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

Martin Brain, Cesare Tinelli, Philipp Rümmer, Thomas Wahl
(and the rest of the SMT community)

University of Oxford

June 24, 2015
Hasn’t this been done before?

**Isabelle**  A formal model of IEEE floating point arithmetic

**HOL**  Interpretation of IEEE-854 floating-point standard and definition in the HOL system.

**HOL Light**  Floating point verification in HOL light: The exponential function (Intel)

**ACL2**  A mechanically checked proof of the AMD5K86TM floating-point division program (AMD and Centaur)

**PVS**  Defining the IEEE-854 floating-point standard in PVS

**Coq**  A generic library for floating-point numbers and its application to exact computing

**Coq**  Floating-point arithmetic in the Coq system

**Coq**  Flocq: A Unified Library for Proving Floating-point Algorithms in Coq
Hasn’t this been done before?

**Isabelle**  A formal model of IEEE floating point arithmetic

**HOL**  Interpretation of IEEE-854 floating-point standard and definition in the HOL system.

**HOL Light**  Floating point verification in HOL light: The exponential function (Intel)

**ACL2**  A mechanically checked proof of the AMD5K86TM floating-point division program (AMD and Centaur)

**PVS**  Defining the IEEE-854 floating-point standard in PVS

**Coq**  A generic library for floating-point numbers and its application to exact computing

**Coq**  Floating-point arithmetic in the Coq system

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

... there is another way to think about theorem-proving ...
Is there an $x$ and $y$ such that ...

\[ 0 < x \]
\[ 0 < y \]
\[ x + y < x \]
Is there an $x$ and $y$ such that ...

\[
\begin{align*}
0 \spadesuit x \\
0 \clubsuit y \\
x \heartsuit y \spadesuit x
\end{align*}
\]
Is there an $x$ and $y$ such that ...

\[ 0 \clubsuit x \]
\[ 0 \clubsuit y \]
\[ x \spadesuit y \spadesuit x \]

It depends on the interpretation (of \clubsuit and \spadesuit)!

\[ D = \mathbb{Z} \]
\[ [\clubsuit] = \mathbb{Z} \]
\[ [\spadesuit] = +\mathbb{Z} \]

NO!
Is there an $x$ and $y$ such that ...

$0 \clubsuit x$

$0 \clubsuit y$

$x \spadesuit y \ spadesuit x$

It depends on the interpretation (of $\clubsuit$ and $\spadesuit$)!

$D = \{00, 01, 10, 11\}$

$[\clubsuit] = \text{bvult}$

$[\spadesuit] = \text{bvplus}$

Yes ($x = 01, y = 11$)
First Order Logic

**Syntax**

Fix a *signature* $\Sigma$

(i.e. $\Sigma = \{\clubsuit, \spadesuit\}$)

**Semantics**

An *interpretation* is

$M = (D, \llbracket . \rrbracket : \Sigma \rightarrow (2^{D^n}))$

**Satisfiability**

An interpretation $M$ *satisfies* a formula $\phi$:

$$M \models \phi$$

If $\phi$ evaluated over $D$ (using $\llbracket . \rrbracket$) is true.
How Do We Fix The *Meaning* of Symbols?

**Option 1 – Axiomatic**

\[ M \models \text{Axioms} \Rightarrow \phi \]

\[
\text{Axioms} = \forall a, b, c \cdot a \circ b \land b \circ c \Rightarrow a \circ c
\]

\[
= \forall a \cdot \neg a \circ a
\]

... 

Formalisation is solver *INPUT*.

**Pros**
- Easy to implement
- Flexible
- Can add theorems

**Cons**
- All formulae quantified
- Axioms not always simple
- Hard to solve
How Do We Fix The *Meaning* of Symbols?

Option 2 – Algebraic

Fix signature $\Sigma'$ and its interpretation $M' = (D, [.] : \Sigma' \rightarrow (2^{D^n}))$.

$$D = \mathbb{Z} \quad [\spadesuit] = \langle \mathbb{Z} \quad [\clubsuit] = +\mathbb{Z}$$

Is there $M$ extension of $M'$ such that:

$$M \models \phi$$

Formalisation is solver *SPECIFICATION*.

**Pros**
+ Fast decision procedures
+ Counter-examples
+ Few quantifiers

**Cons**
- Theory has to be built into solver
- Implementation harder
How Do We Fix The *Meaning* of Symbols?

**Option 2 – Algebraic**

Fix signature $\Sigma'$ and its interpretation $M' = (D, [\cdot] : \Sigma' \to (2^{D^n}))$.

$$D = \mathbb{Z} \quad [\heartsuit] = <_\mathbb{Z} \quad [\spadesuit] = +_\mathbb{Z}$$

Is there $M$ extension of $M'$ such that:

$$M \models \phi$$

Formalisation is solver *SPECIFICATION*.

**Pros**

+ Fast decision procedures
+ Counter-examples
+ Few quantifiers

**Cons**

- Theory has to be built into solver
- Implementation harder
SAT Modulo Theory (SMT)

- The major school of algebraic solvers.
- Theories = specifications of (sets of) interpretations.
- SMT-LIB: international standard for SMT solvers.
- Mature implementations:
  - CVC4, Z3, MathSAT, Yices, STP, Boolector, OpenSMT, ...
- Near ubiquitous in software verification.
# Requirements

## Principles

- **Bit-Exact**  Must do *exactly* what the hardware does
- **Precise**  Gives SAT / UNSAT (ideally with model / proof)
- **Automated**  Ideally fast and “out of the box”
- **Flexible**  Support different decision procedures

## Target Applications

- Path feasibility / test-case generation
- Generation of special values
- Numerical instability
- Undefined behaviour
- Hardware verification
- Functional correctness
- Automated numerical analysis
3.2 Specification levels

Floating-point arithmetic is a systematic approximation of real arithmetic, as illustrated in Table 3.1. Floating-point arithmetic can only represent a finite subset of the continuum of real numbers. Consequently, certain properties of real arithmetic, such as associativity of addition, do not always hold for floating-point arithmetic.

<table>
<thead>
<tr>
<th>Level 1</th>
<th>{-∞ ... 0 ... +∞}</th>
<th>Extended real numbers.</th>
</tr>
</thead>
<tbody>
<tr>
<td>many-to-one ↓</td>
<td>rounding</td>
<td>↑ projection (except for NaN)</td>
</tr>
<tr>
<td>Level 2</td>
<td>{-∞ ... -0} U {+0 ... +∞} U NaN</td>
<td>Floating-point data—an algebraically closed system.</td>
</tr>
<tr>
<td>one-to-many ↓</td>
<td>representation specification</td>
<td>↑ many-to-one</td>
</tr>
<tr>
<td>Level 3</td>
<td>(sign, exponent, significand) U {-∞, +∞} U qNaN U sNaN</td>
<td>Representations of floating-point data.</td>
</tr>
<tr>
<td>one-to-many ↓</td>
<td>encoding for representations of floating-point data</td>
<td>↑ many-to-one</td>
</tr>
<tr>
<td>Level 4</td>
<td>0111000...</td>
<td>Bit strings.</td>
</tr>
</tbody>
</table>

The mathematical structure underpinning the arithmetic in this standard is the extended reals, that is, the set of real numbers together with positive and negative infinity. For a given format, the process of rounding (see 4) maps an extended real number to a floating-point number included in that format. A floating-point datum, which can be a signed zero, finite non-zero number, signed infinity, or a NaN (not-a-number), can be mapped to one or more representations of floating-point data in a format.
Level 1: Extended Reals

\[ \mathbb{R}^* = \mathbb{R} \cup \{ +\infty, -\infty, \text{NaN} \} \]

(partially ordered, additive and multiplicative commutative monoid with the distributivity property)

\[
\begin{align*}
    u + \text{NaN} &= \text{NaN} + u = \text{NaN} & +\infty \leq w & \iff w = +\infty \\
    -\text{NaN} &= \text{NaN} & w \leq -\infty & \iff w = -\infty \\
    u \cdot \text{NaN} &= \text{NaN} \cdot u = \text{NaN} & -(+\infty) &= -\infty \\
    \text{NaN}^{-1} &= \text{NaN} & -(-\infty) &= +\infty \\
    \text{NaN} \leq u & \iff u = \text{NaN} & +\infty^{-1} &= 0 \\
    u \leq \text{NaN} & \iff u = \text{NaN} & -\infty^{-1} &= 0 \\
    & & 0^{-1} &= +\infty
\end{align*}
\]

\ldots
Level 2(ish) : Domain

\[
\begin{align*}
F_{\epsilon,\sigma} &= \{\text{NaN}\} \\
F_{\epsilon,\sigma} &= FZ_{\epsilon,\sigma} \cup FS_{\epsilon,\sigma} \cup FN_{\epsilon,\sigma} \cup FI_{\epsilon,\sigma} \\
FZ_{\epsilon,\sigma} &= \{(s, e, m) \in B_{\epsilon,\sigma} \mid e = 0_{\epsilon}, m = 0_{\sigma-1}\} \\
FS_{\epsilon,\sigma} &= \{(s, e, m) \in B_{\epsilon,\sigma} \mid e = 0_{\epsilon}, m \neq 0_{\sigma-1}\} \\
FN_{\epsilon,\sigma} &= \{(s, e, m) \in B_{\epsilon,\sigma} \mid e \neq 1_{\epsilon}, e \neq 0_{\epsilon}\} \\
FI_{\epsilon,\sigma} &= \{(s, e, m) \in B_{\epsilon,\sigma} \mid e = 1_{\epsilon}, m = 0_{\sigma-1}\}
\end{align*}
\]

\[v_{\epsilon,\sigma} : F_{\epsilon,\sigma} \rightarrow \mathbb{R}^*\]
5. Operations

5.1 Overview

All conforming implementations of this standard shall provide the operations listed in this clause for all supported arithmetic formats, except as stated below. 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 (see 4 and 7). Clause 6 augments the following specifications to cover ±0, ±∞, and NaN. Clause 7 describes default exception handling.
Upper and Lower Adjoint
Upper and Lower Adjoints
Upper and Lower Adjoints

\[ +\infty \]

\[ -\infty \]

\[ +0 \]

\[ -0 \]

\[ +Inf \]

\[ -Inf \]
Rounding is (Just) Selecting Between Adjoints!

\[ \text{rnd}(v, \text{mode}, sz, r) = \overline{v}(r) \text{ or } \underline{v}(r) \]

This allows us to round to any format of float, bit-vectors, \( \mathbb{Z} \), integer valued floats, ...
add_{\epsilon,\sigma}(\text{rm}, f, g) = \text{rnd}(v, \text{rm}, \text{addSign}(\text{rm}, f, g), v(f) + v(g))

sub_{\epsilon,\sigma}(\text{rm}, f, g) = \text{rnd}(v, \text{rm}, \text{subSign}(\text{rm}, f, g), v(f) - v(g))

mul_{\epsilon,\sigma}(\text{rm}, f, g) = \text{rnd}(v, \text{rm}, \text{xorSign}(f, g), v(f) \times v(g))

div_{\epsilon,\sigma}(\text{rm}, f, g) =
\begin{cases} 
\text{neg}_{\epsilon,\sigma}(\text{rnd}(v, \text{rm}, \top, -(v(f)/v(g)))) & \text{xorSign}(f, g) \\
\text{rnd}(v, \text{rm}, \bot, v(f)/v(g)) & \neg\text{xorSign}(f, g)
\end{cases}

fma_{\epsilon,\sigma}(\text{rm}, f, g, h) = \text{rnd}(v, \text{rm}, \text{fmaSign}(\text{rm}, f, g, h), (v(f) \times v(g)) + v(h))
Limitations and Omissions

- No decimal floats
- Only one NaN (no signaling / quiet, no payload)
- No exceptions
- No attributes
- No trigonometric functions
# Implementations

<table>
<thead>
<tr>
<th></th>
<th>Bit-blast</th>
<th>ACDL</th>
<th>Axiomatic</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVC4</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Z3</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>MathSAT</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Sonolar</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Alt-Ergo</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CBMC</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SMT

SMT-LIB Theory of Floating-Point

Conclusions
Help Needed!

- Correctness
- Examples
  - (edge cases, tests, challenge problems)
- “Diamond free” circuits
  - (multiply, divide, shift, float add, normalise)
- Elementary functions
- Floating-point remainder
Conclusions

1. Formalisation as input (axiomatic) vs. formalisation as specification (algebraic)
2. Rounding as choice of adjoints.
3. Have a specification (and implementations) of an SMT-LIB standard of floating-point.
Conclusions

1. Formalisation as input (axiomatic) vs. formalisation as specification (algebraic)
2. Rounding as choice of adjoints.
3. Have a specification (and implementations) of an SMT-LIB standard of floating-point.

Thank you for your time and attention.
Made using only Free Software