,\sigma}$ there is not necessarily a $g \in \mathbb{F}_{\varepsilon,\sigma}$ such that $f = \mathrm{mul}_{\varepsilon,\sigma}(rm, g, g)$ and $\mathrm{leq}_{\varepsilon,\sigma}(0, g)$. Furthermore, in the case of subnormal numbers, there are potentially multiple, distinct, positive square roots. Thus care must be taken with implicit definitions of sqrt just using mul or fma.

[8] We consider $\mathbb{Z}$ a subset of $\mathbb{R}$ and hence $\mathbb{Z}^*$ a subset of $\mathbb{R}^*$.

Note that the underspecification is an issue only when one of the inputs to $\max_{\varepsilon,\sigma}$ or $\min_{\varepsilon,\sigma}$ is -0 and the other is $+0$. However, it means that we consider as acceptable models any structures with function families $\max_{\varepsilon,\sigma}$ and $\min_{\varepsilon,\sigma}$ that satisfy the specifications above, regardless of whether they return -0 or $+0$ for $(-0,+0)$, and for $(+0,-0)$. This is necessary because IEEE-754 itself allows either value to be returned, and compliant implementations do vary. For example, on some Intel processors the result returned by the x87 and SSE units is different.

All the preceding operations have floating-point input and outputs in the same set, $\mathbb{F}_{\varepsilon,\sigma}$. To convert between different floating-point domains the following map is needed:

$$\begin{aligned}&\mathrm{cast}_{\varepsilon,\sigma,\varepsilon',\sigma'} : \mathrm{RM} \times \mathbb{F}_{\varepsilon',\sigma'} \to \mathbb{F}_{\varepsilon,\sigma}\\ &\mathrm{cast}_{\varepsilon',\sigma',\varepsilon,\sigma}(rm, f) \;=\; \mathrm{rnd}(\mathrm{v}_{\varepsilon,\sigma}, rm, \mathrm{isNeg}(f), \mathrm{v}_{\varepsilon',\sigma'}(f))\end{aligned}$$

If $\varepsilon \geqslant \varepsilon'$ and $\sigma \geqslant \sigma'$, the rounding mode argument is irrelevant since then all values of $\mathbb{F}_{\varepsilon',\sigma'}$ are representable exactly in $\mathbb{F}_{\varepsilon,\sigma}$. However, this cannot be regarded as syntactic sugar because whether a value is a normal or a subnormal number does change depending on the floating-point domain.

D. Combinations with Other Theories

For many applications, the theory of floating-point is not sufficient to reason about the full problem; other theories such as integers, bit vectors, or reals are needed as well. This section describes the extensions to the intended models required to account for these additional domains, and possible mappings between them. IEEE-754 includes a number of functions to convert to "integer formats." We define here such conversions as well as extensions to IEEE-754 covering conversion to real and from real and bit vectors. Many of the additional operations are underspecified in that *out of bounds* and other *error* conditions do not have prescribed return values.

1) Real Numbers: For a model of the theory of floating-point to also be a model of the theory of reals, its universe has to be extended to a disjoint union with $\mathbb{R}$. Using the connections between $\mathbb{F}_{\varepsilon,\sigma}$ and $\mathbb{R}^*$, we add the following two conversion operations:

$$\begin{aligned}&\mathrm{realToFP}_{\varepsilon,\sigma} : \mathrm{RM} \times \mathbb{R} \to \mathbb{F}_{\varepsilon,\sigma}\\ &\mathrm{realToFP}_{\varepsilon,\sigma}(rm, r) \;=\; \mathrm{rnd}(\mathrm{v}, rm, \bot, r)\end{aligned}$$

$$\begin{aligned}&\mathrm{FPToReal}_{\varepsilon,\sigma} : \mathbb{F}_{\varepsilon,\sigma} \to \mathbb{R}\\ &\mathrm{FPToReal}_{\varepsilon,\sigma}(f) \;=\; \begin{cases}\mathrm{v}(f) & \mathrm{v}(f) \in \mathbb{R}\\ x & x \in \mathbb{R}, \text{ otherwise}\end{cases}\end{aligned}$$

We do not specify what the value of $\mathrm{FPToReal}_{\varepsilon,\sigma}(f)$ is when f does not correspond to a real number. This means again that we accept as a model any structure with a function family $\mathrm{FPToReal}_{\varepsilon,\sigma}$ that satisfies the specification above.

2) Fixed-size Bit Vectors: Similarly to the previous case, to form a joint model of the theories of floating-point and fixed-width bit vectors, the domain must be extended to a disjoint union with $\mathbb{BV}_\nu$ for every $\nu > 0$. Let $\bullet : \mathbb{BV}_\mu \times \mathbb{BV}_\nu \to \mathbb{BV}_{\mu+\nu}$ be the bit vector concatenation function for each $\mu, \nu > 0$. The following function converts a bit vector of length $\varepsilon + \sigma$, with $\varepsilon, \sigma > 1$, to a floating-point number in $\mathbb{F}_{\varepsilon,\sigma}$ by slicing the bit vector in three:

$$\begin{aligned}&\mathrm{bitpatternToFP}_{\varepsilon,\sigma} : \mathbb{BV}_{\varepsilon+\sigma} \to \mathbb{F}_{\varepsilon,\sigma}\\ &\mathrm{bitpatternToFP}_{\varepsilon,\sigma}(b) \;=\; \begin{cases}(s,e,m) & b = s \bullet e \bullet m,\\ & (s,e,m) \in \mathrm{F}_{\varepsilon,\sigma}\\ \mathrm{NaN} & \text{otherwise}\end{cases}\end{aligned}$$

The function $\mathrm{bitpatternToFP}$ is not injective as there are multiple bit-patterns which represent NaN. This implies that it is not possible to give a reverse map without fixing the encoding of NaN to a particular value.

The next two functions first convert the bit vector to the integer value it denotes in binary, in 2's complement and unsigned format respectively, and then round that value to the corresponding floating-point. The last two functions do the inverse conversion.

$$\mathrm{sIntToFP}_{\nu,\varepsilon,\sigma} : \mathrm{RM} \times \mathbb{BV}_\nu \to \mathbb{F}_{\varepsilon,\sigma}$$
$$\mathrm{sIntToFP}_{\nu,\varepsilon,\sigma}(rm, b) \;=\; \mathrm{rnd}(\mathrm{v}, rm, \bot, \mathrm{sb}_\nu(b))$$

$$\mathrm{uIntToFP}_{\nu,\varepsilon,\sigma} : \mathrm{RM} \times \mathbb{BV}_\nu \to \mathbb{F}_{\varepsilon,\sigma}$$
$$\mathrm{uIntToFP}_{\nu,\varepsilon,\sigma}(rm, b) \;=\; \mathrm{rnd}(\mathrm{v}, rm, \bot, \mathrm{ub}_\nu(b))$$

$$\mathrm{FPToSInt}_{\nu,\varepsilon,\sigma} : \mathrm{RM} \times \mathbb{F}_{\varepsilon,\sigma} \to \mathbb{BV}_\nu$$
$$\mathrm{FPToSInt}_{\nu,\varepsilon,\sigma}(rm, f) \;=\; \begin{cases} b & \mathrm{sb}_\nu(b) = \mathrm{rnd}(\mathrm{in}, rm, \bot, \mathrm{v}(f)) \\ \mathrm{NaN} & \text{otherwise} \end{cases}$$

$$\mathrm{FPToUInt}_{\nu,\varepsilon,\sigma} : \mathrm{RM} \times \mathbb{F}_{\varepsilon,\sigma} \to \mathbb{BV}_\nu$$
$$\mathrm{FPToUInt}_{\nu,\varepsilon,\sigma}(rm, f) \begin{cases} b & \mathrm{ub}_\nu(b) = \mathrm{rnd}(\mathrm{in}, rm, \bot, \mathrm{v}(f)) \\ \mathrm{NaN} & \text{otherwise} \end{cases}$$

VI. From Models to Theory

We now formalize a logical theory of floating-point numbers based on the structures defined in the previous section. We use the version of many-sorted logic adopted by the SMT-LIB 2 standard [5] whose main differences with traditional many-sorted logic is that (i) it allows sorts to be denoted by terms instead of just constants and (ii) it allows sort, function and predicate symbols to be indexed by one or more natural number indices. For example, possible sorts may be not just constants like `Int` but also terms like $\texttt{Array}(\texttt{Int}, \texttt{Real})$ or $\texttt{BitVec}_n$ for all $n > 0$.

A. Preliminaries

A *(logical) signature* is consists of a set of sort symbols (of arity ≥ 0) and a set of function symbols f with an associated *rank*, a tuple $(S_1, \ldots, S_n, S)$ of sort terms specifying the sort of f's arguments, namely, $S_1, \ldots, S_n$, and result, S. Constants are represented by nullary function symbols; predicate symbols by function symbols whose return sort is a distinguished sort `Bool`. Every signature Σ is assumed to contain `Bool` and constants `true` and `false` of that sort, as well as an overloaded symbol $=$ of rank $(S, S, \texttt{Bool})$ for each sort S, for the identity relation over S. Given a set of sorted variables for each of the sorts in Σ, well-sorted terms and well-sorted formulas of signature Σ are defined as usual.

For every signature Σ, a Σ-*interpretation* $\mathcal{I}$ is a structure that interprets each sort S in Σ as a non-empty set $[\![S]\!]_\mathcal{I}$, each variable x of sort S as an element $[\![x]\!]_\mathcal{I}$ of $[\![S]\!]_\mathcal{I}$, and each function symbol f of rank $(S_1, \ldots, S_n, S)$ as an element $[\![f]\!]_\mathcal{I}$ of the (total) function space $[\![S_1]\!]_\mathcal{I} \times \cdots \times [\![S_n]\!]_\mathcal{I} \to [\![S]\!]_\mathcal{I}$. Additionally, $\mathcal{I}$ interprets `Bool` as $\mathbb{B} = \{\top, \bot\}$ and each $=$ of rank $(S, S, \texttt{Bool})$ as the function that maps $(x, y) \in S \times S$ to $\top$ iff x is y. For each sort S, $\mathcal{I}$ induces a mapping $[\![\_]\!]_\mathcal{I}$ from terms of sort S to $[\![S]\!]_\mathcal{I}$ as expected. A satisfaction relation $\models$ between Σ-interpretations and Σ-formulas is also defined as expected. A *theory* of signature Σ as a pair $T = (\Sigma, \mathbf{I})$ where $\mathbf{I}$ is a set of Σ-interpretations, the *models of* T, that is closed under variable reassignment.[9] We say that a Σ-formula φ is *satisfiable* (resp., *unsatisfiable*) *in* T if $\mathcal{I} \models \varphi$ for some (resp., no) $\mathcal{I} \in \mathbf{I}$.

[9]That is, every Σ-interpretation that differs from one in $\mathbf{I}$ only in how it interprets the variables is also in $\mathbf{I}$.

TABLE VII
SORTS AND THEIR INTERPRETATION REQUIREMENTS

$$\begin{aligned} [\![\texttt{RoundingMode}]\!]_{\mathcal{I}} &= \mathrm{RM} & [\![\texttt{FloatingPoint}_{\varepsilon,\sigma}]\!]_{\mathcal{I}} &= \mathbb{F}_{\varepsilon,\sigma} \\ [\![\texttt{Real}]\!]_{\mathcal{I}} &= \mathbb{R} & [\![\texttt{BitVec}_{\nu}]\!]_{\mathcal{I}} &= \mathbb{BV}_{\nu} \end{aligned}$$

TABLE VIII
CONSTRUCTOR SYMBOLS AND THEIR INTERPRETATION

Symbols of rank $\texttt{RM}$:

$$\begin{aligned} [\![\texttt{rne}]\!]_{\mathcal{I}} &= \mathrm{rne} & [\![\texttt{rna}]\!]_{\mathcal{I}} &= \mathrm{rna} & [\![\texttt{rtp}]\!]_{\mathcal{I}} &= \mathrm{rtp} \\ [\![\texttt{rtn}]\!]_{\mathcal{I}} &= \mathrm{rtn} & [\![\texttt{rtz}]\!]_{\mathcal{I}} &= \mathrm{rtz} \end{aligned}$$

Symbols of rank $\texttt{FP}_{\varepsilon,\sigma}$:

$$\begin{aligned} [\![\texttt{+oo}_{\varepsilon,\sigma}]\!]_{\mathcal{I}} &= (0, \mathbf{1}_{\varepsilon}, \mathbf{0}_{\sigma-1}) & [\![\texttt{+zero}_{\varepsilon,\sigma}]\!]_{\mathcal{I}} &= (0, \mathbf{0}_{\varepsilon}, \mathbf{0}_{\sigma-1}) \\ [\![\texttt{-oo}_{\varepsilon,\sigma}]\!]_{\mathcal{I}} &= (1, \mathbf{1}_{\varepsilon}, \mathbf{0}_{\sigma-1}) & [\![\texttt{-zero}_{\varepsilon,\sigma}]\!]_{\mathcal{I}} &= (1, \mathbf{0}_{\varepsilon}, \mathbf{0}_{\sigma-1}) \\ [\![\texttt{NaN}_{\varepsilon,\sigma}]\!]_{\mathcal{I}} &= \mathrm{NaN} \end{aligned}$$

Symbols of rank $(\texttt{BV}_1, \texttt{BV}_{\varepsilon}, \texttt{BV}_{\sigma}, \texttt{FP}_{\varepsilon,\sigma})$:

$$[\![\texttt{fp}]\!]_{\mathcal{I}} = \lambda(b_1, b_{\varepsilon}, b_{\sigma-1}).\, \mathrm{bitpatternToFP}_{\varepsilon,\sigma}(b_1 \bullet b_{\varepsilon} \bullet b_{\sigma-1})$$

B. A Theory of Floating-Point Numbers

In the following we define a theory T_{FP} of floating-point numbers in the sense above by specifying a signature Σ_{FP} and a set of $\mathbf{I}_{\mathsf{FP}}$ of Σ_{FP}-interpretations.

The sorts of Σ_{FP} consist, besides `Bool`, of two individual sorts: `RoundingMode` and `Real`; and two sort families: $\texttt{BitVec}_{\nu}$, indexed by an integer $\nu > 0$, and $\texttt{FloatingPoint}_{\varepsilon,\sigma}$, indexed by two integers $\varepsilon, \sigma > 1$. The set of function symbols of Σ_{FP}, and their ranks, is given in Table VIII through X. In those tables, we abbreviate `RoundingMode`, `FloatingPoint`, and `BitVec` respectively as `RM`, `FP`, and `BV`.

We define the set $\mathbf{I}_{\mathsf{FP}}$ as the set of *all possible* Σ_{FP}-interpretations $\mathcal{I}$ that interpret sort and function symbol as specified in Table VII through X in terms of the sets and functions introduced in Section V. Many of the function symbols are *overloaded* for having different ranks; so we specify their interpretation separately for each rank.

As shown in Table VIII, the theory has a family of symbols denoting the floating-point infinities, zeros, and NaN for each pair of exponent and significand length. It also has a ternary function symbol `fp` that constructs a floating point number from a triple of bit vectors respectively storing the sign, exponent and significant. This allows us to represent all non-NaN values with bit-level precision.

Table IX lists function symbols for the various arithmetic operations over floating-point numbers, and provides their semantics in terms of the operations defined in Subsection V-C. The table lists predicate symbols corresponding to the relations defined in Subsection V-B and to the various subsets of $\mathbb{F}_{\varepsilon,\sigma}$. For simplicity and by a slight abuse of notation, we identify functions h of type $D_1 \times \cdots \times D_n \to \mathbb{B}$ with the n-ary relations $\{(x_1, \ldots, x_n) \in D_1 \times \cdots \times D_n \mid h(x_1, \ldots, x_n) = \top\}$.

Finally, Table X lists function symbols corresponding to the various conversion functions introduced in Subsection V-D as well as the casting function between floating-point sets of different precision.

VII. RELATED WORK

The earliest formalizations of floating-point [10] were limited by the diversity of floating-point formats and systems they had to cover. For example, the formalization needed to support a range of bases, as 2, 16 and 10 were all in use. Overflow and underflow were particularly problematic as, again, systems in common usage took very different approaches to handling them. One interesting note is that the motivation behind this early work, to support writing portable numerical software, was also one of the

TABLE IX
MAIN SYMBOLS AND THEIR INTERPRETATION

Symbols of rank $(\mathtt{FP}_{\varepsilon,\sigma}, \mathtt{FP}_{\varepsilon,\sigma})$:

$$[\![\mathtt{fp.abs}]\!]_{\mathcal{I}} = \mathrm{abs}_{\varepsilon,\sigma} \qquad [\![\mathtt{fp.neg}]\!]_{\mathcal{I}} = \mathrm{neg}_{\varepsilon,\sigma}$$

Symbols of rank $(\mathtt{FP}_{\varepsilon,\sigma}, \mathtt{FP}_{\varepsilon,\sigma}, \mathtt{FP}_{\varepsilon,\sigma})$:

$$[\![\mathtt{fp.max}]\!]_{\mathcal{I}} = \max_{\varepsilon,\sigma} \qquad [\![\mathtt{fp.min}]\!]_{\mathcal{I}} = \min_{\varepsilon,\sigma}$$
$$[\![\mathtt{fp.rem}]\!]_{\mathcal{I}} = \mathrm{remrne}_{\varepsilon,\sigma}$$

Symbols of rank $(\mathtt{RM}, \mathtt{FP}_{\varepsilon,\sigma}, \mathtt{FP}_{\varepsilon,\sigma})$:

$$[\![\mathtt{fp.sqrt}]\!]_{\mathcal{I}} = \mathrm{sqrt}_{\varepsilon,\sigma} \qquad [\![\mathtt{fp.roundToIntegral}]\!]_{\mathcal{I}} = \mathrm{rti}_{\varepsilon,\sigma}$$

Symbols of rank $(\mathtt{RM}, \mathtt{FP}_{\varepsilon,\sigma}, \mathtt{FP}_{\varepsilon,\sigma}, \mathtt{FP}_{\varepsilon,\sigma})$:

$$[\![\mathtt{fp.add}]\!]_{\mathcal{I}} = \mathrm{add}_{\varepsilon,\sigma} \qquad [\![\mathtt{fp.sub}]\!]_{\mathcal{I}} = \mathrm{sub}_{\varepsilon,\sigma}$$
$$[\![\mathtt{fp.mul}]\!]_{\mathcal{I}} = \mathrm{mul}_{\varepsilon,\sigma} \qquad [\![\mathtt{fp.div}]\!]_{\mathcal{I}} = \mathrm{div}_{\varepsilon,\sigma}$$

Symbols of rank $(\mathtt{RM}, \mathtt{FP}_{\varepsilon,\sigma}, \mathtt{FP}_{\varepsilon,\sigma}, \mathtt{FP}_{\varepsilon,\sigma}, \mathtt{FP}_{\varepsilon,\sigma})$:

$$[\![\mathtt{fp.fma}]\!]_{\mathcal{I}} = \mathrm{fma}_{\varepsilon,\sigma}$$

Symbols of rank $(\mathtt{FP}_{\varepsilon,\sigma}, \mathtt{Bool})$:

$$[\![\mathtt{fp.isNormal}]\!]_{\mathcal{I}} = \mathrm{FN}_{\varepsilon,\sigma} \qquad [\![\mathtt{fp.isNegative}]\!]_{\mathcal{I}} = \mathrm{isNeg}_{\varepsilon,\sigma}$$
$$[\![\mathtt{fp.isSubnormal}]\!]_{\mathcal{I}} = \mathrm{FS}_{\varepsilon,\sigma} \qquad [\![\mathtt{fp.isPositive}]\!]_{\mathcal{I}} = \mathrm{isPos}_{\varepsilon,\sigma}$$
$$[\![\mathtt{fp.isInfinite}]\!]_{\mathcal{I}} = \mathrm{FI}_{\varepsilon,\sigma} \qquad [\![\mathtt{fp.isZero}]\!]_{\mathcal{I}} = \mathrm{FZ}_{\varepsilon,\sigma}$$
$$[\![\mathtt{fp.isNaN}]\!]_{\mathcal{I}} = \{\mathrm{NaN}\}$$

Symbols of rank $(\mathtt{FP}_{\varepsilon,\sigma}, \mathtt{FP}_{\varepsilon,\sigma}, \mathtt{Bool})$, where $\mathrm{gt}_{\varepsilon,\sigma}$ is the converse of $\mathrm{lt}_{\varepsilon,\sigma}$:

$$[\![\mathtt{fp.lt}]\!]_{\mathcal{I}} = \mathrm{lt}_{\varepsilon,\sigma} \qquad [\![\mathtt{fp.leq}]\!]_{\mathcal{I}} = \mathrm{leq}_{\varepsilon,\sigma}$$
$$[\![\mathtt{fp.gt}]\!]_{\mathcal{I}} = \mathrm{gt}_{\varepsilon,\sigma} \qquad [\![\mathtt{fp.geq}]\!]_{\mathcal{I}} = \mathrm{geq}_{\varepsilon,\sigma}$$
$$[\![\mathtt{fp.eq}]\!]_{\mathcal{I}} = \mathrm{eq}_{\varepsilon,\sigma}$$

TABLE X
CONVERSION SYMBOLS AND THEIR INTERPRETATION

Conversions to floating-point

$$[\![\mathtt{to\_fp}_{\varepsilon,\sigma} : (\mathtt{RM}, \mathtt{FP}_{\varepsilon',\sigma'}, \mathtt{FP}_{\varepsilon,\sigma})]\!]_{\mathcal{I}} = \mathrm{cast}_{\varepsilon',\sigma',\varepsilon,\sigma}$$
$$[\![\mathtt{to\_fp}_{\varepsilon,\sigma} : (\mathtt{BV}_{\varepsilon+\sigma}, \mathtt{FP}_{\varepsilon,\sigma})]\!]_{\mathcal{I}} = \mathrm{bitpatternToFP}_{\varepsilon,\sigma}$$
$$[\![\mathtt{to\_fp}_{\varepsilon,\sigma} : (\mathtt{RM}, \mathtt{Real}, \mathtt{FP}_{\varepsilon,\sigma})]\!]_{\mathcal{I}} = \mathrm{realToFP}_{\varepsilon,\sigma}$$
$$[\![\mathtt{to\_fp}_{\varepsilon,\sigma} : (\mathtt{RM}, \mathtt{BV}_{\nu}, \mathtt{FP}_{\varepsilon,\sigma})]\!]_{\mathcal{I}} = \mathrm{sIntToFP}_{\nu,\varepsilon,\sigma}$$
$$[\![\mathtt{to\_fp\_unsigned}_{\varepsilon,\sigma} : (\mathtt{RM}, \mathtt{BV}_{\nu}, \mathtt{FP}_{\varepsilon,\sigma})]\!]_{\mathcal{I}} = \mathrm{uIntToFP}_{\nu,\varepsilon,\sigma}$$

Conversions from floating-point

$$[\![\mathtt{fp.to\_sbv}_{\nu} : (\mathtt{FP}_{\varepsilon,\sigma}, \mathtt{BV}_{\nu})]\!]_{\mathcal{I}} = \mathrm{FPToSInt}_{\nu,\varepsilon,\sigma}$$
$$[\![\mathtt{fp.to\_ubv}_{\nu} : (\mathtt{FP}_{\varepsilon,\sigma}, \mathtt{BV}_{\nu})]\!]_{\mathcal{I}} = \mathrm{FPToUInt}_{\nu,\varepsilon,\sigma}$$
$$[\![\mathtt{fp.to\_real} : (\mathtt{FP}_{\varepsilon,\sigma}, \mathtt{Real})]\!]_{\mathcal{I}} = \mathrm{FPToReal}_{\nu,\varepsilon,\sigma}$$

drivers of the development of IEEE-754. The first formalization of IEEE-754 [6] is notable on several grounds. It was the first to use a formal language (Z) and to be used to verify algorithms for the basic operations. The verification was manual, using Hoare logic, and the algorithms in question were those implemented in the firmware of the T800 Transputer. During the formalization, a few issues in IEEE-754 were found and the verification uncovered bugs that would have been difficult to find with testing [28]. Foreshadowing the issue in the Pentium 1, bugs were found in the Transputer's handling of floating-point,

introduced by the translation to machine code and manual "tidying up" [17], suggesting a need to extend the proof chain to the whole development process and a need for greater automation.

The FDIV bug in the Pentium 1 and the cost of the resultant recall spurred the use of machine-checked formal proofs in the design of floating-point hardware [20], [21]. To this end, IEEE-754 was formalized in a variety of interactive theorem provers, including Isabelle [29], HOL [11], HOL Light [19] (used by Intel), ACL2 [26] (used by AMD and Centaur), PVS [24] and Coq [14], [23], [7]. These and related approaches [2] share a number of common characteristics due to the provers they targeted. They are all instances of the axiomatic approach described in Section III; generally reduce floating-point numbers to integers and reals; are intended for use in machine checked proofs; and are normally used to verify implementations of floating-point and specific algorithms based on them. In contrast, this work (and its precursor [27]) follows the algebraic approach; builds on computationally simple primitives; and is intended to be a formal reference for automatic theorem provers providing built-in support for reasoning about floating-point arithmetic.

A number of SMT solvers provide support for early versions of our theory by encoding floating-point expressions as bit-vector expressions based on the circuits used to implement floating-point operations. To improve performance, they often rely on over and under approximation schemes. To our knowledge, the earliest implementation of this approach was given in the CBMC model checker [9]. The approach is now used in Z3 [16], MathSAT [12], SONOLAR [22] and CVC4 [3], and improving it remains an active area of research [30]. An alternative approach is based on abstract interpretation. It uses intervals or other abstract domains to over-approximate possible models, and a system of branching and learning similar to the SAT algorithm CDCL to narrow these to particular concrete models [18], [8]. There has also been work to integrate the automated prover Gappa [15] into SMT solvers [13], although these solvers are not known to implement the semantics presented in this paper.

VIII. Discussion

This work originated in the context of the SMT-LIB initiative[10] with the goal of defining a standard, reference SMT-LIB theory of floating-point numbers for SMT solvers. The process of defining this reference theory was guided by three, somewhat conflicting, main principles:

1) minimise the amount of effort required to implement a solver for such a theory;
2) support a wide range of applications (including avoiding hardware or language specific idioms);
3) conform strictly with IEEE-754in the sense that any compliant implementation of IEEE-754 would form a model of the theory and vice versa.

A. Process of Standardisation

An initial proposal for an SMT-LIB floating-point theory was made by P. Rümmer and T. Wahl in 2010 [27]. A second draft based on that proposal was produced by C. Tinelli using initial community feedback. This draft was used by a number of system implementors to produce benchmarks and theory solvers. To draw on the experience of using the theory, a work group was formed to decide on a first official theory definition. The suggestions made and changes requested by the work group were written up by C. Tinelli and M. Brain and released to the SMT-LIB community mailing list. We used feedback and corrections from this final review to prepare the version of the theory presented here. The semantics we developed for this theory, discussed in Section IV, was inspired by the initial work in [27], but differs substantially from that work by providing a bit-precise semantics of IEEE-754 floating-point.

B. Limitations and Omissions

The theory presented here does not cover all of the functionality of IEEE-754. Here is a list of salient omissions and restrictions and their rationale.

[10] See www.smt-lib.org .

- Two radices, 2 and 10, are included in IEEE-754, although compliant implementations are not required to support both. Our theory only covers radix 2 (binary) representations, as decimal representations are not widely implemented, supported by languages, or used.
- IEEE-754 gives four "specification levels" (extended real, floating-point data, representations and bit-strings) with maps between them. The formalization in this paper is based on the second level of specification. Thus there is only one NaN, as quiet and signalling NaN are introduced at level three. Also, NaN is not regarded to have a sign or a payload as these concepts are introduced at level four. As a consequence, functionality that uses these properties of NaN, such as *copy sign*, *sign of* and the *total order* predicate, has been omitted.
- Our formalization does not cover the IEEE-754 notion of exceptions or flags (the default handling of executions). This is simply because there is no notion of execution order in a logical formula, and so there is no meaningful way of expressing those notions directly in theory.
- The 2008 revision of IEEE-754 adds the notion of attributes, a means of attaching implicit parameters to sections of programs. Only the most commonly supported attribute, rounding direction, is modeled in our theory—by adding an explicit rounding mode parameter to operations affected by it. Some of the other attributes could be effectively modeled but we chose not to do so because they are programming language specific.
- We do not support unordered variants of comparison operators (those that return true if one or more of the arguments are NaN) since they are rarely used. It would, however, be relatively simple to model these variants.
- IEEE-754 contains recommendations for trigonometric functions and exponentials but neither are mandated. The accuracy of implementations of these functions vary significantly, making it very hard to come up with logical models that are widely applicable but also meaningfully constrained. Similarly, there are only recommendations for reduction operations, and implementations vary.

IX. Conclusion

This paper presents a formalization of IEEE-754 binary floating-point arithmetic as a logical theory, with the goal of enabling automated bit-precise reasoning about programs using floating-point data. The theory is defined as a set of function symbols that model several floating-point operations, and a set of mathematical structures that act as the intended models. This work represents the culmination of a community-driven process to define a standard theory of floating-point arithmetic for SMT, and is intended to be provide a common formal reference for both developers and users of SMT solvers, especially in the context of verification applications.

Acknowledgments: We would like to to thank François Bobot, David Cok, Alberto Griggio, Florian Lapschies, Leonardo de Moura, Gabriele Paganelli, Cody Roux, and Christoph Wintersteiger for their input, feedback and active involvement in the work group.

References

[1] I. S. Association. IEEE standard for floating-point arithmetic, 2008. http://grouper.ieee.org/groups/754/.
[2] A. Ayad and C. Marché. Multi-prover verification of floating-point programs. In *IJCAR'10*, pages 127–141. Springer, 2010.
[3] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *CAV'11*, CAV'11, pages 171–177. Springer, 2011.
[4] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli. Satisfiability modulo theories. In A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 185, chapter 26, pages 825–885. IOS Press, February 2009.
[5] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. Technical report, Department of Computer Science, The University of Iowa, 2010.
[6] G. Barrett. Formal methods applied to a floating-point number system. *IEEE Trans. Softw. Eng.*, 15(5):611–621, May 1989.
[7] S. Boldo and G. Melquiond. Flocq: A Unified Library for Proving Floating-point Algorithms in Coq. In *ARITH'11*, pages 243–252. IEEE, July 2011.
[8] M. Brain, V. D'Silva, A. Griggio, L. Haller, and D. Kroening. Interpolation-based verification of floating-point programs with abstract CDCL. In *SAS'13*, pages 412–432. Springer, June 2013.
[9] A. Brillout, D. Kroening, and T. Wahl. Mixed abstractions for floating-point arithmetic. In *FMCAD'09*. Springer, 2009.

[10] W. S. Brown. A simple but realistic model of floating-point computation. *ACM Trans. Math. Softw.*, 7(4):445–480, Dec. 1981.

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

[12] A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani. The MathSAT5 SMT Solver. In *TACAS'13*, volume 7795. Springer, 2013.

[13] S. Conchon, G. Melquiond, C. Roux, and M. Iguernelala. Built-in treatment of an axiomatic floating-point theory for SMT solvers. In *SMT'12*, pages 12–21. EasyChair, 2013.

[14] M. Daumas, L. Rideau, and L. Théry. A generic library for floating-point numbers and its application to exact computing. In *TPHOL'01*, pages 169–184. Springer, 2001.

[15] F. de Dinechin, C. Lauter, and G. Melquiond. Certifying the floating-point implementation of an elementary function using Gappa. *IEEE Trans. on Computers*, 60(2), 2011.

[16] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS'08*, pages 337–340. Springer, 2008.

[17] J. Gibbons. Formal methods: Why should I care? – the development of the T800 transputer floating-point unit. In *New Zealand Computer Society Conference*, pages 207–217, 1993.

[18] L. Haller, A. Griggio, M. Brain, and D. Kroening. Deciding floating-point logic with systematic abstraction. In *FMCAD'12*, pages 131–140, October 2012.

[19] J. Harrison. Floating point verification in HOL light: The exponential function. In *AMAST'97*, pages 246–260. Springer, 1997.

[20] J. Harrison. Floating-point verification using theorem proving. In *SFM'06*, pages 211–242. Springer, 2006.

[21] J. Harrison. Floating-point verification. *J. UCS*, 13(5):629–638, 2007.

[22] E. V. Jan Peleska and F. Lapschies. Automated test case generation with SMT-solving and abstract interpretation. In *NFM'11*, LNCS, pages 298–312. Springer, April 2011.

[23] G. Melquiond. Floating-point arithmetic in the Coq system. *Information and Computation*, 216:14–23, July 2012.

[24] P. S. Miner. Defining the IEEE-854 floating-point standard in PVS. Technical report, NASA Langley Research Center, 1995.

[25] D. Monniaux. The pitfalls of verifying floating-point computations. *ACM Trans. Program. Lang. Syst.*, 30(3), 2008.

[26] J. Moore, T. Lynch, and M. Kaufmann. A mechanically checked proof of the AMD5K86TM floating-point division program. *IEEE Trans. on Computers*, 47(9):913–926, Sep 1998.

[27] P. Ruemmer and T. Wahl. An SMT-LIB theory of binary floating-point arithmetic. In *SMT'10*, 2010.

[28] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal methods: Practice and experience. *ACM Comput. Surv.*, 41(4):19:1–19:36, Oct. 2009.

[29] L. Yu. A formal model of IEEE floating point arithmetic, July 2013. http://afp.sf.net/entries/IEEE_Floating_Point.shtml.

[30] A. Zeljic, C. M. Wintersteiger, and P. Rümmer. Approximations for model construction. In *IJCAR'14*, pages 344–359. Springer, 2014.