A Formally-Verified C Compiler Supporting Floating-Point Arithmetic

Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, Guillaume Melquiond

A Formally-Verified C Compiler Supporting Floating-Point Arithmetic

Sylvie Boldo Jacques-Henri Jourdan Xavier Leroy Guillaume Melquiond

***Abstract*—Floating-point arithmetic is known to be tricky: roundings, formats, exceptional values. The IEEE-754 standard was a push towards straightening the field and made formal reasoning about floating-point computations possible. Unfortunately, this is not sufficient to guarantee the final result of a program, as several other actors are involved: programming language, compiler, architecture. The CompCert formally-verified compiler provides a solution to this problem: this compiler comes with a mathematical specification of the semantics of its source language (ISO C90) and target platforms (ARM, PowerPC, x86-SSE2), and with a proof that compilation preserves semantics. In this paper, we report on our recent success in formally specifying and proving correct CompCert's compilation of floating-point arithmetic. Since CompCert is verified using the Coq proof assistant, this effort required a suitable Coq formalization of the IEEE-754 standard; we extended the Flocq library for this purpose. As a result, we obtain the first formally verified compiler that provably preserves the semantics of floating-point programs.**

***Index Terms*—floating-point arithmetic; verified compilation; formal proof; floating-point semantic preservation;**

I. Introduction

Use and study of floating-point (FP) arithmetic have intensified since the 70s [1], [2]. At that time, computations were not standardized and various architectures gave different answers on the same program. Since the IEEE-754 standard of 1985 and its revision in 2008 [3], things should have changed as reproducibility was a keyword. Each basic operation is supposed to be computed as if the computation was done with infinite precision and then rounded. The goal was that the same program could be run on various platforms and give the same result. It allowed the development of many algorithms coming with mathematical proofs based on the fact that operations were correctly rounded. Since the 2000s, this was even pushed to formal proofs of algorithms or hardware components: in PVS [4], in ACL2 [5], in HOL-light [6] and in Coq [7], [8]. The basic axiom for algorithms and the basic goal for hardware components was still that all the operations be correctly rounded.

To complicate matters further, the processor architecture is not the only party responsible for the computed results. Stand also accused the programming language and the compiler used. We will focus on the compiler, as it can be unfaithful to what the programmer wants or what was proved from the written code. To illustrate what the compiler can change, here is a small example in C:

```
#include <stdio.h>

int main () {
  double y, z;
  y = 0x1p-53 + 0x1p-78;       // y = 2^-53 + 2^-78
  z = 1. + y - 1. - y;
  printf("%a\n", z);
  return 1;
}
```

Experts may have recognized a Fast-Two-Sum [2] that computes the round-off error of a FP addition by $((a+b)-a)-b$ for $|a| \geq |b|$. This very simple program compiled with GCC 4.6.3 gives three different answers on an x86 32-bit architecture depending on the chosen level of optimization.

Optimization level	Program result
-O0	-0x1p-78
-O1, -O2, -O3	0x1.fffffp-54
-Ofast	0x0p+0

How can we explain the three results? For the first two results, the answer lies in the x86 architecture: it may compute with double precision (64 bits, 53 bits of precision) or with extended precision (80 bits, 64 bits of precision). For each operation, the compiler may choose to round the infinitely-precise result either to extended precision, or to double precision, or first to extended and then to double precision. The latter is called a *double rounding*. In all cases, y is computed exactly: $y = 2^{-53} + 2^{-78}$.

With the -O0 optimization, all the computations are performed with extended precision and rounded in double precision only once at the end. With -O1 and higher, the result $(1 + y) - 1$ is pre-computed by the compiler and the program only computes the last subtraction and prints the value. With -Ofast, there is no computation at all in the program but only the output of the constant 0. This optimization level turns on `-funsafe-math-optimizations` which allows the reordering of FP operations. It is explicitly stated in GCC documentation that this option "can result in incorrect output for programs which depend on an exact implementation of IEEE or ISO rules/specifications for math functions".

Another possible discrepancy comes from the use of the *fused-multiply-add* operator (FMA). For example, consider $a \times b + c \times d$. When a FMA is available, the compiler may choose either $\circ(a \times b + \circ(c \times d))$, or $\circ(\circ(a \times b) + c \times d)$, or $\circ(\circ(a \times b) + \circ(c \times d))$ which may give different results. A wide set of examples of strange FP behaviors can be found in [9], [10].

As surprising as it may seem, all the discrepancies described so far are allowed by the ISO C standard [11], which leaves much freedom to the compiler in the way it implements FP computations. Sometimes, optimizing compilers take additional liberties with the source programs, generating

executable code that exhibits behaviors not allowed by the specification of the source language. This is called *miscompilation*. Consider the following example, adapted from GCC's infamous "bug #323":

```
void test(double x, double y)
{
  const double y2 = x + 1.0;
  if (y != y2) printf("error\n");
}

int main()
{
  const double x = .012;
  const double y = x + 1.0;
  test(x, y);
  return 0;
}
```

On x86 32-bits at optimization level -O1, all versions of GCC prior to 4.5 miscompile this code as follows: the expression `x + 1.0` in function `test` is computed in extended precision, as allowed by C, but the compiler omits to round it back to double precision when assigning to `y2`, as prescribed by the C standard. Consequently, `y` and `y2` compare different, while they must be equal according to the C standard. Miscompilation happens more often than one may think: Yang *et al* [12] tested many production-quality C compilers using differential random testing, and found hundreds of cases where the compiler either crashes at compile-time or—much worse—silently generates an incorrect executable from a correct source program.

As the compiler gives so few guarantees on how it implements FP arithmetic, it therefore seems impossible to guarantee the result of a program. In fact, most analysis of FP programs assume correct compilation and a strict application of the IEEE-754 standard where no extended registers nor FMA are used. This assumption is correct for embedded software such as those used in avionics. For the automatic analysis of C programs, a successful approach is based on abstract interpretation, and tools include Astre [13], [14] and Fluctuat [15]. Another method to specify and prove behavioral properties of FP programs is deductive verification system: specification languages has to take into account FP arithmetic. This has been done for Java in JML [16], for C in ACSL [17], [18]. However, all these works only follow strictly the IEEE-754 standard, with neither FMA, nor extended registers, nor considering optimization aspects. Recently, several possibilities have been offered to take these aspects into account. One approach is to cover all the ways a compiler may have compiled each FP operation and to compute an error bound that stands correct whatever the compiler choices [19]. Another approach is to analyze the assembly code to get all the precision information [20].

Our approach is different: rather than trying to account for all the changes a compiler may have silently introduced in a FP program, we have focused on getting a correct and predictable compiler that supports FP arithmetic. Concerning compilers and how to make them more trustworthy, Milner and Weyrauch [21] were the first to mechanically prove the correctness of a compiler, although for a very simple language of expressions. Moore [22] extended this approach to an implementation of the Piton programming language. Li *et al* [23] showed that one can compile programs with proof, directly from the logic of the HOL4 theorem prover. A year later, Myreen [24] made contributions both to approaches for verification of programs and methods for automatically constructing correct code.

To build our compiler, we started from CompCert [25], a formally-verified compiler described in Section III and extended it with FP arithmetic. As CompCert is developed using the Coq proof assistant, we had to build on a Coq library formalizing FP arithmetic: we relied on the Flocq library [8] and extended it to serve the needs of a verified compiler. With all these components, we were able to get a correct, predictable compiler that conforms strictly to the IEEE-754 standard.

In this article, we present in Section II the semantics of FP arithmetic in programs, depending in particular on the programming language. In Section III, we describe the CompCert certified compiler. We will explain in Section IV the required additions to Flocq to represent all IEEE-754 FP numbers. In Section V, we detail what modifications to CompCert were needed to handle FP arithmetic.

II. Semantics of Floating-Point Arithmetic

Starting from an algorithm using FP arithmetic, there is a long road till one gets some machine code running on a processor. First, there is the question of what the original algorithm is supposed to compute. Hopefully, the programmer has used the same semantic as the IEEE-754 standard for the operations, the goal being to get portable code and reproducible results. Then the programmer chooses a high-level programming language, since assembly languages would defeat the point of portability. Unfortunately, high-level language semantics are often rather vague with respect to FP operations, so as to account for as many execution environment as possible, even non-IEEE-754-compliant ones. So the programmer has to make some assumptions on how compilers will interpret the program. Unfortunately, compilers might have made different assumptions while still being compliant with the language standard, or they might just have gone the noncompliance way for the sake of execution speed (possibly controlled by a compilation flag). Finally, the operating system and various libraries play a role too, as they might modify the default behavior of FP units or emulate features not supported in hardware, *e.g.* subnormal numbers.

A. Java

Let us have an overview of some of the possible semantics through the lens of three major programming languages. Java, being a relatively recent language, started with the most specified description of FP arithmetic. It proposed two data types that match the `binary32` and `binary64` formats of IEEE-754. Moreover, arithmetic operators are mapped to the corresponding operators from IEEE-754, but rounding modes other than default are not supported, and neither are the

override of exceptional behaviors. The latter is hardly ever supported by languages so we will not focus on it in the remaining of this paper.

Unfortunately, a non-negligible part of the architectures the Java language was targeting had only access to *i387*-like FP units, which allow to set the precision of computation but not the allowed range of exponents. Thus, they behave as if they were working with exotic FP formats that have the usual IEEE-754 precision but an extended exponent range. On such architectures, complying with the Java semantics was therefore highly inefficient. As a consequence, the language later evolved and the FP semantics were relaxed to account for a potential extended exponent range:

> 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. (15.4 FP-strict expressions, Java SE 7)

The Java language specification, however, introduced a `strictfp` keyword for reinstating the early IEEE-754-compliant behavior.

B. C

The C language comes from a time where FP units were more exotic, so the wording of the standard is even more vague. Intermediate results can not only be computed with an extended range, they can also have an extended precision.

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

In fact, most compilers interpret the standard in an even more relaxed way: values of local variables that are not spilled to memory might preserve their extended range and precision.

Note that this optimization opportunity also applies to the use of a FMA operator for computing the expression $a \times b + c$, as the intermediate product is then performed with a much greater precision.

C. Fortran

The Fortran language gives even more leeway to compilers, allowing them to rewrite expressions as long as they do not change the value that would be obtained if the computations were to be infinitely-precise.

> Two expressions of a numeric type are mathematically equivalent if, for all possible values of their primaries, their mathematical values are equal. (7.1.5.2.4 Evaluation of numeric intrinsic operations, Fortran 2008)

The standard, however, forbids such transformations when they would violate the "integrity of parentheses". For instance, $(a+b)-a-b$ can be rewritten as 0, but $((a+b)-a)-b$ cannot, since it would break the integrity of the outer parentheses.

This allowance for assuming FP operations to be associative and distributive has unfortunately leaked to compilers for other languages, which do not even have the provision about preserving parentheses. For instance, the seemingly innocuous -Ofast option of GCC will enable this optimization for the sake of speed, at the expense of the conformance with the C standard.

D. Stricter Semantics

Fortunately, thanks to the IEEE-754 standard and to hardware makers willing to design strictly-compliant FP units [26], the situation is improving. It now becomes possible to specify programming languages without having to keep the FP semantic vague and obscure so that vastly incompatible architectures can be supported. Moreover, even if the original description of a language was purposely unhelpful, compilers can now document precisely how they interpret FP arithmetic for several architectures at once. In fact, in this work, we are going further: not only are we documenting what the expected semantic of our compiler is, but we are formally proving that the compiler follows it for all the architectures it supports.

III. Formally-Verified Compilation

As mentioned in Introduction, ordinary compilers sometimes *miscompile* source programs: starting with a correct source, they can produce executable machine code that crashes or computes the wrong results. Formally-verified compilers such as CompCert C come with a mathematical proof of *semantic preservation* that rules out all possibilities of miscompilation. Intuitively, the semantic preservation theorem says that the executable code produced by the compiler always executes as prescribed by the semantics of the source program.

Before proving a semantic preservation theorem, we must make its statement mathematically precise. This entails (1) specifying precisely the program transformations (compiler passes) performed by the compiler, and (2) giving mathematical semantics to the source and target languages of the compiler (in the case of CompCert, the CompCert C subset of ISO C90 and ARM/PowerPC/x86 assembly languages, respectively). The semantics used in CompCert associate *observable behaviors* to every program. Observable behaviors include normal termination, divergence (the program runs forever), and abnormal termination on an undefined behavior (such as an out-of-bounds array access). They also include traces of all input/output operations performed by the program: calls to I/O library functions (such as `printf`) and accesses to `volatile` memory locations.

Equipped with these formal semantics, we can state precisely the desired semantic preservation results. Here is one such result that is proved in CompCert:

Theorem 1 (Semantic preservation) *Let S be a source C program. Assume that S is free of undefined behaviors. Further 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.*

The statement of the theorem leaves two important degrees of freedom to the compiler: first, a C program can have several legal behaviors, owing to underspecification in expression evaluation order, and the compiler is allowed to pick any one of them; second, undefined C behaviors need not be preserved during compilation, as the compiler can optimize them away. This is not the only possible statement of semantic preservation: indeed, CompCert proves additional, stronger statements that imply the theorem above; but the bottom line is that the correctness of a compiler can be characterized in a mathematically-precise, yet intuitively understandable way, as soon as the semantics of the source and target languages are specified.

Concerning arithmetic operations in C and in assembly languages, their semantics are specified in terms of two Coq libraries, `Int` and `Float`, which provide Coq types for integer and FP values, and Coq functions for the basic arithmetic and logical operations, for conversions between these types, and for comparisons. The CompCert semantics map C language constructs to these basic operations, making fully precise a number of points that the C standard leaves to the discretion of the implementation. For example, the C standard does not specify the precision and range of the `float` and `double` types; CompCert C maps them to IEEE-754 `binary32` and `binary64` numbers, respectively. Likewise, C does not specify the precision of intermediate results during expression evaluation, requiring only that each FP operation is evaluated with a precision greater or equal to that of each operand; CompCert specifies that all intermediate results are computed in double precision. Finally, the C standard allows the compiler to "contract" several FP operations, such as a multiplication and an addition, in a single operation, such as FMA; the CompCert semantics disallow this contraction.[1]

These choices of implementation are somewhat arbitrary, but they provide programmers with a completely specified, easy-to-understand model of FP arithmetic, which is guaranteed to be implemented faithfully by the compiler. For example, as a consequence of this choice of C semantics and of the semantic preservation theorem, the x86 code generator of CompCert is guaranteed not to generate x87 FP instructions (which operate in extended precision and cannot implement IEEE-754 double precision exactly), generating SSE2 "scalar double" operations instead.

In early versions of CompCert (up to and including 1.11), the formalization of FP arithmetic is, however, less complete and less satisfactory than that of integer arithmetic. The `Int` library defines machine integers and their operations in a fully constructive manner, as Coq mathematical integers (type `Z`) modulo 2^{32}. In turn, Coq's mathematical integers are defined from first principles, essentially as lists of bits plus a sign. As a consequence of these constructive definitions, all the algebraic identities over machine integers used to justify optimizations and code generation idioms are proved correct in Coq, such as the equivalence between left-shift by $n \geq 0$ bits and multiplication by 2^n.

In contrast, in early versions of CompCert, the `Float` library was not constructed, but only axiomatized: the type of FP numbers is an abstract type, the arithmetic operations are just declared as functions but not realized, and the algebraic identities exploited during code generation are not proved to be true, but only asserted as axioms. (Section V-B shows examples of these identities.) Consequently, conformance to IEEE-754 could not be guaranteed, and the validity of the axioms could not be machine-checked. Moreover, this introduced a regrettable dependency on the host platform (the platform that runs the CompCert compiler), as we now explain.

The `Int` and `Float` Coq libraries are used not only to give semantics to the CompCert languages, modeling runtime computations, but also to specify the CompCert passes that perform numerical computations at compile-time. For instance, the constant propagation pass transforms the expression `2.0 * 3.0` into the constant `6.0` obtained by evaluating `Float.mul(2.0,3.0)` at compile-time. All the verified passes of the CompCert compiler are specified in executable style, as Coq recursive functions, from which an executable compiler is automatically generated by Coq's extraction mechanism, which produces equivalent OCaml code that is then compiled to an executable. For a fully-constructive library such as `Int`, this process produces an implementation of machine integers that is provably correct and entirely independent from the host platform, and can therefore safely be used during compilation.[2]

In contrast, for an axiomatized library such as the early versions of `Float`, there is no other choice than to map FP operations of the library onto those of the host, namely the FP operations provided by OCaml. However, OCaml's FP arithmetic is not guaranteed to implement IEEE-754 double precision: on the x86 architecture running in 32-bit mode, OCaml compiles FP operations to x87 machine instructions, resulting in excess precision and double-rounding issues. Likewise, conversion of decimal FP literals to `binary32` or `binary64` during lexing and parsing was achieved by calling into the corresponding OCaml library functions, which then call into the `strtod` and `strtof` C library functions, which are known to produce incorrectly-rounded results in several C standard libraries.

The discussion above points to a strong need for a fully-constructive Coq formalization of IEEE-754 arithmetic, providing implementations of FP arithmetic and conversions that are proved correct against the IEEE-754 standard, and can be invoked during compilation to perform constant propagation and other optimizations without being dependent on the host platform. We now describe how we extended the Flocq library to reach these goals.

[1] On target platforms that support it, CompCert makes the FMA instructions available as compiler built-in functions, but they must be explicitly used by the programmer.

[2] This is similar in spirit to GCC's use of exact, GMP-based integer arithmetic during compilation, to avoid dependencies on the integer types of its host platform.

IV. A Bit-Level Coq Formalization of IEEE-754 Binary Floating-Point Arithmetic

Flocq (Floats for Coq) is a formalization for the Coq system [8]. It provides a comprehensive library of theorems on a multi-radix multi-precision arithmetic. In particular, it encompasses radix-2 and 10 arithmetics, all the standard rounding modes, and it supports fixed- and floating-point arithmetics. The latter comes in two flavors depending on whether underflow is gradual or abrupt. The core of Flocq does not comply with IEEE-754 though, as it only sees FP numbers as subsets of real numbers, that is, it neither distinguishes the sign of zero nor handles special values. We therefore had to extend it to fully support IEEE-754 binary arithmetic. Moreover, this extension had to come with some effective computability so that it could be used in CompCert.

A. Formats and Numbers

Binary FP data with numeric values can be seen as rational numbers $m \cdot 2^e$, that is, pairs of integers (m, e). This is the generic representation that Flocq manipulates. Support for exceptional values is built upon this representation by using a dependent sum.

```
Inductive binary_float :=
  | B754_zero : bool -> binary_float
  | B754_infinity : bool -> binary_float
  | B754_nan : binary_float
  | B754_finite : forall (s : bool) (m : positive)
      (e : Z), bounded m e = true -> binary_float.
```

The above Coq code says that a value of type `binary_float` can be obtained in four different ways (depending on whether one wants a zero, an infinity, a *NaN*, or a finite number), and that, for instance, to build a finite number, one has to provide a boolean s, a positive integer m, an integer e, and a proof of the property `bounded m e = true`.

This property ensures that both m and e are integers that fit into the represented format. This format is described by two variables (precision and exponent range) that are implicit in the above definition. By setting these variables later, one gets specific instances of `binary_float`, for instance the traditional formats `binary32` and `binary64`. The `bounded` predicate also checks that m is normalized whenever e is not the minimal exponent. This constraint does not come from the IEEE-754 standard: any element of a FP cohort could be used, but it helps in some proofs to know that this element is unique.

In addition to finite numbers (both normal and subnormal), the `binary_float` type also supports signed zeros and signed infinities. Notice that there is a single datum *NaN* though, even if the IEEE-754 standard mandates numerous bit-level representations. We chose to abstract *NaN*s because the IEEE-754 standard underspecifies what happens to their sign and their payload. By ignoring them, we get a data type that encompasses all the compliant architectures.

The function `B2R` converts a `binary_float` value to a real number. For finite values, it returns $(-1)^s \times m \times 2^e$. Otherwise it returns zero. The sign of a value can be obtained by applying the `Bsign` function.

B. Executable Operations

Once the types are defined, the next step is to implement FP operators and prove their usual properties. An operator takes one or more `binary_float` inputs and a rounding mode, which tells which FP value to choose when the infinitely-precise result cannot be represented. The code of these operators always has the same structure. First, they perform a pattern matching on the inputs and handle all the special cases. Only finite numbers are left.

There are two different approaches for defining arithmetic operations. The first one is to have a `round` function that takes a rounding mode and a real number and return the closest FP number (according to the rounding mode m). For instance, the sum of two finite FP numbers can be defined as $a \oplus b = \texttt{round}(m, \texttt{B2R}(a) + \texttt{B2R}(b))$, assuming it does not overflow. The upside is that this operation trivially matches the IEEE-754 standard, since that is the way the standard defines arithmetic operation. The downside is that it depends on an abstract addition and an abstract rounding function, and thus it does not carry any computable content. As such, it cannot be used in a compiler that needs to perform FP operations to propagate constant values. This approach is used in the Pff [7] library and in the Flocq core library [8].

The second approach is to define arithmetic operators that actually perform computations on integers to construct a FP result. This time, the code of these operators can be used by a compiler for emulating FP operations, which is what we want. The downside is that, not only are these functions complicated to write, but there are no longer any guarantee that they are compliant with the IEEE-754 standard. So one also has to formally prove such theorems. This approach is used in the FP formalizations for ACL2 [5] and HOL Light [6].

We have mixed both approaches for our purpose: the second one offers effective computability, while stating and proving that the first one is equivalent provides concise specifications for our operations. Currently supported operations are opposite, addition, subtraction, multiplication, division, and square root. As an example, here is the correctness theorem for the FP multiplication `Bmult`.

Theorem 2 (Bmult_correct) *Given x and y two* `binary_float` *numbers, a rounding mode m, and denoting* $\texttt{round}(m, \texttt{B2R}(x) \times \texttt{B2R}(y))$ *by z, we have*

$$\begin{cases} \texttt{B2R}(\texttt{Bmult}(m,x,y)) = z & \text{if } |z| < 2^E, \\ \texttt{Bmult}(m,x,y) = & \\ \quad \texttt{overflow}(m, \texttt{Bsign}(x) \times \texttt{Bsign}(y)) & \text{otherwise.} \end{cases}$$

Note that Flocq's `round` function returns a real number that would be representable by a FP number if the format had no upper bound on the exponents. In particular, if the product overflows, then z is a number larger than the largest representable FP number $(1-2^{-p}) \cdot 2^E$. In that case, the `overflow` function is used to select the proper result depending on the rounding mode (either an infinity or the largest representable

number) according to the IEEE-754 standard.

Notice that the theorem works even for exceptional inputs since `B2R` maps them to zero. This makes it a bit simpler to apply. This simplicity is even more sensible for the square root, since it is proved to never overflow. Moreover, Coq's square root is a total function that returns zero for negative inputs, while the FP operator `Bsqrt` returns *NaN*. So the input x does not even have to be nonnegative for the theorem to hold.

Theorem 3 (Bsqrt_correct) *Given x a* `binary_float` *number and m a rounding mode, we have*

$$\texttt{B2R}(\texttt{Bsqrt}(m,x)) = \texttt{round}\left(m, \sqrt{\texttt{B2R}(x)}\right).$$

These correctness theorems specify fully only the case when both inputs of the operators are finite numbers. Indeed this is the difficult case. When one or both inputs are exceptional, no theorems are needed since one can simply execute the operators to recover their values.

C. Bit-Level Representation

Finally, the last part needed to build a compiler is the ability to go from and to the representation of FP numbers as integer words. We provide two functions for this purpose and a few theorems about them. Again, it is important that these functions are effectively computable.

The `binary_float_of_bits` function takes an integer, splits it into the three parts of a FP datum, looks whether the biased exponent is minimal (meaning the number is zero or subnormal), maximal (meaning infinity or *NaN*), or in between (meaning a normal number with an implicit bit), and constructs the resulting FP number of type `binary_float`. The `bits_of_binary_float` performs the converse operation. Note that it always returns the same *NaN* (all bits set to 1, except for the sign bit).

Both functions have been proved to be inverse of each other (except for *NaN*s) for bounded integers. This property also guarantees that we did not get these conversion functions too wrong. Indeed, it ensures that all the bits of the memory representation are accounted for and that there are no overlap between the three fields of the binary representation.

V. A Verified Compiler for Floating-Point Computations

We integrated the Coq formalization of IEEE-754 arithmetic described in Section IV into the CompCert compiler, version 1.12, effectively replacing the axiomatization of FP arithmetic used in earlier versions (see Section III) by a provably-correct, executable implementation.

As a first benefit, we obtain more precise semantic specifications for the source and target languages of CompCert. The semantics for the source CompCert C language now guarantees that FP arithmetic is performed as prescribed by IEEE-754, a guarantee that programmers can rely on. Symmetrically, the semantics for the target assembly languages (ARM, PowerPC, x86) now require that the hardware implements IEEE-754 correctly. Two of CompCert's target architectures have several FP instruction sets, with different characteristics. Our semantics only model the instructions actually generated by CompCert: for ARM, the scalar VFD instruction set, omitting vector instructions; and for x86, the scalar SSE2 instruction set, leaving aside vector instructions and x87 extended-precision instructions.

As another benefit of building on a Coq formalization of IEEE-754 arithmetic, we can now prove, as Coq theorems, the axioms about the `float` abstract type previously used by Compcert. As we explain in the following, these theorems prove the correctness of CompCert's compile-time handling of FP arithmetic: first, FP computations performed at compile-time by the compiler (such as FP literal parsing or constant propagation); second, the code generation strategies used to implement C's FP operations in terms of the instructions provided by the target architectures.

A. Verifying Compile-Time Computations

The CompCert compiler performs FP computations at different stages of compilation: (1) parsing of FP literals, (2) the constant propagation optimization, and (3) conversion of FP numbers to their bit-level representation when generating the final executable code. For conducting these operations, we need an implementation of FP arithmetic that is proved correct in Coq, executable via extraction from Coq to OCaml, and reasonably efficient. As shown in Section IV, our extension to the Flocq library provides such an implementation. In particular, the `bits_of_binary_float` function described in Section IV-C directly answers usage (3) above. We now discuss the use of Flocq for purposes (1) and (2).

Constant propagation is a basic but important optimization in compilers. It consists in evaluating, at compile-time, arithmetic and logical operations whose arguments can be statically determined. For instance, the Fast-Two-Sum example of the introduction is reduced to the printing of a single constant; no FP operations are performed by the executable code. For another example, consider the following C code fragment:

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

Combining constant propagation with function inlining, the body of function `g` is optimized into `return 0x1.5555555555555p-2`. Not only the division `1.0 / x` but also the conditional statement `x < 1.0` have been evaluated at compile-time. These evaluations are performed by the executable operations provided by the Flocq library, making them independent from the FP arithmetic of the host platform running the compiler, and guaranteeing that the constant propagation optimization preserves the semantics of the source program.

The evaluation of FP literals is delicate: literals are often written in decimal, requiring nontrivial conversion to IEEE-

754 binary format; moreover, correct rounding must be guaranteed. It is known, for example, that the `strtod` and `strtof` functions of the GNU C standard library incorrectly round the result in some corner cases. To avoid these pitfalls, we use a simple but correct Flocq-based algorithm for evaluating these literals.

In C, a FP literal consists of an integral part, a fractional part, an exponent part, and a precision suffix (which indicates at which precision the literal should be evaluated). Each of these parts can be omitted, in which case 0 can be used as a default value (this operation is done in an early stage of parsing in our compiler). The whole number can be written either in decimal or in hexadecimal. The exponent is given as a power of 2 if hexadecimal is used or as a power of 10 if decimal is used. To summarize, a literal number always has the form $\mathtt{I.F} \times b^{\mathtt{E}}$ with $b = 2$ or $b = 10$.

The first part of our algorithm consists in shifting the point to the right, while modifying the exponent in order to transfer the fractional part `E` into the integral part `I`. Then, it parses both the exponent and the new integral part as arbitrary-precision integers. The last part consists in actually evaluating the FP number, using Flocq with the precision specified by the precision suffix. When $\mathtt{E} \geq 0$, we compute $\mathtt{I} \times b^{\mathtt{E}}$ using exact integer arithmetic, then round the result to the nearest representable FP number. When $\mathtt{E} < 0$, we first compute $b^{-\mathtt{E}}$ using exact integer arithmetic, then perform the FP division $\circ(\mathtt{I}/b^{-\mathtt{E}})$, using the proved division of Flocq.

It is clear that the result is evaluated as in the reals before being rounded at the very last step. We believe this implementation is one of the simplest one could give, and we would use it as a specification to a more complicated algorithm if better performance is needed.

B. Verifying Code Generation for Floating-Point Operations

Most FP operations of the C language map directly to hardware-implemented instructions of the target platforms. However, some operations, such as certain comparisons and conversions between integers and FP numbers, are not directly supported by some target platforms, forcing the compiler to implement these operations by sometimes convoluted combinations of other instructions. The correctness of these code generation strategies depends on the validity of algebraic identities over FP operations, identities that we were able to verify in Coq using the theorems provided by Flocq.

A first example is FP comparisons on the PowerPC architecture. The PowerPC provides a `fcmp` instruction that produces 4 bits of output: "less than", "equal", "greater", and "uncomparable", and conditional branch instructions that test any one of these bits. To compile a large inequality test such as "less than or equal", CompCert produces code that performs the logical "or" of the "less than" and "equal" bits, then conditionally branches on the resulting bit. Semantically, this is justified by the identity $(x \leq y) \equiv (x < y) \vee (x = y)$, which holds for any two FP numbers x and y.

Another example is conversions between integers and FP numbers, which come in 4 variants, depending on the direction of the conversion (from an integer or to an integer) and on the type of the integer: either signed, in the range $[-2^{31}, 2^{31})$, or unsigned, in the range $[0, 2^{32})$. Of the 3 CompCert target platforms, only ARM provides hardware implementations of all 4 conversions. The x86-SSE2 instruction set only provides conversions to and from signed integers, requiring the unsigned integer conversions to be implemented by case analysis: if `n` is an unsigned 32-bit integer variable, the C conversion `(double) n` is compiled like the C conditional expression

```
n < 0x80000000
? (double)((int) n)
: (double)((int)(n - 0x80000000)) + 0x1.p31
```

Likewise, if `d` is a `binary64` variable, the C conversion `(unsigned int) d` is compiled like

```
d < 0x1.p31 ? (int) d
            : (int)(d - 0x1.p31) + 0x8000000
```

We proved the correctness of this code generation strategy, using the fact that all 32-bit signed and unsigned integers are exactly representable as `binary64`, and that conversions from FP numbers to integers are undefined if the argument falls outside the range of the destination integer type.

The PowerPC 32-bit architecture is even more problematic, as the only conversion it implements in hardware is `binary64` to signed integer. Conversion to an unsigned integer is implemented as shown above. Conversions from integers to FP numbers are synthesized via bit-level manipulations over the `binary64` numbers, as suggested by IBM [27]. If `n` is an unsigned 32-bit integer, the conversion `(double) n` is compiled as

```
fmake(0x43000000, n) - fmake(0x43000000, 0)
```

where `fmake(hi,lo)` is a compiler built-in function that returns a double whose 64-bit binary representation is the concatenation of the 32 bits of integer `hi` followed by the 32 bits of integer `lo`. If, instead, `n` is a signed integer, `(double) n` is compiled as

```
fmake(0x43000000, n ^ 0x80000000)
          - fmake(0x43000000, 0x80000000)
```

The correctness of this implementation technique is far from obvious. Taking the unsigned case as example, we first note that `fmake(0x43000000, n)` is equal to $2^{52} + n$, and that `fmake(0x43000000, 0)` is 2^{52}. We then prove that the `binary64` subtraction between these two numbers is exact and produces the `binary64` number equal to n. Mechanizing this proof in Coq brings much confidence in this implementation.

VI. Conclusions

In this article, we have presented a formally-verified compiler that supports FP computations. Producing such a compiler required us to define the FP semantics for the C language and for the target architectures, and to prove that the compiler preserves the semantics between a C program and the produced executable code. Flocq has been extended with a formalization of the IEEE-754 standard; this formalization is used by CompCert to define the semantics, parse literal

FP constants, and perform constant propagation at compile-time. This development has been integrated into version 1.12 of CompCert available at http://compcert.inria.fr/.

This approach gives a correct and predictable compiler that conforms to the IEEE-754 standard. The actual interpretation of FP operations can be seen in the `Float` module of CompCert; one does not have to wade through all the optimization passes to understand what happens to them, since their semantics is provably preserved. Another advantage is that having strict semantics paves the way to simpler, more precise, and even verified, static analyzers.

For the sake of completeness, one should note that CompCert's formal semantics does not support certain features of the IEEE-754 standard. First, CompCert does not know about directed rounding modes and assumes that all the FP operations are performed with the default rounding mode. As a consequence, on architectures that have dynamic rounding modes, changing the mode prevents CompCert's semantics from being preserved. For instance, constant propagation might give a different result from actual execution. CompCert could be extended to support a dynamic mode, *e.g.* by representing it as a pseudo global variable. Constant propagation would then only happen if either the rounding mode is statically known, or if the result would be the same whatever the mode.

Another peculiarity of CompCert is that all the intermediate computations are performed in double precision, as allowed by the C standard. It is still possible to achieve `binary32` computations by following each operation by a store to a `binary32` variable. Double rounding occurs but is known to produce the correctly-rounded result [28].

The integration of Flocq and CompCert opens the way to adding more optimizations specific to FP arithmetic, and to prove them correct. FP identities such as $x - 0.0 \equiv x$ or $x \times 2 \equiv x + x$ or $x/2^n \equiv x \times 2^{-n}$ can be exploited to generate shorter or cheaper instruction sequences. For the semantic preservation theorem to hold, however, only identities that hold for all representable FP numbers can be used, and there are few of them. Other tempting simplifications are incorrect for some values of their arguments: for example, $x + 0.0 \equiv x$ does not hold if $x = -0.0$, and Brisebarre *et al*'s technique to replace a FP division by a constant with a multiplication and an FMA [29] is not always correct for subnormal arguments. The only way to exploit these simplifications while preserving semantics would be to apply them conditionally, based on the results of a static analysis (such as FP interval analysis) that can exclude the problematic cases.

The problem is even more acute for aggressive loop optimizations such as vectorization, which often entail reassociating FP operations, and cannot be guaranteed to preserve semantics except in very special cases. We conclude that the compiler is probably the wrong place to perform aggressive program transformations over FP operations, because it lacks much of the information necessary for this endeavor. Automatic code generation tools, however, are in a more favorable position to preserve or improve precision by reassociation and other aggressive transformations [30].

REFERENCES

[1] P. H. Sterbenz, *Floating point computation*. Prentice Hall, 1974.

[2] T. J. Dekker, "A floating point technique for extending the available precision," *Numerische Mathematik*, vol. 18, no. 3, pp. 224–242, 1971.

[3] Microprocessor Standards Subcommittee, "IEEE Standard for Floating-Point Arithmetic," *IEEE Std. 754-2008*, pp. 1–58, Aug. 2008.

[4] V. A. Carreño and P. S. Miner, "Specification of the IEEE-854 floating-point standard in HOL and PVS," in *HOL95: 8th International Workshop on Higher-Order Logic Theorem Proving and Its Applications*, Aspen Grove, UT, Sep. 1995.

[5] D. M. Russinoff, "A mechanically checked proof of IEEE compliance of the floating point multiplication, division and square root algorithms of the AMD-K7 processor," *LMS Journal of Computation and Mathematics*, vol. 1, pp. 148–200, 1998.

[6] J. Harrison, "Formal verification of floating point trigonometric functions," in *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, Austin, Texas, 2000, pp. 217–233.

[7] S. Boldo, "Preuves formelles en arithmétiques à virgule flottante," Ph.D. dissertation, École Normale Supérieure de Lyon, 2004.

[8] S. Boldo and G. Melquiond, "Flocq: A unified library for proving floating-point algorithms in Coq," in *Proceedings of the 20th IEEE Symposium on Computer Arithmetic*, E. Antelo, D. Hough, and P. Ienne, Eds., Tübingen, Germany, 2011, pp. 243–252.

[9] D. Monniaux, "The pitfalls of verifying floating-point computations," *TOPLAS*, vol. 30, no. 3, p. 12, May 2008.

[10] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres, *Handbook of Floating-Point Arithmetic*. Birkhäuser, 2010.

[11] ISO, "International standard ISO/IEC 9899:2011, Programming languages – C," 2011.

[12] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011*. ACM Press, 2011, pp. 283–294.

[13] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, "The ASTRÉE analyzer," in *ESOP*, ser. Lecture Notes in Computer Science, no. 3444, 2005, pp. 21–30.

[14] D. Monniaux, "Analyse statique : de la théorie à la pratique," Habilitation to direct research, Université Joseph Fourier, Grenoble, France, Jun. 2009.

[15] D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and F. Védrine, "Towards an industrial use of FLUCTUAT on safety-critical avionics software." in *FMICS*, ser. LNCS, vol. 5825. Springer, 2009, pp. 53–69.

[16] G. T. Leavens, "Not a number of floating point problems," *Journal of Object Technology*, vol. 5, no. 2, pp. 75–83, 2006.

[17] S. Boldo and J.-C. Filliâtre, "Formal Verification of Floating-Point Programs," in *18th IEEE International Symposium on Computer Arithmetic*, Montpellier, France, June 2007, pp. 187–194.

[18] A. Ayad and C. Marché, "Multi-prover verification of floating-point programs," in *Fifth International Joint Conference on Automated Reasoning*, ser. Lecture Notes in Artificial Intelligence, J. Giesl and R. Hähnle, Eds. Edinburgh, Scotland: Springer, Jul. 2010.

[19] S. Boldo and T. M. T. Nguyen, "Proofs of numerical programs when the compiler optimizes," *Innovations in Systems and Software Engineering*, vol. 7, pp. 151–160, 2011.

[20] T. M. T. Nguyen and C. Marché, "Hardware-dependent proofs of numerical programs," in *Certified Programs and Proofs*, ser. Lecture Notes in Computer Science, J.-P. Jouannaud and Z. Shao, Eds. Springer, Dec. 2011.

[21] R. Milner and R. Weyhrauch, "Proving compiler correctness in a mechanized logic," in *Proc. 7th Annual Machine Intelligence Workshop*, ser. Machine Intelligence, B. Meltzer and D. Michie, Eds., vol. 7. Edinburgh University Press, 1972, pp. 51–72.

[22] J. S. Moore, "A mechanically verified language implementation," *Journal of Automated Reasoning*, vol. 5, no. 4, pp. 461–492, 1989.

[23] G. Li, S. Owens, and K. Slind, "Structure of a proof-producing compiler for a subset of higher order logic," in *Proceedings of the 16th European conference on Programming*, ser. ESOP'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 205–219.

[24] M. O. Myreen, "Formal verification of machine-code programs," Ph.D. dissertation, University of Cambridge, 2008.

[25] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, 2009.
[26] J. Nickolls and W. Dally, "The GPU computing era," *IEEE Micro*, vol. 30, no. 2, pp. 56–69, 2010.
[27] IBM, *The PowerPC Compiler Writer's Guide*. Warthman Associates, 1996.
[28] S. A. Figueroa, "When is double rounding innocuous?" *SIGNUM Newsletter*, vol. 30, no. 3, pp. 21–26, 1995.
[29] N. Brisebarre, J.-M. Muller, and S. K. Raina, "Accelerating correctly rounded floating-point division when the divisor is known in advance," *IEEE Trans. Computers*, vol. 53, no. 8, pp. 1069–1072, 2004.
[30] A. Ioualalen and M. Martel, "A new abstract domain for the representation of mathematically equivalent expressions," in *Static Analysis - 19th International Symposium, SAS 2012*, ser. Lecture Notes in Computer Science, vol. 7460. Springer, 2012, pp. 75–93.