Bit-Exact Automated Reasoning About Floating-Point Arithmetic

Martin Brain

University of Oxford

July 4, 2018
What Do You Want To Know?
int takeNSamples (unsigned int n, float start, float end) {

if (((n == 0) || (end - start <= 0)))
    return ERROR;

float increment = (end - start) / (n + 1.0f);
float location = start + increment;

while (!location >= end)) {
    sample(location);
    location += increment;
}

return DONE;
}
Want To Use Program Analysis Tools ... 

... So Need An SMT Solver That Is ...

- **Bit-Exact**  Must do *exactly* what the hardware does
- **Precise**    Gives SAT / UNSAT
- **Model Generating** Gives a model if SAT
- **Automated**  Ideally fast and “out of the box”
Other Use Cases

✓ Path feasibility / test-case generation
✓ Generation of special values
✓ Numerical instability
✓ (Language-level) Undefined behaviour
✓ Verification of hardware floating-point units
✓ Something about neural nets...
? Functional correctness of numerical code
? Automated numerical analysis
Let’s Build a Number System!

But what about over-flow?

OK, but what is $\infty - \infty$?

That’s cool but what about underflow?

Hold on; this can’t be algebraically closed?

Yeah but what are the actual values?
Let’s Build a Number System!

But what about over-flow?
Let’s Build a Number System!

But what about over-flow?  OK, but what is $\infty - \infty$?
Let’s Build a Number System!

But what about over-flow?  OK, but what is $\infty - \infty$?  That’s cool but what about underflow?
Let’s Build a Number System!

But what about over-flow? OK, but what is $\infty - \infty$? That’s cool but what about underflow? Hold on; this can’t be algebraically closed?
But what about over-flow? OK, but what is $\infty - \infty$?
That’s cool but what about underflow?
Hold on; this can’t be algebraically closed?
Yeah but what are the actual values?
Representing Rationals As...

\[-1^{\text{sign}} \times \text{base}^{\text{exponent}} \times \text{significand}\]

\[7.28125 = (2^2 + 2^1 + 2^0) + (2^{-2} + 2^{-5})\]
\[= 111.01001_2\]
\[= -1^0 \times 2^2 \times 1.1101001_2\]
Constant Relative Error?

For normal $f$ is $\frac{f - \text{prev}(f)}{f} \leq c$ ?
Why is Reasoning About Floating-Point Hard?

**Combinatorics**

\[ a + b \]

5 rounding modes
- \( a \) is normal OR subnormal OR zero OR infinite OR NaN
- \( b \) is normal OR subnormal OR zero OR infinite OR NaN

\[ \Rightarrow \] 125 cases

**Rounding**

- \((a + b) + c \neq a + (b + c)\)
- \((a * b) * c \neq a * (b * c)\)
- \(a * (b + c) \neq a * b + a * c\)
Why is Reasoning About Floating-Point Software Hard?

Ranges Used vs. Ranges Specified

<table>
<thead>
<tr>
<th></th>
<th>Range</th>
<th>Data Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Likely speed in m/s</td>
<td>$[1 \times 10^{-15}, 3 \times 10^8]$</td>
<td>16</td>
</tr>
<tr>
<td>Positive range of double</td>
<td>$[1 \times 10^{-324}, 1 \times 10^{-308}]$</td>
<td>63</td>
</tr>
</tbody>
</table>

$[-1, 1]$ is half of all floats!

Specifications are “obvious”

“Obviously it doesn’t work for . . . but it doesn’t need to.”
“It won’t ever be called with . . . anyway.”
Standards

How standards proliferate:

(See: A/C chargers, character encodings, instant messaging, etc)

SITUATION: There are 14 competing standards.

14?! Ridiculous! We need to develop one universal standard that covers everyone’s use cases. Yeah!

SOON:

SITUATION: There are 15 competing standards.
Standards

ISO-9899

Uses

IEEE-754

Formalises

SMT-LIB

Implements

ARM / x86
The SMT-LIB Initiative

http://smt-lib.org

- International initiative
- Rigorously standardise descriptions of theories for SMT
  Arithmetic ($\mathbb{Z}$ and $\mathbb{R}$), arrays, bit-vectors, floating-point, data-types
  (in preparation strings, sets, separation logic ...)
- Promote common syntax for SMT interactions
- Benchmarks
- Annual competition
How To Specify an SMT-LIB Theory

Fix a signature $\Sigma$ ...

- In the theory files on the website
  http://smt-lib.org/theories-FloatingPoint.shtml
- Files have a formal(ish) syntax and semantics
- Intended as a specification for users / solvers

... and its interpretation $(D, I)$

- Given as (human readable) maths
- Most theories it is in the :values $(D)$ and :definition $(I)$
  sections of the theory file.
- For floating-point it is a separate document
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 3.1—Relationships between different specification levels for a particular format

<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} ∪ {+0 ... +∞} ∪ 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>(\text{sign, exponent, significand}) ∪ {-∞, +∞} ∪ qNaN ∪ 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.
D : Domain of Interpretation

Level 1 \( \mathbb{R}^+ = \mathbb{R} \cup \{+\infty, -\infty\} \)

Level 2 \( F_{\varepsilon,\sigma} = F_{\varepsilon,\sigma} \cup \{\text{NaN}\} \)

Level 3 -

Level 4 (fp #b0 #b11..0 #b10...1)
D : Examples of Constants

Examples

; 1.0f
(fp #b0 #b01111111 #b00000000000000000000000000000000)

; +0
(fp #b0 #b00000000 #b00000000000000000000000000000000)
(_ +zero 8 24)

; NaN
(fp #b1 #b11111111 #b111111111111111111111111111111)
(fp #b0 #b11111111 #b00000000000000000000000000000001)
...
(_ NaN 8 24)
IEEE Std 754-2008
IEEE Standard for Floating-Point Arithmetic

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.
I: Extended Reals

\[ F_{\varepsilon, \sigma} \]

\[ \begin{align*}
+0 & \quad 0 \\
-0 & \quad 0 \\
+\infty & \quad +\infty \\
-\infty & \quad -\infty
\end{align*} \]
I : Extended Reals

\[ \mathbb{R}^\dagger \]

Floating-Point

Decision Procedures

Bit-blasting with SymFPU

Conclusion
I: Extended Reals

\[ \text{Floating-Point} \]

\[ \text{Decision Procedures} \]

\[ \text{Bit-blasting with SymFPU} \]

\[ \text{Conclusion} \]
I : Extended Reals

\[ \text{ rnd}(v, rm, zsign, r) = \overline{v}(r) \lor \text{ rnd}(v, rm, zsign, r) = \underline{v}(r) \]
I: Operations

Σ: Syntax

\[(\text{fp.sub RoundingMode}
\begin{align*}
&\text{(FloatingPoint eb sb)} \\
&\text{(FloatingPoint eb sb)} \\
&\text{(FloatingPoint eb sb)}
\end{align*}
)\]

I: Semantics

\[\text{[fp.sub]}(rm, f, g) = \text{rnd}(v, rm, \text{subSign}(rm, f, g), v(f) - v(g))\]

\[\text{subSign}(rm, f, g) =
\begin{cases}
\text{isNeg}(f) \land \neg\text{isNeg}(g) & rm \neq \text{rtn} \\
\text{isNeg}(f) \lor \neg\text{isNeg}(g) & rm = \text{rtn}
\end{cases}\]

Example

\[(\text{fp.sub rnd end start})\]
I : Relations

Σ : Syntax

(fp.eq (_ FloatingPoint eb sb) (_ FloatingPoint eb sb) Bool)
(fp.leq (_ FloatingPoint eb sb) (_ FloatingPoint eb sb) Bool)
(fp.lt (_ FloatingPoint eb sb) (_ FloatingPoint eb sb) Bool)

I : Semantics

\[ [fp.eq] = \{(f, g) \in F_{\varepsilon, \sigma} \times F_{\varepsilon, \sigma} \mid v(f) = v(g)\} \]
\[ [fp.leq] = \{(f, g) \in F_{\varepsilon, \sigma} \times F_{\varepsilon, \sigma} \mid v(f) \leq v(g)\} \]
\[ [fp.lt] = \{(f, g) \in F_{\varepsilon, \sigma} \times F_{\varepsilon, \sigma} \mid v(f) < v(g)\} \]

Example

(fp.eq start end)
(fp.leq (fp.sub rnd end start) (+zero 8 24))
Putting It All Together

From our program

```c
float increment = (end - start) / (n + 1.0f);
```

In SMT-LIB

```smt
(declare-fun n () (_ BitVec 32))
(declare-fun start () Float32)
(declare-fun end () (_ FloatingPoint 8 24))
(define-fun rnd () RoundingMode RNE)

(define-fun increment () (_ FloatingPoint 8 24)
  (fp.div rnd
    (fp.sub rnd end start)
    (fp.add rnd
      ((_ to_fp_unsigned 8 24) rnd n)
      (fp #b0 #b01111111 #b00000000000000000000000000))))
```
Let’s Have A Break!
How Do We Come Up With Decision Procedure?

What Do We Know About Floating-Point?
How Do We Come Up With Decision Procedure?

What Do We Know About Floating-Point?

- Hardware implementations $\rightarrow$ “Bit-blasting”
- Is weakly monotonic $\rightarrow$ Interval
- Can be formalised $\rightarrow$ Axiomatic
- Fast implementation $\rightarrow$ Numeric
"Bit-Blasting"

Reduce to the Bit-Vector Theory

- Use circuits for a floating-point unit
- Simple to implement on top of a bit-vector solver
- Detailed development, computationally heavy
- Under and over approximation of the problem

Implementations

CBMC, Z3, MathSAT, SONOLAR, CVC4
## Intervals

### Weak Monotonicity

\[ s < 0 \land a < b \Rightarrow a + s \leq b + s \]

So...

\[ x \in [x_l, x_h], \ y \in [y_l, y_h], \]

\[ (fp.add \ rnd \ x \ y) \in \ [(fp.add \ RTN \ x_l, y_l), (fp.add \ RTP \ x_h, y_h)] \]

### Implementations

- **Propagate++**  FPCS, COLIBRI
- **Search++**  MathSAT-ACDCL, iSAT
- **AI**  Fluctuat, Astrée, Polyspace, CodePeer
- **Error bound**  Rosa, Daisy
Axiomatic

Full Eager Axiomatisation (in $\mathbb{R}$ and $\mathbb{Z}$)
- Isabelle, HOL, HOL Light, ACL2, PVS, Coq and Meta-Tarski
- Limited automation in ITP
- Decideability issues

Partial and Lazy Axiomatisation
- Gappa Heuristic instantiation of an axiomatisation
- Gappa / Alt-Ergo Gappa combined with Alt-Ergo
- Why3 Eager partial axiomatisation of float over reals
- Alt-Ergo Lazy partial axiomatisation of float over reals
- KLEE-FP Rewriting based handling of equalities
Numerical

Fast Handling of Complete Assignments
- Hardware FPUs are cheap and fast!
- Can do various “black-box” search techniques
- Can’t show UNSAT

Implementations
- XSAT  Float-valued semi-definite solver
- goSAT  Compile formula to executable code
- JFS   External fuzzing tool
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
</tr>
<tr>
<td>2</td>
<td>Floating-Point</td>
</tr>
<tr>
<td>3</td>
<td>Decision Procedures</td>
</tr>
<tr>
<td>4</td>
<td>Bit-blasting with SymFPU</td>
</tr>
<tr>
<td>5</td>
<td>Conclusion</td>
</tr>
</tbody>
</table>

### Floating-Point
- Floating-Point Arithmetic
- SMT-LIB Theory of Floating-Point

### Decision Procedures
- “Bit-Blasting”
- Intervals
- Axiomatic
- Numerical
SymFPU

https://github.com/martin-cs/symfpu/

- C++ (header only) library of bit-vector encodings / circuits
- Parametric in bit-vector, propositional, rounding mode representations
- Reasonably well optimised (good circuit ≠ good encoding!)
- Correct for at least Float16, Float32, Float64, Float128
Basic FPU Data-path

Unpack

Operate

Round

Pack
Unpacked Formats (Level 2 representation)

Options:

- Flags for NaN, $\infty$, 0, subnormals, ...
- Hidden-bit in significand
- Exponent biased or unbiased
- Handling of subnormals
- Lazy normalisation
- Redundant representations
People say they want to see the encodings . . .
Correctness

IEEE-754

vendor QA

manual inspection

SMT-LIB

handbook of FPA

MPFR

Müller & Paul

SYM-FPU

PyMPF

CBMC

MathSAT

Z3

CVC4 symbolic

CVC4 literals

--check-model

SMT-LIB Benchmarks

PyMPF

Schanda Crafted

NyxBrain Executable

NyxBrain Crafted

Wintersteiger
## Performance

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>AE</th>
<th>Col</th>
<th>CVC4</th>
<th>goSAT</th>
<th>MS</th>
<th>MS-A</th>
<th>SON</th>
<th>Z3</th>
<th>Z3-SF</th>
<th>VBS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CBMC</td>
<td>63.0</td>
<td>55.6</td>
<td>9.3</td>
<td>50.0</td>
<td><strong>64.8</strong></td>
<td>38.9</td>
<td>44.4</td>
<td>46.3</td>
<td>81.5*</td>
<td></td>
</tr>
<tr>
<td>Schanda</td>
<td>82.8</td>
<td><strong>85.4</strong></td>
<td>1.5</td>
<td>66.7*</td>
<td>28.3*</td>
<td>82.3</td>
<td>81.8</td>
<td>95.5*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Griggio</td>
<td>0.9*</td>
<td>60.7</td>
<td>59.8</td>
<td>41.1</td>
<td>49.1</td>
<td><strong>68.2</strong></td>
<td>60.7</td>
<td>33.6</td>
<td>34.1</td>
<td>88.3*</td>
</tr>
<tr>
<td>Heizmann</td>
<td>14.0</td>
<td><strong>76.3</strong></td>
<td>58.5</td>
<td>27.5</td>
<td>2.9</td>
<td>40.6</td>
<td>34.3</td>
<td>89.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Industrial 1</td>
<td>68.6*</td>
<td><strong>90.8</strong></td>
<td>65.4</td>
<td>62.5</td>
<td>91.3*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Industrial 1 (QF)</td>
<td>72.0</td>
<td><strong>99.2</strong></td>
<td>96.6</td>
<td>85.8</td>
<td>83.4</td>
<td>82.6</td>
<td>99.7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NyxBrain</td>
<td>99.8</td>
<td><strong>99.9</strong></td>
<td>34.2</td>
<td>95.4</td>
<td>95.0</td>
<td>99.2</td>
<td>99.9</td>
<td>99.8</td>
<td>&gt;99.9</td>
<td></td>
</tr>
<tr>
<td>PyMPF</td>
<td>92.2</td>
<td><strong>99.7</strong></td>
<td>0.3</td>
<td>39.3</td>
<td>35.8</td>
<td>98.8</td>
<td>98.4</td>
<td>99.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spark FP</td>
<td>64.1*</td>
<td><strong>84.6</strong></td>
<td>82.1</td>
<td>74.8</td>
<td>88.9*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spark FP (QF)</td>
<td>88.6</td>
<td><strong>94.9</strong></td>
<td>82.7</td>
<td>78.6</td>
<td>88.8</td>
<td>87.9</td>
<td>99.6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wintersteiger</td>
<td>49.9*</td>
<td>✓</td>
<td>✓</td>
<td>13.9</td>
<td>85.8</td>
<td>85.8</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

Percentage of problems solved in each category of benchmarks
1 Introduction

2 Floating-Point
   - Floating-Point Arithmetic
   - SMT-LIB Theory of Floating-Point

3 Decision Procedures
   - “Bit-Blasting”
   - Intervals
   - Axiomatic
   - Numerical

4 Bit-blasting with SymFPU

5 Conclusion
Open Problems

- New approaches
- Mixed real / float decision procedures
- Mixed real / float / bit-vector decision procedures
- Trigonometric functions
- Scalability!
Conclusions

1. Tool support is vital for floating-point in critical software.
2. The theory of floating-point in the SMT-LIB standard.
3. There are “off-the-shelf” solvers.
4. Some interesting challenges.
Conclusions

1. Tool support is vital for floating-point in critical software.
2. The theory of floating-point in the SMT-LIB standard.
3. There are “off-the-shelf” solvers.
4. Some interesting challenges.

Thank you for your time and attention.

Made using only Free Software