Real Semantics: Capturing Floating-Point Imprecision

Hannah Blumberg Yihe Huang Dan King Paola Mariselli

School of Engineering and Applied Sciences, Harvard University

hannahblumberg@college.harvard.edu yihehuang@g.harvard.edu
daniel.zidan.king@gmail.com paolamariselli@fas.harvard.edu

Abstract

Floating-point numbers and their respective arithmetic operations are often used with the assumption of a certain level of precision. However, there are times when exact precision is necessary. In these cases, floating-point arithmetic is defective in that it is fundamentally interval arithmetic wherein the interval lengths are a function of the median point.

Therefore, we present Real Semantics, an automated tool that reasons about the accuracy of floating-point arithmetic. Real Semantics systematically discovers instances where the floating-point arithmetic result is not the most accurate floating-point approximation to the real number implied by any given algorithm. Thereafter, Real Semantics shows the user where the discrepancy is taking place in the form of a line number and variable name.

Keywords Floating-point numbers. Floating-point arithmetic. Precision errors.

1. Introduction

1.1 Motivation

It is impossible to represent infinitely many real numbers using finitely many bits. As a result, most computer systems that aim to work with real numbers use the floating-point system to represent real numbers.

Despite the frequency with which floating-point numbers and their associated operations are used, floating-point numbers have many confusing properties that make it difficult to use them correctly. For example, floating-point operations are not closed and are generally not associative. That is, a binary operation that takes two floating-point values may not result in a value that can be exactly represented in the floating-point system. Consequently, the value must be rounded or truncated. Precision loss due to rounding or truncation causes operations to produce different results when applied to the floating-point values in different orders [10]. In most applications, the precision loss due to sloppy programming is insignificant and does not impose a threat to correctness. However, because of the silent nature of these kind of errors, small errors in floating-point numbers can accumulate and cause more severe problems.

One such severe problem occurring as a result of floating-point imprecision is the Ariane 5 rocket explosion. The $500 million rocket owned by the European Space Agency was destroyed in 1996 due to a floating-point error. After the rocket launched, one of the systems produced a 64-bit floating-point number which was then sent to the on-board system. When the on-board system converted it into a 16-bit integer, an overflow error occurred and caused the main system to shut down. Since this error was unexpected, no coding policies had been implemented to safeguard against it [14]. This resulted in the system interpreting the change as a course change. This veered the rocket off course and caused a major disaster.

Floating-point imprecision can also cause other major problems that cost businesses significant financial resources. Variances in digits when calculating floating-point arithmetic can cause numbers to be rounded the wrong direction. This in turn can cause numeric issues, which lead to errors in financial calculations, particularly when these small errors add up. Although most financial organizations take heed on this issue and use other data representations instead, every so often floating-point errors re-surface. For example, with the introduction of the EURO, when old local currencies were converted to the EURO or to other local currencies, conversion, reconversion, and totalising errors occurred [14].

1.2 Contributions

Given that floating-point errors are so important to programmers and the products they create, we built Real Semantics, a dynamic floating-point imprecision detector based on the LLVM interpreter. This tool captures significant floating-point imprecision and reports it to the user. By doing this, we expose otherwise silent floating-point precision errors that may eventually cause significant, yet difficult-to-detect bugs.

Thus, our main contribution is a modified LLVM interpreter that:

- tracks higher-precision representations of the numbers calculated by the interpreted program,
- notifies the user of floating-point imprecision *only when* it substantially changes the behavior of the program, and
- identifies precision loss events by operation, variable name, and/or location (file name and line number).

2. Related Work

2.1 Static Analysis

Barr et al. [7] built Ariadne. Ariadne applies symbolic execution to statically detect floating-point related bugs. Ariadne statically discovers occurrences of underflow, overflow, and arguments outside a function's domain, including division-by-zero. Ariadne converts C, C++, and Fortran programs to a real number companion program that makes explicit the possibility of floating-point errors. Ariadne discovers real number constraints that trigger the explicit floating-point errors using symbolic execution, an SMT solver, and a custom algorithm for solving non-linear constraints. Finally, Ariadne converts the real number constraints to floating-point numbers that witness the bugs.

Ariadne does not require test inputs nor does it actually execute the program - these are two improvements over our contribution, Real Semantics. However, Ariadne does not detect accumulated floating-point imprecision that modifies program behavior relative to the real number values while Real Semantics does. Moreover, Ariadne requires sophisticated tools and techniques to create and solve real number constraints.

2.2 Formal Verification

Boldo and Filliatre [8] present a formal approach using Coq to verify floating-point programs. In contrast to Real Semantics and Ariadne, Boldo, et al. attempt to prove the non-existence of floating-point bugs. Moreover, they prove error bounds on the outputs of floating-point valued functions. Their work does not support the verification of programs using transcendental functions, such as sine and cosine.

Very recently, Boldo et al. [9] extended the CompCert compiler with a verified correct translation of IEEE-754 floating-point semantics into hardware. This is an important step towards verified correct floating-point computation because we can now trust that correct floating-point implementations in C will be compiled to correct machine code.

A major component of the work of Boldo et al. [9] was a formally specified semantics of IEEE-754 floating-point arithmetic. Interestingly, there did not previously exist such a rigorous, machine digestible definition of the IEEE standard.

Verification is an attractive alternative for critical systems, however, current methods are impractical for large programs owing both to the reasoning complexity and the necessary human effort.

2.3 Precise By Construction

Panchekha et al. [13] developed Herbie. Herbie is a tool for automatically improving the accuracy of floating-point programs. The user specifies a numeric expression in a simple s-expression language. Herbie then performs rounds of testing random floating-point values and applying a table of known mathematical identities. Finally, Herbie outputs an expression that is equivalent on the reals, but more precise on the floating-point numbers.

Of the programs generated by Herbie, after conversion to C and compilation with GCC, the median program is 40% slower than the unadulterated programs. However, the generated programs are impressively more accurate; often achieving nearly maximum accuracy for the given floating-point format. Unfortunately, Herbie is necessarily limited to purely mathematical expressions. It cannot comprehend nor improve full programs in Turing-complete languages. Our tool, Real Semantics, complements Herbie by following the propagation of floating-point imprecision through imperative code.

3. Design

3.1 Floating-point Numbers

Although our implementation was a relatively complex set of changes to the LLVM interpreter [2], our design is exceedingly simple.

Floating-point numbers are represented as a significand (also known as the mantissa), representing the significant digits, and an exponent, representing the magnitude of the number. Each floating-point number represents an uncountably infinite set of real numbers. As such, the set of reals represented by a floating-point number

$$f = d.ddd...dd \times b^e$$

is defined as the real numbers falling in the ball centered at f with radius:

$$0.000...00d\prime \times b^e$$

where $d\prime$ is the digit $b/2$, assuming an even base. If floating-point numbers actually used interval arithmetic, then multiplication and addition would lead to ever increasing intervals. In the worst case, the interval could be large enough to make each digit in the significand meaningless.

Our tool aims to use much higher precision numbers so that the intervals stay much smaller than the regular precision numbers. We can then report the discrepancies between these numbers.

3.2 Augmented Floating-point Semantics

Floating-point imprecision can occur after any floating-point operation, depending on the magnitude and value of the arguments. We consider two sets of floating-point operations. Those that are floating-point valued and those that are not:

$$\begin{aligned} \oplus &::= / \mid - \mid + \mid * \mid sqrt \mid abs \mid exp \mid cos \\ &\mid sin \mid tan \mid atan \mid atan2 \mid \cdots \\ \sqsubseteq &::= < \mid \leq \mid > \mid \geq \mid = \mid \cdots \end{aligned}$$

Many of these operations have non-trivial relationships between input error and output error. We side-step all these issues by simply accompanying every floating-point number by a higher-precision partner.

We perform a conceptually very simple program transformation. We replace every literal floating-point number with a pair of that literal represented in both floating-point precision as well as higher precision. Since both values are numbers and the product operator is a Functor, we can lift all numeric operations point-wise. We denote the higher precision representation of a number x as $\hat{x}$

$$\begin{aligned} x &\mapsto (x, \hat{x}) \\ (x, \hat{x}) \oplus (y, \hat{y}) &\mapsto (x \oplus y, \hat{x} \oplus \hat{y}) \end{aligned}$$

Floating-point numbers interact with the other types of the C programming language at three specific points:

- conversion to integral-typed numbers,
- comparison operations, and
- output operations (such as `printf`).

We could simply make our interpreter an Abstract Interpretation and use approximations for other types, such as sets of Booleans and integral intervals. This approach, however, did not seem particularly useful nor practical.

Real Semantics inserts floating-point precision checks before these sorts of operations. We do not halt the execution, as this would prevent the discovery of further floating-point imprecision errors. Instead, we report the precision loss to the user. Again, we use

$$
\begin{aligned}
(INTTYPE)f &\mapsto \mathit{checkPrecisionAndReport}(f);(INTTYPE)f \\
(x,\hat{x}) \sqsubseteq (y,\hat{y}) &\mapsto \texttt{if}\ ((\hat{x} \sqsubseteq \hat{y})\ \texttt{!=}\ (x \sqsubseteq y))\ \texttt{then}\ \mathit{report}\ \texttt{fi}\ ;(x \sqsubseteq y) \\
\texttt{printf}\ (\mathit{fmt},(x,\hat{x})) &\mapsto \texttt{if strcmp}\ (\ \texttt{sprintf}\ (\mathit{fmt},x),\ \texttt{sprintf}\ (\mathit{fmt},\hat{x}))\ \texttt{!=}\ 0 \\
&\ \texttt{then}\ \mathit{report}\ \texttt{fi}\ ;\ \texttt{printf}\ (\mathit{fmt},(x,\hat{x}))
\end{aligned}
$$

Figure 1. The program transformation that inserts precision checks.

a very conceptually simple program transformation depicted in Figure 1.

The *checkPrecisionAndReport* function converts the high precision representation to the nearest low-precision number and checks for equality against the low-precision number. If the numbers are not equal then it reports the values, the variable name, the line number, and the file name based on their availability.

4. Implementation

We present a modified version of the LLVM interpreter that catches floating-point imprecision, which interprets program in LLVM intermediate representation (IR) [3]. We choose to work at the IR level in order to increase the power and flexibility of our tool. IR provides a version of the original source code that is both machine- and language-independent. As a result, our tool can be used to check any strongly-typed program that can be compiled to bitcode format. Furthermore, the IR is much a simpler language than most source code languages we wish to analyze, but does not lose any of the information necessary for accurate analysis.

4.1 Real Number Computations

Initially, we considered using a C++ package called RealLib for our real number computations [12]. However, we soon found that this library was not exactly ready for use out-of-the-box. For example, the following code snippet output "Infinity".

```
RealLib::Real sum("0");

std::cout << std::setprecision(15) << sum <<
std::endl;
```

Since RealLib seemed unable to correctly represent zero and is not well-supported or discussed online, we chose to use the GNU MPFR library instead. MPFR is a well-supported C library that can be used for multiple-precision floating-point computations with correct rounding [5].

MPFR correctly handles the situation mentioned above. The only shortcoming is that MPFR requires that we know a priori how much precision we will need to compute a particular answer. In general, the intermediate precision necessary to correctly compute an answer can be arbitrarily higher than the precision necessary to store the final result. For example, consider alternating between adding 0.1 and 10000000 ten million times. The final result, will use only a couple bits, as it is a multiple of ten, however, the intermediate values need to correctly store 10000000.1 which requires a greater number of bits.

Nevertheless, for our purposes, MPFR was a better solution for real number computations than RealLib.

4.2 SmartFloat

Our tool is able to identify floating-point imprecision by attaching extra information to every `float` or `double` that is used in the source code. In particular, when a value of type `float` or `double` is created, we replace it with a `SmartFloat`. Each `SmartFloat` contains the original `float` or `double` value along with a "real" representation of the value. The real representation uses MPFR to achieve 4,096 bits of precision.

The `SmartFloat` is then stored in a global map. Whenever a `float` or `double` is loaded from memory, the map is traversed to find the appropriate `SmartFloat`.

At a high level, operations that are performed on the original `float` or `double` value are now performed twice: once on the `float` or `double` value with typical floating-point semantics and once on the "real" representation of the value with higher precision. This is described in detail in section 4.4.

4.3 Types in LLVM IR

The LLVM IR has a type system that is very similar to C and C++'s. Because of the useful type information at the IR level, the interpreter understands the type of the data being operated on. For example, when passing arguments to a function, the interpreter usually knows the exact type of each argument, unless it is explicitly casted to untyped bytes (or a pointer to untyped bytes) in the higher level language. Our modified interpreter provides additional semantics for data of `float` or `double` type by maintaining internal `SmartFloat` objects. LLVM IR program constructs that operate on data of floating-point types will hence operate on these `SmartFloat` objects instead.

Note that the newly introduced `SmartFloat` type is not an extension to the LLVM IR. We are only introducing `SmartFloat` objects when interpreting programs in LLVM IR.

4.4 Supported Program Constructs

Since our tool operates at the LLVM IR level, we extend the implementations of certain LLVM IR constructs in the LLVM interpreter to provide additional semantics. We will describe the changes made into these constructs here in detail.

4.4.1 Floating-Point Binary Operations

Floating-point binary operation instructions, or BINOPs, are floating-point operations that take two operands. In many programs, most floating-point instructions are BINOPs, such as add, subtract, and multiply. The original LLVM interpreter has a macro that handles these instructions, and we modify that macro such that it performs the requested BINOP on both the native floating-point value and the "real" value in MPFR. As mentioned above, both the floating-point value and the "real" value reside in a `SmartFloat` object managed by the interpreter.

4.4.2 Floating-Point Conversions

Floating-point conversion instructions are floating-point truncation and extension operations that convert a floating-point value to either a different floating-point format or an integer. For conversions between floating-point numbers, we keep the "real" value in our SmartFloat object unchanged and perform a type cast on the floating-point value as required. Our aim is to be careful about conversions between floating-point values and integers because the two representations of the same value in a SmartFloat object could be rounded to different integers due to floating-point imprecision. For this reason, we conduct a precision check before the conversion. See section 4.5 for more details.

4.4.3 Function Calls

Function calls are the most complicated program constructs we must support in a usable system. For ease of implementation and also performance reasons, we categorize function calls in a program into three classes and interpret them separately.

The first class of functions are called *intercepted* functions because they are not actual function calls (i.e. an unconditional jump followed by stack allocations) but are instead intercepted and short-circuited within the interpreter. We initially referred to them as math library functions, however, upon further examination, we discovered we needed to intercept more functions than just math library functions in order to make the tool more usable. Despite the change in terminology, though, most functions in this class are indeed math library functions. When a program calls a math library function like `sin`, the interpreter will perform the requested math operation on both the floating-point value and the real value - using library functions from both the C math library and the MPFR math library.

We also intercept the `printf` function due to its unique semantics (e.g. depending on the format string, the actual output of `printf` may be the same even if different floating-point values are passed to it). See section 4.5 for more details.

The second class of functions are *external* functions. An LLVM IR program may contain references to functions provided by external libraries that are not available in LLVM bitcode format. For the tool to be useful, these functions have to be properly handled because otherwise our tool will not have been compatible with any program that conducts any type of system calls, such as writing a file or printing to `stdout`. Therefore, Real Semantics follows a solution the original LLVM interpreter adopts which is to use a library called Foreign Function Interface (FFI) [11] to solve this problem.

Since FFI has no knowledge about the SmartFloat object our tool uses under the hood, we have to prepare arguments of FFI calls properly so that all floating-point values are downgraded to native floating-point formats. Our interpreter will inspect the type information of the argument list at the call site and perform conversions when necessary. Again, floating-point imprecision can potentially affect the behavior of external functions. See section 4.5 for more details.

The rest of the functions are *internal* functions declared and defined within the LLVM IR program. These functions can handle SmartFloat objects correctly because they are completely interpreted by the interpreter.

Another issue that comes up with supporting function calls is memory management. Memory allocated on stack shall be freed and reused after the function returns, but since we maintain a SmartFloat map (see section 4.2) that is separate from the actual memory, it is hard to track all the floating-point values being allocated and freed during a function call. Our reliance on type information also causes some issues when we try to handle `malloc` and `free` correctly.

Consequently, we choose an approach that strikes a good balance between simplicity and completeness: we do not actively free SmartFloat objects that represent floating-point values allocated on stack, but we erase them from the map when we detect that non-floating-point values have been written to overlapping memory regions. This "on-demand" garbage collection scheme does assume that the original program does not contain undefined behaviors like reading from unallocated or uninitialized memory.

4.4.4 Untyped Memory Access to Floating-point Values

Untyped memory accesses bypass our special machinery handling floating-point numbers in the interpreter because of the lack of type information. Most of these accesses take the form of calling the `memcpy` function or methods alike that move bytes around without understanding the content of the data being moved. When floating-point values stored in memory are being accessed in this way, the interpreter cannot construct or keep track of SmartFloat objects correctly. We originally planned not to address this problem as we did not expect untyped floating-point accesses to be common. However, we later discovered that due to compiler optimizations, `memcpy` calls are often inserted by the compiler to initialize global variables or large arrays. Due to the ubiquity of `memcpy` usage in programs, we decided that we had to address this untyped access problem - at the very least at a superficial level.

Thus, we use a workaround that also assumes the correctness of the program being interpreted. Whenever we do a typed "load" of a floating-point value from memory, we will first try to find it in the SmartFloat map. If the memory address requested is not found in the map, the interpreter will load a value as a floating-point directly from the memory address supplied, construct a fresh SmartFloat object out of this value, and insert it to the map. Note that we are only doing this trick when we do a typed load. We consider it to be sufficient because before anything interesting (like a BINOP) happens, the floating-point value must be loaded with its type information so that the execution engine, let it be the interpreter or the actual hardware, can operate on it using the correct semantics.

4.5 Detecting Errors

Throughout the program's execution, the two values stored in each SmartFloat are compared in order to determine if there was a loss of precision using the typical floating-point semantics.

In order to compare the two values, the "real" representation is converted into the closest possible floating-point representation. If the two are not equal, our tool reports the loss of precision. The programmer is told the expected and actual value, along with the variable name and line number if available.

Although the current implementation of our tool requires that the floating-point numbers used are the closest possible representations of the real values, we could easily introduce an adjustable threshold that would allow the programmer to set an "acceptable" range of imprecision. If the floating-point representation is within an "acceptable" range, we would not display any error messages.

The aforementioned comparisons are made when

- an external function is called with a SmartFloat as an argument,
- a comparison operation involving a SmartFloat occurs, or
- a conversion from a `float` or `double` to an integral type takes place.

We need to check for a loss of precision before passing a `SmartFloat` to an external function because we cannot perform the "real" computations that correspond to the potential floating-point computations that occur within the external function.

There are a few exceptions to this rule. First, we intercept several functions including 46 of the 58 `cmath` library functions and perform the appropriate "real" computations alongside the floating-point computations (see section 4.4.3 for more details). As a result, we do not check for precision loss before passing a `SmartFloat` to one of these intercepted `cmath` functions.

A second special case is `printf`. We aim to report a loss of precision in a printed floating-point value only if it will affect the string printed. If the programmer uses `printf` to print a `float` or `double` value, we make a best-effort attempt to compare output string produced using the `float` or `double` value with the output string produced using the "real" representation of the value. It is not an exact comparison given the complexity of the LLVM interpreter external function code (i.e. using `sprintf` is not straightforward). If the best effort comparison determines that the strings are different, a loss of precision is reported.

Additionally, when a comparison operation involving `SmartFloat` occurs, we perform the comparison twice: once using the `float` or `double` value(s) and once using the "real" representation of the value(s). If the Boolean results of these two comparisons are not equal, we consider the loss of precision significant and display an error.

Finally, we check for a loss of precision when performing a conversion from a `float` or `double` to an integral type because we will no longer maintain the "real" representation of the value if it is not a `float` or `double`.

5. Evaluation

In order to evaluate Real Semantics, we first describe our performance on a particular program, then we present a number of case studies that demonstrate a few possible use cases of our tool. For the case studies, we chose programs that are significantly smaller than those found in production code both because they allow us to focus on the use of our tool rather than the intricacies of the programs and because we did not have the computational resources required to test longer programs. The latter issue will be further addressed in section 6.

Nevertheless, one could easily imagine how the relatively small computations and algorithms described here might be used in the context of a larger program; the instances of floating-point imprecision would still exist and may have more significant consequences.

5.1 Ray Tracer

In order to get a sense of the performance of our tool, we considered a ray tracing program. Ray tracing is technique used to generate images by tracing the expected path of light through pixels in a plane. We selected a basic program that creates a simple image, shown in Figure 2 [6].

When executed natively, this program takes approximately 0.629 seconds to run. When executed using the (unmodified) LLVM interpreter, it takes approximately 12 minutes and 20.860 seconds to execute. Using Real Semantics (the modified LLVM interpreter), it takes approximately 16 minutes and 10.675 seconds to execute.

It is clear that the major limitation is not our modifications to the LLVM interpreter, but the interpreter itself. We discuss possible solutions to this performance issue in the section 6.

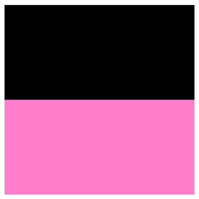

Figure 2. Ray Tracer Output

```
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;
}
```

Figure 3. Calculating $P(A \cup B)$

5.2 Probability

A simple program that demonstrates the impact of floating-point semantics is one that calculates the probability that at least one of two independent events occur, $P(A \cup B)$. Consider the following two mathematically equivalent calculations:

1. $P(A \cup B) = P(A) + P(B) - P(A)P(B)$
2. $P(A \cup B) = 1 - (1 - P(A))(1 - P(B))$

Regardless of the probabilities $P(A)$ and $P(B)$, these two formulas will produce the same result using real number semantics; however, if $P(A)$ are small, the first formula will provide a much more accurate result. This is because very small floating-point values (e.g. $P(A)P(B)$) can be more accurately represented than floating-point values that are close to 1 (e.g. $(1 - P(A))$ and $(1 - P(B))$).

Figure 5 shows a simple function that calculates $P(A \cup B)$ given $P(A)$ and $P(B)$ using both formulas discussed above.

If we were to run this function with the input pa $=$ 5E-8 and pb $=$ 2E-10, our tool would print the following:

```
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
```

Whereas the first formula computed $P(A \cup B)$ without a notable loss in precision, the error message informs us that we lost precision when computing $P(A \cup B)$ using the second formula. Again, we expect these two computations to be equivalent, but they produce different results under floating-point semantics when the probabilities $P(A)$ and $P(B)$ are small.

Our tool correctly catches the loss of precision caused by the second formula and reports it to the user.

5.3 Matrix Inversion

Calculating the inverse of a matrix can be used to solve linear systems. Before calculating the inverse, the programmer should first

```
1   int main()
2   {
3     float a[25][25], k, d;
4
5     k = 3;
6     a[0][0] = 3; a[0][1] = 5; a[0][2] = 2;
7     a[1][0] = 1; a[1][1] = 5; a[1][2] = 8;
8     a[2][0] = 6.6; a[2][1] = 11;
9     a[2][2] = 4.4;
10
11    d = determinant(a, k);
12    if (d == 0)
13      printf("not␣possible\n");
14    else
15      cofactor(a, k);
16    return 0;
17  }
```

Figure 4. Code Snippet of a Matrix Inversion Program

verify that the inverse of a matrix exists by checking whether the *determinant* of the matrix is zero. However, due to floating-point imprecision, simple equality checks like `det == 0` are likely to always return false. Figure 4 shows the main routine of a matrix inversion program retrieved from the Internet [4]. Lines 6 through 9 show the input matrix, whose determinant should be zero in real domain. Due to floating-point imprecision, however, numbers like 6.6 and 4.4 cannot be represented precisely and will be rounded. This causes the check on line 12 to return false; thus, the program will continue with the calculation and end up with a wildly inaccurate result.

Running this program with our tool, however, reveals this problem. During the execution of the program, our tool generated error messages, such as

```
Possible precision loss at printf!
Our checker is expecting the output string: -0.000002,
but with floating-point imprecision the output string is
instead: -0.000015
```

These messages were triggered by a `printf` call that prints out the calculated determinant of the matrix. The determinant values are very different when calculated with different precisions, indicating a significant imprecision error. The suspiciously small determinant values derived by either precision should alert the programmer of the possibility that an inverse actually does not exist.

Instead of doing a simple equality check like the one on line 12, an experienced programmer would check whether the calculated value falls within a small interval around zero, and let the program generate a warning if this is the case. This approach is theoretically satisfying as well: computing the equality of two real numbers is generally uncomputable. Checking for inclusion in a rational-bounded interval is, in contrast, computable.

5.4 Numeric Integral

We also used our tool to detect floating-point imprecision in numeric integral calculation. Figure 5 shows a program that computes the numeric integral $\int_0^\pi \sin(x)dx$ using a step size of 0.0001. Our tool reports multiple error messages while the program is running:

```
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!
```

```
1   int main() {
2     float lower = 0.0;
3     float upper = M_PI;
4     float step = 0.0001;
5
6     float result = 0.0;
7     float x;
8
9     for (x = lower; x < upper;
10        x += step) {
11      result += sin(x) * step;
12    }
13
14    printf("%f\n", result);
15
16    return 0;
17  }
```

Figure 5. Calculating $\int_0^\pi \sin(x)dx$

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Figure 6. The Quadratic Formula. We consider it with $\pm = -$.

```
Our checker is expecting the output string: 2.000000,
but with floating-point imprecision the output string is
instead: 2.000405
```

The "`Precision loss at <`" messages indicate that the loop condition check at line 9 of the program returned different results between native float and MPFR. It is shown in this example that the loop executed for extra iterations under native float format, which resulted in a slightly larger integral result.

This problem exemplifies how Real Semantics contributes both to finding functional correctness bugs as well as efficiency bugs. Due to floating-point imprecision, several extra and unnecessary loop iterations were performed that actually minimized accuracy. Real Semantics points towards the need for a more precise representation or perhaps a different loop iteration style.

5.5 Quadratic Formula

The quadratic formula (Figure 6) needs neither introduction nor justification of its importance. As such, we proceed directly to an exploration of its error properties. We follow a similar development to Panchekha et al. but using our tool in place of Herbie [13].

We used Real Semantics to explore the floating-point imprecision on a small test suite for the following test inputs:

$$a = 100, b = -8356218543 \times 10^{201}, c = -321432$$

Real Semantics points out that the double-precision result, $-inf$, is different from the higher-precision result $-3.8466203145113220 \times 10^{-206}$. We see that a very small value has become negative infinity due to imprecision. This is often caused by catastrophic cancellation. In our case, the numerator should evaluate to almost zero. To address this issue we re-define the quadratic formula piecewise:

$$\begin{array}{ll} \dfrac{2c}{-b+\sqrt{b^2+4ac}} & b<0 \\ \dfrac{-b-\sqrt{b^2-4ac}}{2a} & b>0 \end{array}$$

The former is a simplification allowed by the observation that

$$x - y \approx \frac{x^2 - y^2}{x + y}$$

This enables us to eliminate the square root, yielding:

$$\frac{b^2 - b^2 - 4ac}{2a(-b+\sqrt{b^2-4ac})}$$

which simplifies to

$$\frac{2c}{-b+\sqrt{b^2-4ac}}$$

Note that the subtraction in the numerator has become an addition in the numerator, thus eliminating the catastrophic cancellation: $(-b) - (-b)$.

Unfortunately, on the aforementioned parameters, $a = 100$, $b = -8356218543 \times 10^{201}$, $c = -321432$, our algorithm yields 0 instead of a result on the order of 10^{-206}; however, we have already infinitely improved our result.

Next, we consider a test case involving a large positive b, mainly:

$$a = 4328973e202,\ b = 432789658 \times 10^{201},\ c = 2134e2$$

In this case, a series of cancellations between large values should yield a final result near -9. In reality, we compute $-inf$. Returning again to our expression, we note that large values of b will overflow when squared. We first try re-using the same expression for negative b when b is large:

$$\begin{array}{ll} \dfrac{2c}{(-b+\sqrt{b^2-4ac})} & b<0 \text{ or } b>10^{127} \\ \dfrac{-b-\sqrt{b^2-4ac}}{2a} & b>0 \end{array}$$

This improves the answer to 0, but not quite to the desired -9. Unfortunately, we have now introduced catastrophic cancellation when the b is a large positive number because the denominator approaches $-b + b$.

For large positive values of b, we can manipulate the discriminant into the form $\sqrt{1+\varepsilon}$, which has a series expansion for small ε of $1 + \frac{1}{2}x + O(x^2)$. Inserting this term and massaging the equation slightly, we reveal a triply piecewise defined function:

```
float sumBinary(void) {
  float sum = 0;
  for (int i = 0; i < 10000; i++) {
    sum += 0.0001;
  }
  printf("%.10f\n", sum);
  return sum;
}
```

Figure 7. Summing Binary Numbers

$$\begin{array}{ll} \dfrac{2c}{-b+\sqrt{b^2-4ac}} & b<0 \\ \dfrac{-b-\sqrt{b^2-4ac}}{2a} & 0 \le b \le 10^{127} \\ -\dfrac{b}{a}+\dfrac{c}{b} & b>10^{127} \end{array}$$

Now, the result of our double-precision calculation has as many accurate bits as the result of the higher-precision calculation! However, we are still inaccurate in a few digits on the following inputs:

$$a = 100, b = -8356218543 \times 10^{201}, c = -321432$$
$$a = -1 \times 10^{10}, b = -1e10, c = 1e10$$
$$a = 3.214323 \times 10^{-201}, b = 5e-100, c = 1.32423$$

We leave it as an exercise to the reader to track down the source of these remaining inaccuracies.

5.6 Floating-Point Inaccuracies in Excel

For our final case-study, we wanted to see how our tool would fare on common floating-point errors. Microsoft Support created a list of five examples of simple floating-point computations that may give inaccurate results in Excel, a spreadsheet application that is widely used for computations [1]. We tested our program on C versions of the Excel examples, which are described in detail below.

5.6.1 Example Using Very Large Numbers

When 1.2E200 and 1E100 are added in Excel, the resulting value is 1.2E200. Our tool cannot catch the loss of precision in this example since 1.2E200 and 1E100 are too large to represent as floating-point numbers; however, we can detect this type of error when the numbers are within the range of possible C floating-point values.

5.6.2 Example Using Very Small Numbers

When 0.000123456789012345 and 1 are added in Excel, the resulting value is 1.00012345678901 instead of 1.000123456789012345. Our tool catches this loss of precision in the equivalent C program.

5.6.3 Repeating Binary Numbers and Calculations with Results Close to Zero

If 0.0001 is added together 10000 times in Excel, the resulting value is 0.999999999999996 instead of 1. Our tool catches this loss of precision in the equivalent C program, shown in Figure 7.

5.6.4 Example Adding a Negative Number

Excel computes $(43.1 - 43.2) + 1$ as 0.899999999999999 instead of 0.9. Our tool is able to catch this loss of precision if the computation is done in multiple steps. In this case, our tool will produce an error message for a program containing the following C code:

```
float A1 = (43.1 - 43.2);
A1 += 1;
printf("%.10f", A1);
```

but not when it is replaced with:

```
float A1 = (43.1 - 43.2) + 1;
printf("%.10f", A1);
```

We are unable to detect the floating point imprecision in the second piece of code because the loss of precision occurs before the value is stored as a `float`. In other words, by the time we are able to create a `SmartFloat` for `A1`, the loss of precision has already occurred. This is one limitation of our tool.

5.6.5 Example When a Value Reaches Zero

Excel computes $1.333 + 1.225 - 1.333 - 1.225$ as $-2.2204460492 5031\text{E} - 16$ instead of 0. Similar to the previous example, our tool is able to catch this loss of precision if the computation is done in multiple steps.

6. Future Work

Real Semantics achieved a number of goals outlined at the beginning of this paper. Nevertheless, there is still room for growth. Specifically, we find that there are key areas where our system could be further optimized to allow for better results.

6.1 Performance Optimization

The performance of our tool is a limiting factor right now, and its poor performance is largely due to the overhead of interpreting LLVM bitcode. The interpreter unfortunately introduces large overhead to all instructions, including integer and branch instructions that have nothing to do with floating-point semantics. Maintaining the internal states of the interpreter like the large `SmartFloat` map also comes at a cost that is largely unnecessary if we can execute (at least part of) the program natively.

A potential solution to this issue would be to re-write our tool as a compiler pass. Doing so would allow the non-floating-point-related portion of the program to run completely on the native platform. This is an approach that we would potentially explore as part of future work.

If our tool existed as a compiler pass we could additionally improve the precision of our high-precision floating-point numbers. We do not see floating-point numbers until the C compiler has transformed them to single- or double-precision bitvectors. Numeric literals, such as 3.3 that are not accurately representable in double-precision are already inaccurate by the time we generate a higher-precision analogue. This is particularly disappointing because the high-precision value is initialized by extending the low-precision value with zeros.

6.2 Evaluation Optimization

Part of our larger goal was to evaluate our tool by running large programs and finding meaningful floating point precision errors. However, due to the performance issues outlined above, we were unable to run significantly large programs. In the future, once we optimize performance, we would like to evaluate our tool by running large programs or mathematical suites that would allow us to find a greater variety of floating-point precision bugs.

In general, had we had more time, we would have liked to further automate the developing process by automatically generating test inputs. Such improvements would allow us to have a shorter development life-cycle and spend more time further optimizing the tool. Similarly, although we currently have a hack for `memcpy`, in the future, we would like our tool to be able to better handle memory issues.

Analyzing large programs and suites of algorithms almost certainly requires a significant improvement to our testing strategy. In the future, we want to incorporate randomized swarm testing to alleviate the need for the user to custom design test-suites that reveal floating-point imprecision. As we have learned, programmers have difficulty imagining the way in which their programs may fail.

7. Discussion

7.1 Limitations

When using Real Semantics it is important to keep in mind two key limitations of this system:

- higher-precision values are not arbitrary precision values, and
- higher-precision results do not reveal the *correct* answer, just a more *precise* answer.

An obvious extension to Real Semantics is the use of arbitrary precision arithmetic in place of MPFR's fixed, high-precision arithmetic. Arbitrary precision arithmetic permits the use of whatever precision is necessary to represent intermediate results. Moreover, the memory and computation requirements adapt to the inputs given.

Another possible extension is additionally reporting the results of applying the given inputs to an obviously correct reference implementation. Reference implementations guard against numerical transformations to an algorithm that changes its functional behavior. In particular, the “return 0” function will always be maximally precise.

7.2 Applications

Real Semantics nicely complements existing techniques for static analysis. Particularly because it dynamically searches for precision loss in imperative code, including code with loops. In this manner, it is more general and limited than Herbie [13]. The two tools together form a powerful work-flow for debugging and developing floating-point algorithms. Real Semantics directs the programmer towards problematic areas of code. Once the algorithm at fault is crisply circumscribed and understood, Herbie may be employed to automatically generate a precise-by-construction implementation of the problematic algorithm.

Real Semantics also holds pedagogical value. Floating-point imprecision often seems arbitrary to students. Real Semantics provides a mechanical assistant that calculates an approximation to the error inherent in a given floating-point number. Calculating this error from first principles is complex, even with specific floating-point inputs. Because Real Semantics permits the user to provide the test cases, the student can explore the space of floating-point numbers independently. This is not easy with verification or static analysis tools. An improvement that benefits both students and developers would be a visual representation of inputs that cause dramatic reductions in floating-point precision.

8. Conclusion

We built Real Semantics, a dynamic floating-point imprecision detector that runs on a modified LLVM IR interpreter. Our tool tracks higher-precision representations of the numbers calculated by the interpreted program through the use of MPFR to calculate actual real numbers. Further, Real Semantics notifies the user of imprecision only when it substantially changes program behavior, which results in a high-level of usability. Lastly, our tool identifies precision loss events by line number and variable name in order to allow the user to more easily find and solve, if necessary, any floating-point precision errors.

In this paper, we have outlined our tool in detail, including other related work, our approach for the implementation, and the various test cases used to aid in our evaluation. We also outlined some limitations as explained in the need for future work as it relates to performance and evaluation optimizations. All in all, we have presented a tool that can help users detect floating point imprecision in programs written in C.

As computers continue to increase adoption across all levels of society and more users become programmers – fully fledged citizens in interacting with their computers – the need to support programmers using floating-point arithmetic grows only more urgent. We have three fruitful directions of attack right now: analysis, precise-by-construction, and verification. Verification provides the bedrock on which all other techniques can rely for assurance. Analysis supports both legacy code and explorations of custom written code. Ultimately, precise-by-construction techniques may supersede analyzers just as memory-safe languages have taken large market share from the unsafe languages.

References

[1] Floating-point arithmetic may give inaccurate results in excel. URL https://support.microsoft.com/en-us/kb/78113.

[2] lli - directly execute programs from llvm bitcode. URL http://llvm.org/docs/CommandGuide/lli.html.

[3] LLVM language reference manual. URL http://llvm.org/docs/LangRef.html.

[4] C program to find inverse of a matrix. URL http://www.sanfoundry.com/c-program-find-inverse-matrix/.

[5] Gnu mpfr. URL http://www.mpfr.org/.

[6] Basic ray tracer, stage 1, 2012. URL http://renderspud.blogspot.com/2012/04/basic-ray-tracer-stage-1.html.

[7] E. T. Barr, T. Vo, V. Le, and Z. Su. Automatic detection of floating-point exceptions. *SIGPLAN Not.*, 48(1):549–560, Jan 2013. ISSN 0362-1340. . URL http://doi.acm.org/10.1145/2480359.2429133.

[8] S. Boldo and J.-C. Filliatre. Formal verification of floating-point programs. In *Computer Arithmetic, 2007. ARITH '07. 18th IEEE Symposium on*, pages 187–194, June 2007. .

[9] S. Boldo, J.-H. Jourdan, X. Leroy, and G. Melquiond. Verified compilation of floating-point computations. *Journal of Automated Reasoning*, 54(2):135–163, 2015. ISSN 0168-7433. . URL http://dx.doi.org/10.1007/s10817-014-9317-x.

[10] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR)*, 23(1): 5–48, 1991.

[11] A. Green. libffi - a portable foreign function interface library. URL https://sourceware.org/libffi/.

[12] B. Lambov. The reallib project. URL http://daimi.au.dk/~barnie/RealLib/.

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

[14] K. Vuik. Some disasters caused by numerical errors, 2006. URL http://ta.twi.tudelft.nl/users/vuik/wi211/disasters.html.