Real Semantics: Capturing Floating-Point Imprecision

Hannah Blumberg, Yihe Huang, Dan King, Paola Mariselli —Harvard School of Engineering and Applied Sciences

BACKGROUND

- Floating-point numbers are ubiquitous in computing applications
- Programmers usually treated them just as real numbers but they are really not!

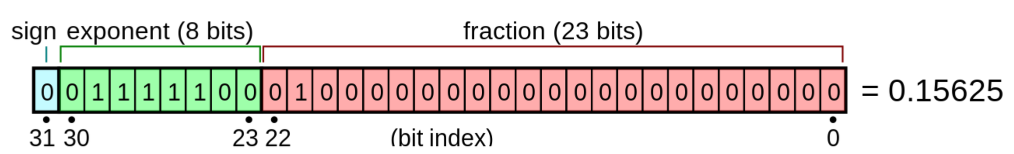

Fig. 1. IEEE-754 single-precision floating-point format (Wikipedia [1]).

- Due to the finite, discrete binary construction of floating-point numbers, simple real numbers like 0.1 in decimal cannot be represented in exact form in floating-point
- Floating-point numbers are more "dense" around zero and are relatively sparse at higher orders of magnitude
- Arithmetic operations over floating-point numbers are not closed, which means an arithmetic operation involving two valid floating-point numbers may end up with a result that doesn't have an exact floating-point representation and therefore rounding is required
- Rounding and cancellation are sources of potentially disastrous imprecision

MOTIVATION

There are existing techniques to analyze or optimize floating-point usage in computer programs, with some limitations:

- Formal verification
  - Good at catching erroneous conditions like div-by-zero exceptions
  - Not sophisticated enough to deal with imprecisions
  - Semantics of floating-point arithmetic are too difficult to model and check formally
- Static analysis tools (e.g. Herbie [2])
  - Can optimize floating-point usage in mathematical expressions
  - Not general enough to enable generic full-program analyses

We decide to approach this problem by dynamic analysis!

IMPLEMENTATION

Real Semantics: a custom LLVM IR interpreter engine that augments floating-point operations with arbitrary precision arithmetic (provided by GNU MPFR).

We choose LLVM IR for generality: any existing program source code that the LLVM front-end can handle can be compiled down to IR.

We introduce a new data type, namely `SmartFloat`, in LLVM IR, which holds two representations for each floating-point value in the program:

- regular precision – native `float` or `double` type
- high precision – `mpfr::real` type

`SmartFloat` is bigger than native floating-point numbers, so we store them in a separate map, keyed by their address in the program's native address space.

We check for precision loss by comparing the two representations in the SmartFloat object. To minimize false positives/excessive error messages, we only report an error when:

- imprecision results in divergence in control
- imprecision results in divergence in external effects (e.g. output, uninterpreted library calls, etc.)

```
class SmartFloat {
private:
  float floatV;
  mpfr::real<PRECISION> realV;
public:
  SmartFloat(float initVal) ...
  bool check_precision() {...}
  SmartFloat operator+(const SmartFloat& rhs) {...}
  SmartFloat operator- ...
  // more operator definitions...
}
```

Fig. 2. Definition of the new `SmartFloat` LLVM IR data type.

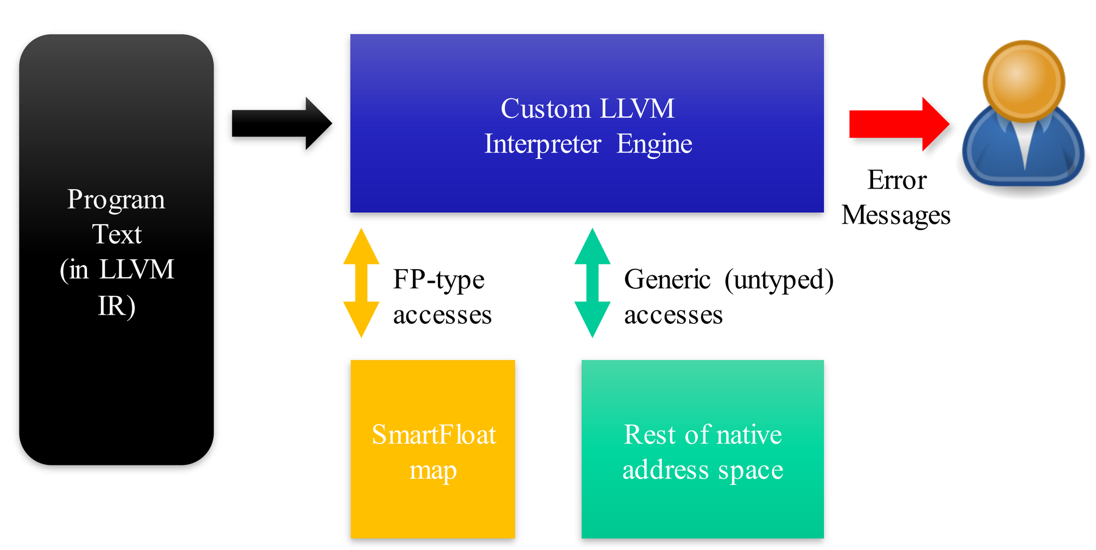

Fig. 3. High-level system architecture.

RESULTS

```
int probAUB(float pa, float pb) {

  float f1 = pa + pb - pa * pb;
  printf("P(A)+P(B)-P(A)P(B)=%.8f\n", f1);

  float f2 = 1 - (1 - pa) * (1 - pb);
  printf("1-(1-P(A))(1-P(B))=%.8f\n", f2);

  return 0;
}
```

```
P(A)+P(B)-P(A)P(B)=0.00000005

Possible precision loss at printf! Our checker is expecting the
output string: 5.02000006e-08, but with floating-point
imprecision the output string is instead: 5.96046448e-08

1-(1-P(A))(1-P(B))=0.00000006
```

Fig. 4. A sample program and its output when running with our tool when we supply `pa = 5e-8` and `pb = 2e-10` as the input. No error messages show up before the first output, which means the `f1` satisfies the precision requirement.

```
#include <math.h>
#include <stdio.h>

int main() {
  float lower = 0.0;
  float upper = M_PI;
  float step = 0.0001;

  float result = 0.0;
  float x;

  for (x = lower; x < upper; x += step) {
    result += sin(x) * step;
  }

  printf("%f\n", result);
  return 0;
}
```

```
Precision loss at < (numeric_int.c:9)
got 1 = 3.14065 < 3.14159
expected 0 = 3.141600e+00 < 3.141593e+00
...
Possible precision loss at printf! Our checker is expecting the
output string: 2.000000, but with floating-point imprecision
the output string is instead: 2.000405

2.000405
```

Fig. 5. Another sample program calculating an integral and its output when running with our tool. Output shows how our tool catches and reports control flow divergence.

CONCLUSIONS

- Real Semantics is a custom LLVM IR interpreter engine that can dynamically find floating-point imprecision errors in computer programs.
- It augments regular floating-point operations with arbitrary precision arithmetic, and it checks for precision loss by comparing the two representations of the same floating-point value.
- For better usability, we only report errors when imprecision results in divergent program control flow or external effects.

Future Work

- Improve performance by using just-in-time compilation or instrumentation
  - The interpreter alone incurs a 200-1000x slowdown
- Better precision measurement by using "real" arbitrary precision arithmetic libraries (like those used by computer algebraic systems) in place of MPFR

REFERENCES

[1] Wikipedia User Stannered. Example of a floating point number. Retrieved from https://en.wikipedia.org/wiki/Single-precision_floating-point_format#/media/File:Float_example.svg. January 8, 2015.

[2] P. Panchekha, J. R. Wilcox, and Z. Tatlock. Automatically improving accuracy for floating point expressions. 2015.

ACKNOWLEDGEMENTS

This project was part of CS260r (Topics and Close Readings in Computer Systems) at Harvard, Spring 2015. We would like to thank our professor Eddie Kohler for his kind support and guidance.

Check out source code at https://github.com/danking/real-semantics