A Formally-Verified C Compiler Supporting Floating-Point Arithmetic

Sylvie Boldo  Jacques-Henri Jourdan
Xavier Leroy  Guillaume Melquiond

Inria Saclay–Île-de-France & LRI, Université Paris Sud, CNRS
Inria Paris–Rocquencourt

ANR-11-INSE-003 VERASCO

2013-10-07
Floating-Point Arithmetic and Optimizations

Example (FastTwoSum)

double y, z;
y = 0x1p-53 + 0x1p-78;  // y = 2^{-53} + 2^{-78} > \frac{1}{2}ulp(1)
z = ((1. + y) - 1.) - y;
printf("%a\n", z);  // Dekker says: z = 2^{-53} - 2^{-78}
Floating-Point Arithmetic and Optimizations

Example (FastTwoSum)

double y, z;
y = 0x1p-53 + 0x1p-78;
z = ((1. + y) - 1.) - y;
printf("\%a\n", z);

 GCC 4.6.3 on x86 architecture

<table>
<thead>
<tr>
<th>Optimization level</th>
<th>Program result</th>
</tr>
</thead>
<tbody>
<tr>
<td>-O0 (x86-32)</td>
<td>-0x1p-78</td>
</tr>
<tr>
<td>-O0 (x86-64)</td>
<td>0x1.fffffffffp-54</td>
</tr>
<tr>
<td>-O1, -O2, -O3</td>
<td>0x1.fffffffffp-54</td>
</tr>
<tr>
<td>-Ofast</td>
<td>0x0p+0</td>
</tr>
</tbody>
</table>
Floating-Point Arithmetic and Optimizations

Example (FastTwoSum)

```c
double y, z;
y = 0x1p-53 + 0x1p-78;  // y = 2^{-53} + 2^{-78} > \frac{1}{2}ulp(1)
z = ((1. + y) - 1.) - y;
printf("%a\n", z);          // Dekker says: z = 2^{-53} - 2^{-78}
```

GCC 4.6.3 on x86 architecture

<table>
<thead>
<tr>
<th>Optimization level</th>
<th>Program result</th>
</tr>
</thead>
<tbody>
<tr>
<td>-00 (x86-32)</td>
<td>-0x1p-78</td>
</tr>
<tr>
<td>-00 (x86-64)</td>
<td>0x1.fffffffffp-54</td>
</tr>
<tr>
<td>-01, -02, -03</td>
<td>0x1.fffffffffp-54</td>
</tr>
<tr>
<td>-0fast</td>
<td>0x0p+0</td>
</tr>
</tbody>
</table>
General opinion

*From a practical perspective, preserving the “floating point” semantics is only interesting if not doing so will result in an execution error. That is, from a programmer’s perspective, playing “fast and loose” with floating semantics is generally OK if the resulting executable does what you want and runs fast.*

— Reviewer
Floating-Point Arithmetic and Compilers

General opinion

*From a practical perspective, preserving the “floating point” semantics is *only interesting if not doing so will result in an execution error. That is, from a programmer’s perspective, playing “fast and loose” with floating semantics is generally OK if the resulting executable does what you want and *runs fast.*

— Reviewer
General opinion

From a practical perspective, preserving the “floating point” semantics is only interesting if not doing so will result in an execution error. That is, from a programmer’s perspective, playing “fast and loose” with floating semantics is generally OK if the resulting executable does what you want and runs fast.

— Reviewer
Trivia

What is the oldest wrong-code bug-report still open for GCC?
Floating-Point Arithmetic and Compilers

### Trivia

What is the oldest *wrong-code* bug-report still open for GCC?

### Answer

Bug #323: “optimized code gives strange floating point results”.
Trivia

What is the oldest wrong-code bug-report still open for GCC?

Answer

Bug #323: “optimized code gives strange floating point results”.

Some people call this a bug in the x87 series. Other call it a bug in gcc. Regardless of where one wishes to put the blame, this problem will not be fixed. Period.

— GCC developer, 2005
Floating-Point Arithmetic and Compilers

Trivia

What is the oldest wrong-code bug-report still open for GCC?

Answer

Bug #323: “optimized code gives strange floating point results”.

Some people call this a bug in the x87 series. Other call it a bug in gcc. Regardless of where one wishes to put the blame, this problem will not be fixed. Period.

— GCC developer, 2005

Answer continued

109 duplicate bug-reports!
Floating-Point Arithmetic and Compilers

Trivia
What is the oldest wrong-code bug-report still open for GCC?

Answer
Bug #323: “optimized code gives strange floating point results”.

Some people call this a bug in the x87 series. Other call it a bug in gcc. Regardless of where one wishes to put the blame, this problem will not be fixed. Period.

— GCC developer, 2005

Answer continued
109 duplicate bug-reports!
Bug #55939: “gcc miscompiles gmp-5.0.5 on m68k-linux”.

S. Boldo, J-H. Jourdan, X. Leroy, G. Melquiond
Floating-Point Arithmetic and Users

What Can Users Expect From FP Arithmetic?

*Rounding takes a number regarded as infinitely precise and, if necessary, modifies it to fit in the destination’s format [...]. Every operation shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then rounded that result [...].*

— IEEE-754 2008
What Languages Say About FP Arithmetic

Java SE 7 (15.4 FP-strict expressions)

Within an expression that is not FP-strict, some leeway is granted for an implementation to use an extended exponent range to represent intermediate results.

C99 (5.2.4.2.2 Characteristics of floating types)

The values of operations with floating operands [...] are evaluated to a format whose range and precision may be greater than required by the type.

Fortran 2008 (7.1.5.2.4 Eval of numeric intrinsic operations)

Two expressions of a numeric type are mathematically equivalent if, for all possible values of their primaries, their mathematical values are equal.
Java SE 7 (15.4 FP-strict expressions)
Within an expression that is not FP-strict, some leeway is granted for an implementation to use an extended exponent range to represent intermediate results.

C99 (5.2.4.2.2 Characteristics of floating types)
The values of operations with floating operands [...] are evaluated to a format whose range and precision may be greater than required by the type.

Fortran 2008 (7.1.5.2.4 Eval of numeric intrinsic operations)
Two expressions of a numeric type are mathematically equivalent if, for all possible values of their primaries, their mathematical values are equal.
Java SE 7 (15.4 FP-strict expressions)
Within an expression that is not FP-strict, some leeway is granted for an implementation to use an extended exponent range to represent intermediate results.

C99 (5.2.4.2.2 Characteristics of floating types)
The values of operations with floating operands [...] are evaluated to a format whose range and precision may be greater than required by the type.

Fortran 2008 (7.1.5.2.4 Eval of numeric intrinsic operations)
Two expressions of a numeric type are mathematically equivalent if, for all possible values of their primaries, their mathematical values are equal.
What Languages Say About FP Arithmetic

Java SE 7 (15.4 FP-strict expressions)
Within an expression that is not FP-strict, some leeway is granted for an implementation to use an extended exponent range to represent intermediate results.

C99 (5.2.4.2.2 Characteristics of floating types)
The values of operations with floating operands [...] are evaluated to a format whose range and precision may be greater than required by the type.

Fortran 2008 (7.1.5.2.4 Eval of numeric intrinsic operations)
Two expressions of a numeric type are mathematically equivalent if, for all possible values of their primaries, their mathematical values are equal.
On Compilers and Trust

Trivia

How do avionics developers explain to a certification authority that their C programs are airworthy? (E.g. DO-178 regulations.)
On Compilers and Trust

Trivia

How do avionics developers explain to a certification authority that their C programs are *airworthy*? (E.g. DO-178 regulations.)

Answer

- They disable compiler optimizations.
- They read the assembly code generated by the C compiler.
On Compilers and Trust

Trivia

How do avionics developers explain to a certification authority that their C programs are airworthy? (E.g. DO-178 regulations.)

Answer

- They disable compiler optimizations.
- They read the assembly code generated by the C compiler.

Trust in the compilers? Absolutely none.
How to Improve the Situation

Proposal

Build a C compiler that can be trusted and does not mess with floating-point code.
How to Improve the Situation

Proposal

Build a C compiler that can be trusted and does not mess with floating-point code.

Components

**CompCert**: a C compiler targeting ARM, PowerPC, x86-SSE2

- mathematical specification of the semantics of C and target,
- formal proof that compilation preserves semantics.
How to Improve the Situation

Proposal

Build a C compiler that can be trusted and does not mess with floating-point code.

Components

**CompCert**: a C compiler targeting ARM, PowerPC, x86-SSE2
- mathematical specification of the semantics of C and target,
- formal proof that compilation preserves semantics.

**Flocq**: a Coq formalization of FP arithmetic
- multi-radix, multi-format, multi-precision arithmetic,
- comprehensive library, including computable operations.
Outline

1. Introduction
2. CompCert, a formally-verified compiler
3. Flocq, a Coq formalization of FP arithmetic
4. CompCert with floating-point support
5. Conclusion
Outline

1. Introduction

2. CompCert, a formally-verified compiler
   - Semantics preservation
   - Floating-point arithmetic in the earlier days

3. Flocq, a Coq formalization of FP arithmetic

4. CompCert with floating-point support

5. Conclusion
Semantics Preservation

**Theorem**

Let $S$ be a source C program free of undefined behaviors. Assume that the CompCert compiler, invoked on $S$, does not report a compile-time error, but instead produces executable code $E$. Then, any observable behavior $B$ of $E$ is one of the possible observable behaviors of $S$. 

S. Boldo, J-H. Jourdan, X. Leroy, G. Melquiond
A Formally-Verified C Compiler Supporting FP Arithmetic
Semantics Preservation

**Theorem**

Let $S$ be a source C program free of undefined behaviors. Assume that the CompCert compiler, invoked on $S$, does not report a compile-time error, but instead produces executable code $E$. Then, any observable behavior $B$ of $E$ is one of the possible observable behaviors of $S$.

**Corollary**

You do not need to know how the compiler works, nor how the target environment behaves, in order to know what the produced executable will compute.
Semantics Preservation

**Theorem**

Let $S$ be a source C program free of undefined behaviors. Assume that the CompCert compiler, invoked on $S$, does not report a compile-time error, but instead produces executable code $E$. Then, any observable behavior $B$ of $E$ is one of the possible observable behaviors of $S$.

**Implicit assumptions**

The compiler behaves as proved.

The target environment is correctly formalized.
Semantics Preservation

Theorem

Let $S$ be a source C program free of undefined behaviors. Assume that the CompCert compiler, invoked on $S$, does not report a compile-time error, but instead produces executable code $E$. Then, any observable behavior $B$ of $E$ is one of the possible observable behaviors of $S$.

Implicit assumptions

- The compiler behaves as proved.
Semantics Preservation

Theorem

Let $S$ be a source C program free of undefined behaviors. Assume that the CompCert compiler, invoked on $S$, does not report a compile-time error, but instead produces executable code $E$. Then, any observable behavior $B$ of $E$ is one of the possible observable behaviors of $S$.

Implicit assumptions

- The compiler behaves as proved.
- The target environment is correctly formalized.
Semantics Preservation

Semantics preservation guarantees that reading the semantics of the input language of the compiler is sufficient to understand how the programmer’s code will end up.
Semantics Preservation

Semantics preservation guarantees that reading the semantics of the input language of the compiler is sufficient to understand how the programmer’s code will end up.

Example (Clight semantics)

\[
\text{Inductive } \text{step} : \text{state } \to \text{trace } \to \text{state } \to \text{Prop} := \\
\quad \ldots \\
\quad | \ \text{step_seq} : \text{forall } f \ s1 \ s2 \ k \ e \ le \ m, \\
\quad \quad \text{step} (\text{State } f \ (\text{Ssequence } s1 \ s2) \ k \ e \ le \ m) \ \\
\quad \quad \text{E0} (\text{State } f \ s1 \ (\text{Kseq } s2 \ k) \ e \ le \ m) \\
\quad \ldots
\]
Semantics Preservation

Semantics preservation guarantees that reading the semantics of the input language of the compiler is sufficient to understand how the programmer’s code will end up.

Example (Clight semantics)

```coq
Inductive step : state -> trace -> state -> Prop :=
  ...
  | step_seq : forall f s1 s2 k e le m,
    step (State f (Ssequence s1 s2) k e le m)
  E0 (State f s1 (Kseq s2 k) e le m)
  ...
```

Disclaimer: it is painful (about 1000 lines of Coq),
Semantics Preservation

Semantics preservation guarantees that reading the semantics of the input language of the compiler is sufficient to understand how the programmer’s code will end up.

Example (Clight semantics)

```coq
Inductive step: state -> trace -> state -> Prop :=
  ... |
  step_seq: forall f s1 s2 k e le m,
  step (State f (Ssequence s1 s2) k e le m)
  E0 (State f s1 (Kseq s2 k) e le m)
  ...
```

Disclaimer: it is painful (about 1000 lines of Coq),
- but not as painful as reading the code of a whole compiler,
- or as reading every generated assembly code.
What a Compiler Does with FP Code

1. **Parse** literal constants from the source code.
2. **Perform** some optimizations, e.g., constant propagation.
3. **Emulate** primitive operations missing from the target, e.g., integer ↔ float conversions.
4. **Output** constants to the assembly code.
How CompCert Handled FP Arithmetic Before

Earlier CompCert: *axiomatized* floating-point arithmetic.
How CompCert Handled FP Arithmetic Before

Earlier CompCert: axiomatized floating-point arithmetic.

**Consequences**

- Parsing done through external functions, e.g. `strtod`.
  \[\Rightarrow\] “rounding error for values very close to half-way points”. 
How CompCert Handled FP Arithmetic Before

Earlier CompCert: *axiomatized* floating-point arithmetic.

**Consequences**

- Parsing done through *external functions*, e.g. `strtod`.
  ⇒ “rounding error for values very close to half-way points”.
- FP constant propagation performed by the *host system*.
  ⇒ double-rounding issues.
Earlier CompCert: *axiomatized* floating-point arithmetic.

**Consequences**

- Parsing done through *external functions*, e.g. `strtod`.
  - “rounding error for values very close to half-way points”.

- FP constant propagation performed by the *host system*.
  - double-rounding issues.

- No proof of semantics preservation.
  - *possibly incorrect* code transformations.
Outline

1. Introduction

2. CompCert, a formally-verified compiler

3. Flocq, a Coq formalization of FP arithmetic
   - Floating-point formats
   - Operations and specification

4. CompCert with floating-point support

5. Conclusion
Flocq’s Binary FP Numbers

Definition (Floating-point numbers as a sum type)

**Inductive** \(\text{binary\_float} :=\)
- \(\text{B754\_zero} : \text{bool} \to \text{binary\_float}\)
- \(\text{B754\_infinity} : \text{bool} \to \text{binary\_float}\)
- \(\text{B754\_nan} : \text{bool} \to \text{nan\_pl} \to \text{binary\_float}\)
- \(\text{B754\_finite} : \forall (s : \text{bool}) (m : \text{positive}) (e : \text{Z}), \text{bounded} m e = \text{true} \to \text{binary\_float}.\)

- parametrized by precision and range of exponent,
- supports signed zeros, infinities, (sub)normal numbers, NaNs.
Floating-point Operators

Supported operations:

- addition, multiplication, division, square root,
Floating-point Operators

Supported operations:
- addition, multiplication, division, square root,
- conversion from/to standard binary representation.
Floating-point Operators

Supported operations:
- addition, multiplication, division, square root,
- conversion from/to standard binary representation.

Critical feature: these are **computable** functions.
IEEE-754 Compliance

**Theorem (Bmult_correct)**

*Given* \(x\) *and* \(y\) *two binary float numbers, \(m\) *a rounding mode, if* \(z = \text{round}(m, B2R(x) \times B2R(y))\), *we have*

\[
\begin{cases}
B2R(B\text{mult}(m, x, y)) = z & \text{if } |z| < 2^E, \\
B\text{mult}(m, x, y) = \\
\text{overflow}(m, B\text{sign}(x) \times B\text{sign}(y)) & \text{otherwise}.
\end{cases}
\]

*Moreover, if the result is not NaN,*

\[B\text{sign}(B\text{mult}(m, x, y)) = B\text{sign}(x) \times B\text{sign}(y).\]
Outline

1. Introduction

2. CompCert, a formally-verified compiler

3. Flocq, a Coq formalization of FP arithmetic

4. CompCert with floating-point support
   - Parsing and output of numeric literals
   - Constant propagation
   - Conversions from/to Integers

5. Conclusion
How to parse 0.314e1 in the C input code?

1. Parse integers 314 and 1.
2. Normalize into 314 · 10^{-2}.
3. Perform a FP division with Flocq: \texttt{round(NE, 314/100)}. 

S. Boldo, J-H. Jourdan, X. Leroy, G. Melquiond

A Formally-Verified C Compiler Supporting FP Arithmetic
Parsing and Output of Numeric Literals

How to parse 0.314e1 in the C input code?

1. Parse integers 314 and 1.
2. Normalize into $314 \cdot 10^{-2}$.
3. Perform a FP division with Flocq: \( \text{round}(\text{NE}, 314/100) \).

How to pass it to the assembler?

1. Ask Flocq for the bit-level representation.
2. Output it as an integer: `.quad 0x40091eb851eb851f`
Constant Propagation

Source code

```c
inline double f(double x) {
    if (x < 1.0) return 1.0; else return 1.0 / x;
}
double g(void) {
    return f(3.0);
}
```

After inlining and constant propagation

```c
double g(void) {
    return 0x1.5555555555555p-2;
}
```

Note: rounding to nearest was assumed.
Constant Propagation

Source code

```c
inline double f(double x) {
    if (x < 1.0) return 1.0; else return 1.0 / x;
}
double g(void) {
    return f(3.0);
}
```

After inlining and constant propagation

```c
double g(void) {
    return 0x1.5555555555555p-2;
}
```

Note: rounding to nearest was assumed.
Emulation: Conversion from/to Integers

Some conversions are not supported by target architectures,

- so we emulate them with some sequences of operations,
- and we have formally proved the semantics preservation.
Emulation: Conversion from/to Integers

Some conversions are not supported by target architectures,
- so we emulate them with some sequences of operations,
- and we have formally proved the semantics preservation.

Example (From unsigned to double)

x86-SSE2 converts to binary64 only from signed 32-bit integers.

\[
\begin{align*}
n < 0x80000000 & \quad ? \quad (\text{double})(\text{(int)} \ n) \\
& \quad : \quad (\text{double})(\text{(int)}(n - 0x80000000)) + 0x1.p31
\end{align*}
\]
Emulation: Conversion from/to Integers

Some conversions are not supported by target architectures,

- so we emulate them with some sequences of operations,
- and we have formally proved the semantics preservation.

Example (From unsigned to double)

x86-SSE2 converts to binary64 only from signed 32-bit integers.

\[
\text{n < 0x80000000 ? (double)(((int) n)}
\ :
\text{(double)(((int)(n - 0x80000000)) + 0x1.p31)}
\]

PowerPC does not support conversion from integers to binary64.

\[
\text{fmake(0x43000000, n ^ 0x80000000)}
\ - \ \text{fmake(0x43000000, 0x80000000)}
\]
Outline

1. Introduction
2. CompCert, a formally-verified compiler
3. Flocq, a Coq formalization of FP arithmetic
4. CompCert with floating-point support
5. Conclusion
   - Performances
   - Conclusion
Performances: FFTW Pseudo-Benchmark

Example (Fastest Fourier Transform in the West)

/* Generated by: ../../../genfft/gen_r2r.native -compact -variables 4 -pipeline-latency 4 -redft01 -n 8 -name e01_8 -include r2r.h */
void e01_8(const R *I, R *O, stride is, stride os, INT v, INT ivs, INT ovs)
{
    const E KP1_662939224 = ((E) +1.662939224605090474157576755235811513477121624);
    const E KP1_111140466 = ((E) +1.11140466039204449485661627897065748749874382);
    const E KP390180644 = ((E) +0.390180644032256656965973695404481855383236);
    const E KP1_961570560 = ((E) +1.9615705608064089825236447268478073947867462);
    ...

    for (i = v; i > 0; i = i - 1, I = I + ivs, O = O + ovs) {
        E T7, Tl, T4, Tk, Td, To, Tg, Tn;
        {
            E T5, T6, T1, T3, T2;
            T5 = I[(is[2])];
            T6 = I[(is[6])];
            T7 = (((KP1_847759065) * (T5)) + (KP765366864 * T6));
            T1 = ((KP765366864 * T5) - ((KP1_847759065) * (T6)));
            ...
        }
    }
}
Performances: FFTW Pseudo-Benchmark

Example (Fastest Fourier Transform in the West)

```c
/* Generated by: ../../../genfft/gen_r2r.native -compact -variables 4 -pipeline-latency 4 -redft01 -n 8 -name e01_8 -include r2r.h */
void e01_8(const R *I, R *O, stride is, stride os, INT v, INT ivs, INT ovs)
{
  const E KP1_662939224 = ((E) +1.662939224605090474157576755235811513477121624);
  const E KP1_111140466 = ((E) +1.11114046603920449485661627897065748749874382);
  const E KP390180644 = ((E) +0.390180644032256535696569736954044481855383236);
  const E KP1_961570560 = ((E) +1.961570560806460898252364472268478073947867462);
  ...
  for (i = v; i > 0; i = i - 1, I = I + ivs, O = O + ovs) {
    E T7 , Tl , T4 , Tk , Td , To , Tg , Tn;
    {
      E T5 , T6 , T1 , T3 , T2;
      T5 = I[(is[2])];
      T6 = I[(is[6])];
      T7 = (((KP1_847759065) * (T5)) + (KP765366864 * T6));
      T1 = ((KP765366864 * T5) - (((KP1_847759065) * (T6)));
    }
  }
}
```

Target: x86-32 with SSE2 arithmetic (everything fits in L1 cache).
Compilers: GCC 4.6.3 (-O3) vs CompCert 1.13.
Results: CompCert’s compiled code is 25% slower than GCC’s,
Performances: FFTW Pseudo-Benchmark

Example (Fastest Fourier Transform in the West)

```c
/* Generated by: ../../../genfft/gen_r2r.native -compact -variables 4 -pipeline latency 4 -redft01 -n 8 -name e01_8 -include r2r.h */
void e01_8(const R *I, R *O, stride is, stride os, INT v, INT ivs, INT ovs) {
    const E KP1_662939224 = ((E) +1.662939224605090474157576755235811513477121624);
    const E KP1_111140466 = ((E) +1.1114046603920449485661627897065748749874382);
    const E KP390180644 = ((E) +0.3901806440322563569659736954044481855383236);
    const E KP1_961570560 = ((E) +1.96157056080640898252364472268478073947867462);
    ... 
    for (i = v; i > 0; i = i - 1, I = I + ivs, O = O + ovs) {
        E T7, Tl, T4, Tk, Td, To, Tg, Tn;
        { 
            E T5, T6, T1, T3, T2;
            T5 = I[(is[2])];
            T6 = I[(is[6])];
            T7 = (((KP1_847759065) * (T5)) + (KP765366864 * T6));
            Tl = ((KP765366864 * T5) - (((KP1_847759065) * (T6)));
        }
    }
    ... 
}
```

Target: x86-32 with SSE2 arithmetic (everything fits in L1 cache).
Compilers: GCC 4.6.3 (-O3) vs CompCert 1.13.
Results: CompCert’s compiled code is **25% slower** than GCC’s, but **160% faster** than GCC’s at -O0.
Features

- simple yet useful semantics for FP numbers (IEEE-754!),
- no dependencies on the host system during compilation,
- a complete formal proof of semantics preservation
  (about 4,300 new lines of Coq proofs).
Conclusion

Features

- simple yet useful semantics for FP numbers (IEEE-754!),
- no dependencies on the host system during compilation,
- a complete formal proof of semantics preservation
  (about 4,300 new lines of Coq proofs).

Current limitations

- rounding to nearest is assumed,
- "float" computations are done in binary64,
- few optimizations (missing some range information).
Questions?

CompCert:  http://compcert.inria.fr/
Flocq:  http://flocq.gforge.inria.fr/
Verasco:  http://verasco.imag.fr/