Verified Compilation of Floating-Point Computations

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

Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

Verified Compilation of Floating-Point Computations

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

the date of receipt and acceptance should be inserted later

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 easier and flourishing. Unfortunately, this is not sufficient to guarantee the final result of a program, as several other actors are involved: programming language, compiler, and 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 (a large subset of ISO C99) 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.

Keywords Floating-point arithmetic · compiler verification · semantic preservation · CompCert · Coq

1 Introduction

Use and study of floating-point (FP) arithmetic have intensified since the 70s [35, 11]. At that time, computations were not standardized and, due to the differences between architectures, the use of the same source program in different contexts gave different results. Thanks to the IEEE-754 standard of 1985 and its revision in 2008 [25], the situation has improved, since reproducibility was a key concept.

S. Boldo · G. Melquiond
Inria, LRI, CNRS UMR 8623, Université Paris-Sud, Bât 650 (PCRI), 91405 Orsay, France
E-mail: sylvie.boldo@inria.fr, guillaume.melquiond@inria.fr

J.-H. Jourdan · X. Leroy
Inria Paris-Rocquencourt, Domaine de Voluceau, BP 105, 78153 Le Chesnay, France
E-mail: jacques-henri.jourdan@inria.fr, xavier.leroy@inria.fr

Indeed, each basic operation is guaranteed 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 extended to formal proofs of algorithms or hardware components: in PVS [8], in ACL2 [34], in HOL Light [17] and in Coq [2,5]. The basic axiom for algorithms and the basic goal for hardware components was that all the operations are correctly rounded.

Unfortunately, the processor architecture is not the only party responsible for the computed results: the programming language and the compiler used also stand accused. We will focus on the compiler, as it can deviate from 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:

```
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 0;
}
```

Experts may have recognized the Fast-Two-Sum algorithm [11] which computes the round-off error of a FP addition $a+b$ by evaluating $((a+b)-a)-b$ for $|a| \geq |b|$. This very simple program compiled with GCC 4.9.1 gives three different answers on an x86 architecture depending on the instruction set and the chosen level of optimization.

Compilation options	Program result
`-O0` (x86-32)	`-0x1p-78`
`-O0` (x86-64)	`0x1.fffffp-54`
`-O1`, `-O2`, `-O3`	`0x1.fffffp-54`
`-Ofast`	`0x0p+0`

How can we explain the various results? For the first three rows, 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 for the 32-bit instruction set, all the computations are performed with extended precision and rounded in double precision only once at the end. With the `-O0` optimization for the 64-bit instruction set, all the computations are performed with double precision. With `-O1` and higher, the intermediate value $(1+y)-1$ is precomputed by the compiler as if performed with double precision; the program effectively computes only the last subtraction and the result does not depend on the instruction set. 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, let us see how a C compiler may implement the FP evaluation of `a * b + c * d`. If $\oplus$ and $\otimes$ denote the FP addition and multiplication, and fma the FP FMA, there are three ways of compiling it: $\mathrm{fma}(a, b, c \otimes d)$, $\mathrm{fma}(c, d, a \otimes b)$, and $(a \otimes b) \oplus (c \otimes d)$. These sequences of operations may give different results, since the round-off errors occur at different points. Many more examples of strange FP behaviors can be found in [27, 29].

As surprising as it may seem, all the discrepancies described so far are allowed by the ISO C standard [20], 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:[1]

```
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;
}
```

For an x86 32-bit target 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* [36] 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. For the automatic analysis of C programs, a successful approach is based on abstract interpretation, and tools include Astrée [10] and Fluctuat [12]. Another method to specify and prove behavioral properties of FP programs is deductive verification system: specification languages have to take into account FP arithmetic. This has been done for Java in JML [21], for C in ACSL [3, 1]. However, all these tools follow

[1] "Optimized code gives strange floating point results", `http://gcc.gnu.org/bugzilla/show_bug.cgi?id=323`

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 [6]. Another approach is to analyze the assembly code to get all the precision information [31].

Our approach is different: rather than trying to account for all the changes a compiler may have silently introduced in a FP program, we focus on getting a correct and predictable compiler that supports FP arithmetic. Concerning compilers and how to make them more trustworthy, Milner and Weyhrauch [26] were the first to mechanically prove the correctness of a compiler, although for a very simple language of expressions. Moore [28] extended this approach to an implementation of the Piton programming language. Li *et al* [24] showed that one can compile programs with proof, directly from the logic of the HOL4 theorem prover. A year later, Myreen [30] made contributions both to approaches for verification of programs and methods for automatically constructing correct code.

To build our compiler, we started from CompCert [22], a formally-verified compiler described in Section 4 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 [5] 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 2 the semantics of FP arithmetic in various programming languages. Section 3 outlines a formal semantics for FP computations in the C language and restrictions that provide stronger guarantees. In Section 4, we describe the CompCert compiler and its verification. We explain in Section 5 the required additions to Flocq to represent all IEEE-754 FP numbers. Sections 6 and 7 detail CompCert's code generation and optimizations for FP arithmetic and how they were proved correct. Section 8 concludes with additional perspectives.

2 Semantics of Floating-Point Arithmetic

As mentioned in the introduction, the IEEE-754 standard mandates that FP operations be performed as if the computation was done with infinite precision and then rounded. Thus, one can introduce a rounding operator $\circ$ that takes a real number and changedreturns the closest FP number, for some definition of closest that depends on the rounding mode and the destination format. This operator is such that the FP addition $x \oplus y$ gives the same result as $\circ(x+y)$ for non-exceptional values x and y. As such, we will use both notations indiscriminately to denote FP addition from now on, and similarly for other FP operators. When there is a possible ambiguity on the destination format, it will be specified as $\circ_{format}$.

Let us now see what it means for an algorithm to use FP arithmetic, since there is a long road until 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 semantics 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 environments as possible, even non-IEEE-754-compliant ones. So the programmer has to make some assumptions on how compilers will interpret the program. Unfortunately, different compilers may take different paths while still being compliant with the language standard, or they might depart from the standard 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.

2.1 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 x87-like FP units, which make it possible 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.

2.2 C

The C language comes from a time where FP units were more exotic, so the wording of the standard leaves much more liberty to the compiler. 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, C11)

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.

While Annex F of the C standard allows a compiler to advertise compliance with IEEE-754 FP arithmetic if it supports a specified set of features, none of these features reduces the leeway compilers have in choosing intermediate formats. Moreover, features of Annex F are optional anyway.

2.3 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.

2.4 Stricter Semantics

Fortunately, thanks to the IEEE-754 standard and to hardware makers willing to design strictly-compliant FP units [32], the situation is improving. It is now 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 (Section 3), but we are formally proving that the compiler follows it for all the architectures it supports (Sections 4 and following).

3 Formalizing C Floating-Point Semantics

We now consider the problem of formally specifying the semantics of FP computations in C programs. Since C is a large language and existing C semantics are very

$$\frac{\varphi \geq F(\tau) \qquad n \in \varphi}{\Gamma \vdash [n, \tau, f] : \tau} \qquad \Gamma \vdash x : \Gamma(x) \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash (x = e) : \Gamma(x)}$$

$$\frac{\Gamma \vdash e : \tau'}{\Gamma \vdash (\tau)\ e : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \diamond e_2 : \max(\tau_1, \tau_2)}$$

Fig. 1 Typing rules for μFP

complex, we study a tiny language of C-like expressions, nicknamed μFP. The idea is to expose the issues that arise due to the possible intermediate extended range and precision. The μFP language is sufficient to demonstrate both these issues and how to formally describe what the compiler is allowed to do and which strategy it may choose.

Types:	τ ::=	`float` | `double` | `long double`	
FP formats:	φ	(processor-dependent)	
Expressions:	e ::=	x	variable
	|	$[n, \tau, \varphi]$	FP value
	|	$(\tau)\ e$	conversion (cast)
	|	$e_1 \diamond e_2$	arithmetic operation

Expressions are formed over variables x and FP values, and include conversions $(\tau)\ e$ to a type τ and arithmetic operations $e_1 \diamond e_2$ where $\diamond$ ranges over addition, subtraction, multiplication, and division. FP values $[n, \tau, \varphi]$ represent both FP literals as they occur in the source expression, and intermediate results of computations. They consist of a FP number n, a static type τ, and a FP format φ that represents the precision and range used for storing n. We shall discuss FP formats shortly, but the important point to note is that the format φ can be more precise than implied by the static type τ, therefore materializing the "excess precision" allowed by the C standards.

Types τ include the three standard C types `float`, `double`, and `long double`, linearly ordered by subtyping: `float` < `double` < `long double`. Every variable x has a static type written $\Gamma(x)$. The typing rules for μFP are summarized in Fig. 1. The type of a given expression is uniquely determined. Moreover, for arithmetic operations involving arguments of different types, the type of the result is the greater of the types of the arguments, as specified by the "usual arithmetic conversions" of C [20, §6.3.1.8].

The μFP language is parameterized by a set of FP formats, which are all the formats supported by the target processor architecture. For the x86 architecture, for example, we have three formats: the IEEE-754 standard formats `binary32` and `binary64`, plus the extended 80-bit format `extended80`. Each format φ specifies a precision and range, and can be viewed as the set of numbers that are representable exactly within this format. Formats are naturally ordered by inclusion: $\varphi_1 \leq \varphi_2$ means that any number exactly representable with φ_1 is also exactly representable with φ_2. In the x86 example, we have `binary32` < `binary64` < `extended80`.

FP formats and C types may be related in nonobvious ways. The application binary interface for the target platform specifies a mapping F from C types to

$$\frac{\Gamma, s \vdash e \to e'}{\Gamma, s \vdash E[e] \to E[e']} \quad \text{(Ctx)} \qquad \Gamma, s \vdash x \to [s(x), \Gamma(x), F(\Gamma(x))] \quad \text{(Var)}$$

$$\frac{n' = [\![\varphi \to F(\tau')]\!](n)}{\Gamma, s \vdash (\tau')\ [n, \tau, \varphi] \to [n', \tau', F(\tau')]} \quad \text{(Conv)}$$

$$\frac{\tau = \max(\tau_1, \tau_2) \qquad n = [\![\diamond]\!]_\varphi(n_1, n_2)}{\Gamma, s \vdash [n_1, \tau_1, \varphi] \diamond [n_2, \tau_2, \varphi] \to [n, \tau, \varphi]} \quad \text{(Arith)}$$

$$\frac{\varphi' \geq F(\tau) \qquad n' = [\![\varphi \to \varphi']\!](n)}{\Gamma, s \vdash [n, \tau, \varphi] \to [n', \tau, \varphi']} \quad \text{(Reformat)}$$

Fig. 2 Operational semantics for μFP

formats. In the x86 case, we have, for example:

$$F(\texttt{float}) = \texttt{binary32} \quad F(\texttt{double}) = \texttt{binary64} \quad F(\texttt{long double}) = \texttt{extended80}$$

However, $F(\tau)$ only specifies the minimal precision that the value of an expression of type τ possesses: extra precision can also be maintained during computations. For example, on the x86 processor, intermediate results kept in the x87 FP stack always have precision `extended80`, regardless of their static types. The crucial invariant that our semantics must maintain is, therefore: any FP value $[n, \tau, \varphi]$ is such that $n \in \varphi$ and $\varphi \geq F(\tau)$.

3.1 The Nondeterministic Semantics

We are now ready to specify a small-step operational semantics for our tiny language, as a reduction relation $\Gamma, s \vdash e \to e'$ between expressions, parameterized by a typing environment Γ and a store s. A store s maps variables x to the values of FP numbers n. In accordance with the C standards, we enforce that FP values are always stored with the minimal precision implied by their static types; in other words, that extra precision is discarded at assignment-time. Therefore, the format of the number $s(x)$ is implicitly $F(\Gamma(x))$ and need not be recorded in the store.

The reduction rules are given in Fig. 2. The context rule (Ctx) enables reduction in any subexpression of the expression under consideration. Contexts E are expressions with a hole $[\,]$:

$$\text{Contexts:} \quad E ::= [\,] \mid (\tau)\ E \mid E \diamond e_2 \mid e_1 \diamond E$$

Rule (Var) looks up the current value of variable x in the store. As discussed above, the FP format of this value is always $F(\Gamma(x))$, without any excess precision.

Rule (Conv) discards any excess precision in the value n being converted, replacing it by $[\![\varphi \to F(\tau')]\!](n)$, where φ is the original format of n and τ' is the destination type of the conversion.

The notation $[\![\varphi \to \varphi']\!]$ denotes the function that converts an FP value with format φ to format φ'. If $\varphi' < \varphi$, appropriate rounding is performed. (For simplicity, we do not model the possibility of changing rounding modes at run-time, and

instead assume that a fixed rounding mode is used throughout.) If $\varphi' \geq \varphi$, the conversion is exact: the FP number is not modified, only (possibly) its machine representation. In particular, we have the following equalities:

$$\begin{array}{ll} [\![\varphi \to \varphi']\!](n) = n & \text{if } n \in \varphi' \\ [\![\varphi \to \varphi']\!](n) = n & \text{if } n \in \varphi \text{ and } \varphi' \geq \varphi \end{array}$$

Rule (Arith) describes the evaluation of an arithmetic operation. Both arguments can have different types, but are required to share a common format φ. The computation is, then, performed at format φ. The notation $[\![\diamond]\!]_\varphi$ stands for the IEEE-754 arithmetic operation corresponding to $\diamond$ at format φ. For example, $[\![+]\!]_{\texttt{binary64}}$ is the addition of two `binary64` numbers using the current rounding mode.

The semantics derives much of its power from the (Reformat) rule, which makes it possible to nondeterministically change the format of an FP value without changing its type. The format can either increase in precision or decrease in precision, as long as it stays at or above the minimal precision $F(\tau)$ specified by the type τ of the value.

The first use of (Reformat) is to satisfy the precondition of rule (Arith) when the two arguments of an arithmetic operation have different types. For example, assuming `x` has type `float` and `y` has type `double`, we have the following reduction sequence:

$$\begin{array}{lll} \Gamma, s \vdash (\texttt{x} + \texttt{y}) & \to ([s(\texttt{x}), \texttt{float}, \texttt{binary32}] + \texttt{y}) & \text{(Ctx, Var)} \\ & \to ([s(\texttt{x}), \texttt{float}, \texttt{binary32}] + [s(\texttt{y}), \texttt{double}, \texttt{binary64}]) & \text{(Ctx, Var)} \\ & \to ([s(\texttt{x}), \texttt{float}, \texttt{binary64}] + [s(\texttt{y}), \texttt{double}, \texttt{binary64}]) & \text{(Ctx, Reformat)} \\ & \to [[\![+]\!]_{\texttt{binary64}}(s(\texttt{x}), s(\texttt{y})), \texttt{double}, \texttt{binary64}] & \text{(Arith)} \end{array}$$

It is necessary to use (Reformat) to expand the value of `x` to `binary64` format, in order to perform the addition at this format.

Rule (Reformat) also makes it possible for the C compiler to introduce or discard excess precision at its discretion. Continuing the example above, (Reformat) could have been used twice to bring the values of `x` and `y` to format `extended80`. Then, the addition would be performed in extended precision, thus introducing excess precision compared with that implied by the static types. Immediately after or at any later time, rule (Reformat) can be invoked to discard this excess precision, rounding the result of the addition to `binary64` format. This is exactly what many compilers for x86 do when they spill an intermediate FP result computed in extended precision in the x87 unit to a 64-bit stack location of `binary64` format.

Finally, our semantics can also account for the contraction of several arithmetic operations into one compound operation that rounds only once, such as fused multiply-add (FMA). For this, we need to introduce a special format called `exact` that stands for infinite precision and range. Then, an expression such as `x * y + z` can be evaluated following the FMA strategy by first reformatting the values of `x`, `y`, and `z`, to `exact` format, performing the multiplication and the addition at `exact` format (hence, without rounding), and finally reformatting the result to a finite format, rounding it in the process. A parallel can be made with the "exact" rounding mode described in [6], also to handle the FMA.

We believe that the semantics above faithfully captures the intent of the ISO C standards concerning excess precision in FP computations, namely that it can be introduced and discarded at any time at the compiler's whim, the only guarantee

being that excess precision is fully discarded by conversions. At the same time, this semantics demonstrates that the evaluation of FP expressions in C is highly nondeterministic, leaving programmers with little control on how and how many times intermediate FP results are rounded. As argued in the introduction, such nondeterminism tremendously complicates reasoning on numerical algorithms and verifying their C implementation.

3.2 Two Deterministic Semantics

We now consider two refinements of the semantics from Section 3.1 that are fully deterministic and that, in our opinion, provide a better basis for reasoning and verification.

The minimal precision semantics The first refinement consists in prohibiting any use of excess precision. In other words, an FP value with static type τ is always represented at the corresponding format $F(\tau)$: all values are of the form $[n, \tau, F(\tau)]$.

In the semantics, rules (Ctx), (Var), and (Conv) are unchanged. Rules (Arith) and (Reformat) are suppressed and replaced by the single combined rule below:

$$\frac{\begin{array}{c}\tau = \max(\tau_1, \tau_2) \\ n = [\![\diamond]\!]_{F(\tau)}([\![F(\tau_1) \to F(\tau)]\!](n_1), [\![F(\tau_2) \to F(\tau)]\!](n_2))\end{array}}{\Gamma, s \vdash ([n_1, \tau_1, F(\tau_1)] \diamond [n_2, \tau_2, F(\tau_2)]) \to [n, \tau, F(\tau)]} \quad \text{(Arith-min)}$$

The maximal precision semantics As an alternative to the first refinement above, we can also elect to perform all FP arithmetic at the maximal precision Φ supported by the target processor. For example, for the x86 processor with x87 FP arithmetic, Φ is `extended80`, all arithmetic is performed via x87 extended-precision operations, and intermediate results are spilled to 80-bit stack locations so as not to lose precision. For ARM, we would likewise take $\Phi =$ `binary64` and perform all FP arithmetic in double precision.

This alternate semantics is captured by keeping rules (Ctx), (Var), and (Conv), and replacing rules (Arith) and (Reformat) by the single combined rule below:

$$\frac{\tau = \max(\tau_1, \tau_2) \qquad n = [\![\diamond]\!]_{\varphi}([\![F(\tau_1) \to \Phi]\!](n_1), [\![F(\tau_2) \to \Phi]\!](n_2))}{\Gamma, s \vdash ([n_1, \tau_1, F(\tau_1)] \diamond [n_2, \tau_2, F(\tau_2)]) \to [n, \tau, \Phi]} \quad \text{(Arith-max)}$$

Comparing the two deterministic semantics Both refinements above succeed in uniquely defining the FP evaluation of a C-like expression. Moreover, both are simple enough to be easily understood by programmers and analyzed by formal verification tools.

Concerning implementability, the "minimal precision" approach requires the target processor to faithfully implement FP arithmetic operations at every FP format used: `binary32`, `binary64`, and the extended format possibly corresponding to `long double`. This is the case for all modern processors, using the SSE2 FP unit in the case of x86, but not for old x86 processors that lack SSE2.

Concerning expressiveness, all numerical computations that can be expressed in the "maximal" semantics can also be expressed in the "minimal" semantics: to force an FP operation to use maximal precision, it suffices for the programmer to insert an explicit conversion to a large enough FP type on one of the arguments.

Conversely, there are computations that cannot, in general, be expressed in the "maximal" semantics. Assume that `x` and `y` have type τ and that the programmer wants to compute `x + y` exactly rounded at format $F(\tau)$. With the maximal semantics, `x + y` will be computed at the possibly higher precision Φ. Writing $(\tau)(\mathtt{x}+\mathtt{y})$ does not suffice, in general, to recover the expected result, because double rounding occurs: the exact sum is first rounded to format Φ, then to format $F(\tau)$. This can lead to incorrect results in the case $\tau =$ `double` and $\Phi =$ `extended80`, for example. For instance, the `binary64` sum of $\mathtt{x} = 1$ and $\mathtt{y} = 2^{-53} + 2^{-78}$ is equal to $1 + 2^{-52}$, yet $(\mathtt{double})(\mathtt{x} + \mathtt{y})$ is equal to 1 when $\Phi =$ `extended80`.

However, if the maximal precision Φ is `binary64`, such double rounding is equivalent to single rounding. This is obvious in the case $F(\tau) =$ `binary64`. The only other case to consider is $F(\tau) =$ `binary32`. As Figueroa [13] proved, for all FP values a and b, we have

$$\circ_{\mathtt{binary32}}\left(\circ_{\mathtt{binary64}}(a+b)\right) = \circ_{\mathtt{binary32}}(a+b),$$

and similarly for subtraction, multiplication, division, and square root. As the intermediate precision is more than twice the output precision, double rounding is innocuous and produces the correctly-rounded result. Therefore, the maximal precision approach with $\Phi =$ `binary64` is as expressive as the minimal precision approach.

3.3 The CompCert C Semantics for Floating-Point Computations

Starting with version 2.4, the CompCert C semantics for floating-point computations closely follows the "minimal precision" semantics formalized on a toy language in the current section.[2] This semantics, extended to the whole C language, can be summarized as follows:

1. The `float` and `double` types are mapped to IEEE-754 `binary32` and `binary64` numbers, respectively. Extended-precision FP numbers are not supported: the `long double` type is either unsupported or mapped to `binary64`, depending on a compiler option.
2. Conversions to a FP type, either explicit ("type casts") or implicit (at assignment, parameter passing, and function return), always round the FP value to the given FP type, discarding excess precision.
3. Reassociation of FP operations, or "contraction" of several operations into one (*e.g.* a multiply and an add being contracted into a fused multiply-add) are

[2] Earlier versions of CompCert used the "maximal precision" semantics, with all intermediate FP results computed in `binary64` format. The main motivation for this earlier choice was to reduce the number of FP arithmetic operations that must be modeled. However, it resulted in some inefficiencies in the machine code generated for programs that use single precision intensively. The switch to the "minimal precision" semantics enables better code to be generated in this case. It also makes it possible, in the future, to support target architectures that support only `binary32` arithmetic.

prohibited. (On target platforms that support them, CompCert makes FMA instructions available as compiler built-in functions, but they must be explicitly used by the programmer.)
4. All intermediate FP results in expressions are computed with the precision that corresponds to their static C types, that is, the greater of the precisions of their arguments.
5. All FP computations round to nearest, ties to even, except conversions from FP numbers to integers, which round toward zero. The CompCert formal semantics makes no provisions for programmers to change rounding modes at run-time.
6. FP literal constants are also rounded to nearest, ties to even.

A detailed presentation of the CompCert C formal semantics is outside the scope of this article. We refer the interested reader to the commented Coq development [23], especially modules `Cop` and `Csem`.

4 Formally-Verified Compilation

As mentioned in changedthe 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 C99 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 encountering 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. The bottom line, however, 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. As outlined in Section 3, 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.

Having thus committed on a semantics for FP computations in CompCert C, it remains to formalize it in the Coq proof assistant so that the correctness proofs of CompCert can guarantee correct compilation of FP arithmetic, just like they already guaranteed correct compilation of integer arithmetic. 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. (Sections 6.2 and 6.3 show 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 run-time 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.[3]

[3] 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.

In contrast, for an axiomatized library such as the early versions of `Float`, we need to map FP operations of the library onto an executable implementation, and trust this implementation to conform to IEEE-754. Early versions of CompCert used the FP operations provided by OCaml as executable implementations. 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.

An alternative that we did not consider at the time is to use MPFR to implement the axiomatized FP operations, as is done in GCC. MPFR provides independence with respect to the host platform, but no formal guarantees of correctness. Instead, we set out to develop 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.

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

Flocq (Floats for Coq) is a formalization for the Coq system [5]. It provides a comprehensive library of theorems on a multi-radix, multi-precision arithmetic. In particular, it encompasses radix-2 and 10 arithmetic, 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. We also generalized some results about rounding to odd in order to formally verify some conversions from integers to FP numbers.

5.1 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 : bool -> nan_pl -> 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, signed infinities, and *NaN*. Zeros and infinities carry their sign, while a *NaN* caries both a sign and a payload (a positive integer that fits with respect to the precision).

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.

5.2 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. If zeros or infinities are produced, the IEEE-754 standard completely specifies their signs, so the implementation is straightforward. For *NaN*, the situation is a bit more complicated, since the standard is under-specified: we know when a *NaN* is produced, but not what its sign nor its payload are. In fact, the implemented behavior varies widely across the architectures supported by CompCert. To circumvent this issue, Flocq's arithmetic operators also take a function as argument. Given the inputs of the operator, this function has to compute a sign and a payload for the resulting *NaN*. CompCert then parametrizes each arithmetic operator by the function corresponding to the target architecture. This takes care of special values, so only finite numbers are left.

There are two different approaches for defining arithmetic operations on finite inputs. The first one involves a `round` function that takes a rounding mode m and a real number as arguments and returns the closest FP number (according to the rounding mode). For instance, the sum of two finite FP numbers can be characterized by $a \oplus b = \texttt{round}(m, \texttt{B2R}(a) + \texttt{B2R}(b))$, 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 operations. The downside is that

it depends on an abstract addition[4] 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 [2] library and in the Flocq core library [5].

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 is no longer any guarantee that they are compliant with the IEEE-754 standard. Therefore, one also has to formally prove such theorems. This approach is used in the FP formalization for ACL2 [34].

As done in HOL Light [16,17], 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. The operations currently formalized in Flocq are opposite, addition, subtraction, multiplication, division, and square root. Note that square root, as well as other standard library functions such as FMA or remainder, are not yet needed in our compiler formalization, as there are no specific inlining optimizations for them.

Theorem 2 is an example of our approach; it shows how the correctness of the FP multiplication `Bmult` is expressed. 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 $(1-2^{-p})\cdot 2^{E_{\max}}$. 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.

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*

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

Moreover, if the result is not NaN, $\texttt{Bsign}(\texttt{Bmult}(m,x,y)) = \texttt{Bsign}(x) \times \texttt{Bsign}(y)$.

While this theorem also holds for exceptional inputs (since `B2R` maps them to zero), it provides a complete specification of the multiplication operator only when both inputs represent finite numbers. When one or both inputs are exceptional, no specific statements are needed, however, since one can simply execute the operator to recover the exact result.

Finally, the statement about the sign of the result might seem redundant with the other statements of the theorem. It is needed in case the multiplication underflows to zero, as $z = 0$ is not sufficient to characterize which floating-point zero is computed

[4] In contrast to integers, the formalization of real numbers in Coq is axiomatic. In other words, arithmetic operations are uninterpreted symbols that cannot be used in any way to perform computations.

5.3 Bit-Level Representation

The last piece of formalization 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` function performs the converse operation.

Both functions have been proved to be inverse of each other for bounded integers. This property 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 is no overlap between the three fields of the binary representation.

5.4 Odd Rounding

Section 7 will detail how conversions between integers and FP numbers are performed. Due to the way they are designed, several of those algorithms are subject to double rounding, *e.g.* converting a 64-bit integer to `binary32` might require to convert it first to `binary64`. This double rounding can be made innocuous by introducing a new rounding mode, called odd rounding, and using it for the first rounding. This was used by Goldberg when converting binary floating-point numbers to decimal representations [14] and formally studied later, notably to emulate the FMA operator [4].

The informal definition of odd rounding is the following: when a real number is not representable, it will be rounded to the adjacent FP number with an odd integer significand. An effective implementation is given in [4]. As the hypotheses characterizing an FP format in Flocq are very loose, we need a few more constraints so that rounding to odd exists for a given format. These constraints are the same as for rounding to nearest, ties to even: if rounding a real value up and down produces two distinct FP numbers, then one should be even and the other odd.

Flocq describes a FP format by a function $\varphi : \mathbb{Z} \to \mathbb{Z}$ that transforms the discrete logarithm of a real number into the canonical exponent for this format [5]. Let us have two different formats: a working format φ_w and an extended format φ. We assume that

$$\forall e \in \mathbb{Z}, \quad \varphi(e) \leq \varphi_w(e) - 2.$$

This informally means that we have an extended format with at least two more digits. Moreover, we assume that both φ_w and φ are valid formats where rounding to nearest, ties to even, can be defined. We also assume that the radix is even (so it works for the usual values 2 and 10). Then, if $\square^{\mathtt{odd}}$ denotes the rounding to odd in format φ, and if $\circ$ denotes a rounding to nearest in format φ_w, with an arbitrary rule for ties, we have

$$\forall x \in \mathbb{R}, \quad \circ\left(\square^{\mathtt{odd}}(x)\right) = \circ(x).$$

The definitions and properties of rounding to odd, and this property amount to one thousand lines of Coq. The main reason is that this was proved with full genericity. The previous result of [4] was only for radix 2 and for floating-point format with gradual underflow. While this was sufficient as far as CompCert is concerned, when porting the proof to Flocq, we have generalized it to any even radix and any reasonable format (as long as rounding to nearest, ties to even can be defined). The fact that the radix is even could be removed in some cases, depending on the parity of the smallest positive FP number. This would have greatly complicated the proof though, and for few possible uses.

What will be used later in Section 7.2 is the deduced theorem:

Theorem 3 *Let us consider two radix-2 FP formats with gradual underflow on p and $p+k$ bits such that the minimal exponent of the extended format is at most the minimal exponent of the other format minus 2. If $k \geq 2$,*

$$\forall x \in \mathbb{R}, \quad \circ_p(x) = \circ_p\left(\square^{\mathtt{odd}}_{p+k}(x)\right).$$

6 A Verified Compiler for Floating-Point Computations

We have integrated the Coq formalization of IEEE-754 arithmetic described in Section 5 into the CompCert compiler, version 1.12, effectively replacing the axiomatization of FP arithmetic used in earlier versions (see Section 4) 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 guarantee 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 assume 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 VFP instruction set, omitting vector instructions; 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, modest optimizations performed on FP arithmetic operations; last, the code generation strategies used to implement C's FP operations in terms of the instructions provided by the target architectures.

6.1 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 5, our extension to the Flocq library provides such an implementation. In particular, the `bits_of_binary_float` function described in Section 5.3 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.[5]

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 [9]. For example, until recently, the `strtod` and `strtof` functions of the GNU C standard library incorrectly rounded the result in some corner cases.[6] 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 is used as default value for the first three parts. (This operation is done in an early stage of parsing in our compiler.) The integral and fractional parts may be written in either decimal or hexadecimal notation; the use of hexadecimal (in both parts) is indicated if the integral part begins with the prefix "`0x`". 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 $\texttt{I.F} \times b^{\texttt{E}}$ 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 `F` into the integral part `I`. Then, it parses both the exponent and the new integral part as arbitrary-

[5] The CompCert C semantics gives programmers no way to change the FP rounding mode during program execution, therefore guaranteeing that all FP arithmetic rounds to nearest even. Programs that need other rounding modes fall outside the perimeter of CompCert's semantic preservation results. They can, however, be supported via a compiler option, `-ffloat-const-prop 0`, which turns FP constant propagation off.

[6] "Incorrect rounding in strtod", `http://sourceware.org/bugzilla/show_bug.cgi?id=3479`

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. Notice that, since Flocq formalizes a multi-precision arithmetic, numbers $\mathtt{I}$ and $b^{-\mathtt{E}}$ do not have to fit into the target format; the division can cope with arbitrarily large numbers.

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.

To estimate the impact on compilation times, we use the following simple benchmark. We compile a C file containing a single statement `double array[100000] = { ... };` When the array is filled with short integer constants, CompCert 2.4 takes about 4 times as long as GCC 4.9.1 to compile this file. When the array also contains some floating-point operations, CompCert is 5 to 6 times slower. Hence, parsing simple constants, performing computations, and outputting binary representation of floating-point numbers are reasonably efficient.

For constants with a large negative decimal exponent, parsing becomes much more expensive. For instance, if the array is filled with `1e-20`, CompCert is about 10 times slower than GCC. The cost grows with the exponent: for `1e-40`, it is about 20 times slower; for `1e-80`, 50 times; and for `1e-160`, 120 times. Most of the time is spent in dividing large integers. Hopefully, few C files contain such constants, so this inefficiency should go unnoticed in the common case.

Note that, on all these tests, GCC 4.9.1 compilation times are 2 to 3 times those of Clang 3.4.

6.2 Verifying Algebraic Simplifications over Floating-Point Operations

For integer computations, compilers routinely apply algebraic identities to generate shorter instruction sequences and use cheaper instructions. Examples include reassociation and distribution of constants (*e.g.* $(n-1) \times 8 + 4$ becomes $n \times 8 - 4$); multiplication by certain constants being transformed into shifts, additions and subtractions (*e.g.* $n \times 7$ becomes $(n \ll 3) - n$); and divisions by constants being replaced by multiplications and shifts [15].

For FP computations, there are much fewer opportunities for compile-time simplifications. The reason is that very few algebraic identities hold over FP arithmetic operations for all possible values of their FP arguments. CompCert implements two modest FP optimizations based on such identities. The first is replacement of double-precision divisions by multiplications if the divisor is an exact power of 2:

$$x \oslash 2^n \to x \otimes 2^{-n} \quad \text{if } |n| < 1023 \tag{1}$$

This optimization is valuable, since FP multiplication is usually much faster than FP division. Such a replacement works for a handful of divisors other than powers of 2, but they are not supported by CompCert.

A second optimization replaces FP multiplications by 2.0 with FP additions:

$$x \otimes 2.0 \to x \oplus x \tag{2}$$
$$2.0 \otimes x \to x \oplus x \tag{3}$$

FP multiplication and addition take about the same time on modern processors, but the optimized form avoids the cost of loading the constant 2.0 in an FP register.

As simple as the optimizations above are, their correctness proof in the case where x is a *NaN* already require additional hypotheses about the payloads produced by FP operations, hypotheses that are, fortunately, satisfied on our three target architectures.

Several other plausible FP optimizations are regrettably unsound for certain values of their arguments:

$$x \oplus 0.0 \not\to x \tag{4}$$
$$x \oplus (-0.0) \not\to x \tag{5}$$
$$x \ominus 0.0 \not\to x \tag{6}$$
$$x \ominus (-0.0) \not\to x \tag{7}$$
$$-(-x) \not\to x \tag{8}$$
$$(-x) \oplus y \not\to y \ominus x \tag{9}$$
$$y \oplus (-x) \not\to y \ominus x \tag{10}$$
$$y \ominus (-x) \not\to y \oplus x \tag{11}$$

Viewed as algebraic identities, (4) and (7) do not hold for $x = -0.0$; (5), (6), and (8) do not hold if x is a signaling *NaN*; and the two sides of (9), (10), and (11) produce *NaN*s of different signs if x (the negated argument) is *NaN* and y (the other argument) is not *NaN*.

Another valuable optimization that is not always correct is the replacement of a FP division by a constant with a multiplication and a fused multiply-add, as described by Brisebarre *et al* [7]:

$$x \oslash c \to \mathrm{fma}(x, c_1, x \otimes c_2) \tag{12}$$

For many values of c, there exists constants c_1 and c_2 that can be computed at compile-time and that validate identity (12) for big enough x. However, when x is small, identity (12) does not always hold, for instance when $x \otimes c_2$ underflows.

The only way to exploit simplifications such as (4)–(12) above while preserving semantics is to apply them conditionally, based on the results of a static analysis that can exclude the problematic cases. As a trivial example of static analysis, in the `then` branch of a conditional statement `if` $(x \geq 1.0)$, we know that x is neither *NaN* nor -0.0 nor subnormal, therefore optimizations (4)–(12) are sound. We are currently developing and verifying a static analysis for FP intervals that could provide similar guarantees in other, less obvious cases.

6.3 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 and x86 architectures. The PowerPC provides an `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. Note that, even if two *NaN*s have the same sign/payload and thus are equal from the mathematical point of view of Coq, the comparison operators defined by the compiler still know that *NaN*s shall be unordered [25].

On the x86 architecture, the `comisd x y` SSE2 instruction sets the ZF, CF and PF condition flags in such a way that only the following relations between `x` and `y` (and their negations) can be tested by a single conditional branch instruction:

- `x == y` or `x`, `y` are unordered (instructions `je`, `jne`)
- `x >= y` (`jae`, `jb`)
- `x > y` (`ja`, `jbe`)
- `x`, `y` are unordered (`jp`, `jnp`)

Therefore, a branch on equality `x == y` or disequality `x != y` must be compiled as a comparison `comisd x y` followed by two conditional branches:

```
if (x == y) goto L    →        comisd x, y
                               jp L'
                               je L
                          L' :
if (x != y) goto L    →        comisd x, y
                               jp L
                               jne L
```

The first `jp` conditional branch handles the case where `x` and `y` are unordered. In this case, they compare as not equal, hence the code for `x != y` branches to the label L, and the code for `x == y` does not branch to L, instead continuing on the label L' for the following code. Once the unordered case is taken care of, the IA32 condition `e`, meaning "equal or unordered", coincides with "equal", hence the `je` or `jne` conditional branch to L implements the correct behavior.

Likewise, branches on `x < y` or `x <= y` can be compiled as `comisd x y` followed by two conditional branches (`jp-jb` and `jp-jbe`, respectively). However, one conditional branch suffices if the compiler reverses the operands of the `comisd` instruction and tests `y > x` or `y >= x` instead. Again, the soundness of these code generation tricks follows from semantic properties of FP comparisons that we easily verified in Coq, namely $x < y \equiv y > x$ and $x \leq y \equiv y \geq x$, and the fact that the four outcomes of a FP comparison (less, equal, greater, unordered) are mutually exclusive.

The most convoluted code generation schemes for FP operations are found in the conversions between integers and FP numbers. In the C language, such

from \ to	s32	u32	s64	u64
f32	AS	A	–	–
f64	APS	A	–	–
f80	X	–	X	–

from \ to	f32	f64	f80
s32	AS	AS	X
u32	A	A	–
s64	–	–	X
u64	–	–	–

Table 1 Conversions between integers and FP numbers that are natively supported on three processor architectures. A stands for ARM with VFPv2 or higher; P for PowerPC 32 bits; S for the SSE2 instructions of x86 in 32-bit mode; and X for the x87 extended-precision instructions of x86.

conversions occur either explicitly ("type casts") or implicitly (during assignments and function parameter passing). There are many such conversions to implement. In the case of CompCert 2.0, there are four integer types and two FP types to consider:

`s32`	32-bit signed integers	`f32`	`binary32` FP numbers
`u32`	32-bit unsigned integers	`f64`	`binary64` FP numbers
`s64`	64-bit signed integers		
`u64`	64-bit unsigned integers		

for a total of 8 integer-to-FP conversions and 8 FP-to-integer conversions.

Table 1 summarizes which of these 16 conversions are directly supported by hardware instructions for each of CompCert's three target architectures. (For completeness, we also list the conversion instructions that operate over the `f80` extended-precision format of the Intel x87 floating-point coprocessor.) As shown by this table, none of our target architectures provides instructions for all 16 conversions. The PowerPC 32-bit architecture is especially unhelpful in this respect, providing only one FP-to-integer conversion (`s32_f64`) and zero integer-to-FP conversions.

All conversions not directly provided by a processor instruction must, therefore, be synthesized by the compiler as sequences of other instructions. These instruction sequences are often nonobvious, and their correctness proofs are sometimes delicate — so much so that we devote the next section (Section 7) entirely to this topic.

7 Implementing Conversions Between FP and Integer Numbers

In this section, we list a number of ways in which conversions between FP and integer numbers that are not natively supported in hardware can be synthesized in software from other processor instructions. These implementation schemes include those used by CompCert, plus several schemes observed in the code generated by GCC version 4. We have formally proved in Coq the correctness of all claimed equalities. The formal proofs can be found in the CompCert 2.4 distribution.

We write t_t' for the conversion from type t' to type t. The types of interest are, on one side, the FP formats `f32` and `f64`, and, on the other side, the integer types `s32`, `u32`, `s64`, and `u64` (signed or unsigned, 32 or 64 bits). For example, `f64_u32` is the conversion from 32-bit unsigned integers to `binary64` FP numbers.

7.1 From 32-bit Integers to FP Numbers

Among CompCert's target architectures, only ARM VFP provides instructions for all four conversions `f64_s32`, `f64_u32`, `f32_s32`, and `f32_u32`. The x86-32 architecture provides one SSE2 instruction for the `f64_s32` conversion. Its unsigned counterpart, `f64_u32`, can be synthesized from `f64_s32` by a case analysis that reduces the integer argument to the $[0, 2^{31})$ range:

$$\begin{aligned} \texttt{f64\_u32}(n) = \ & \texttt{if } n < 2^{31} \\ & \texttt{then f64\_s32}(n) \\ & \texttt{else f64\_s32}(n - 2^{31}) \oplus 2^{31} \end{aligned} \tag{13}$$

Both the `f64_s32` conversion and the FP addition in the `else` branch are exact, the latter because it is performed at `binary64` format.

Conversions from 32-bit integers to `binary32` format are trivially implemented by first converting to `binary64`, then rounding to `binary32`:

$$\texttt{f32\_s32}(n) = \texttt{f32\_f64}(\texttt{f64\_s32}(n)) \tag{14}$$

$$\texttt{f32\_u32}(n) = \texttt{f32\_f64}(\texttt{f64\_u32}(n)) \tag{15}$$

The inner conversion is exact and the outer `f32_f64` conversion rounds the result according to the current rounding mode, as prescribed by the IEEE-754 standard and the ISO C standards, appendix F.

The x87 extended precision FP instructions provide the following alternative implementations, where the inner conversion is also exact:

$$\texttt{f64\_s32}(n) = \texttt{f64\_f80}(\texttt{f80\_s32}(n)) \tag{16}$$

$$\texttt{f64\_u32}(n) = \texttt{f64\_f80}(\texttt{f80\_s64}(\texttt{zero\_ext}(n))) \tag{17}$$

$$\texttt{f32\_s32}(n) = \texttt{f32\_f80}(\texttt{f80\_s32}(n)) \tag{18}$$

$$\texttt{f32\_u32}(n) = \texttt{f32\_f80}(\texttt{f80\_s64}(\texttt{zero\_ext}(n))) \tag{19}$$

However, these alternative instruction sequences can involve more data transfers through memory than the SSE2 implementations described above.

The PowerPC 32-bit architecture offers a bigger challenge, since it fails to provide any integer-to-FP conversion instruction. The PowerPC compiler writer's guide [18] describes the following software implementation, based on bit-level manipulations over the `binary64` format combined with a regular FP subtraction:

$$\texttt{f64\_u32}(n) = \texttt{f64make}(\texttt{0x43300000}, n) \ominus 2^{52} \tag{20}$$

$$\texttt{f64\_s32}(n) = \texttt{f64make}(\texttt{0x43300000}, n + 2^{31}) \ominus (2^{52} + 2^{31}) \tag{21}$$

We write $\texttt{f64make}(h, l)$, where h and l are 32-bit integers, for the `binary64` FP number whose in-memory representation is the 64-bit word obtained by concatenating h with l. This `f64make` operation can easily be implemented by storing h and l in two consecutive 32-bit memory words, then loading a `binary64` FP number from the address of the first word.

The reason why this clever implementation produces correct results is that the `binary64` number $A = \texttt{f64make}(\texttt{0x43300000}, n)$ is equal to $2^{52} + n$ for any integer $n \in [0, 2^{32})$. Hence,

$$A \ominus 2^{52} = \circ\left(\left(2^{52} + n\right) - 2^{52}\right) = \circ(n) = n.$$

Likewise, the `binary64` number $B = \texttt{f64make}(\texttt{0x43300000}, n + 2^{31})$ is equal to $2^{52} + n + 2^{31}$ for any integer $n \in [-2^{31}, 2^{31})$. Hence,

$$B \ominus \left(2^{52} + 2^{31}\right) = \circ\left(\left(2^{52} + n + 2^{31}\right) - \left(2^{52} + 2^{31}\right)\right) = \circ(n) = n.$$

7.2 From 64-bit Integers to FP Numbers

None of CompCert's target architectures provide instructions for the conversions `f64_s64`, `f64_u64`, `f32_s64`, and `f32_u64`. The closest equivalent is the `f80_s64` conversion instruction found in the x87 subset of the x86 32-bit architecture, which gives the following implementations:

$$\texttt{f64\_s64}(n) = \texttt{f64\_f80}(\texttt{f80\_s64}(n)) \tag{22}$$

$$\begin{aligned}\texttt{f64\_u64}(n) = \texttt{f64\_f80}(&\texttt{if } n < 2^{63} \\ &\texttt{then f64\_s64}(n) \\ &\texttt{else f64\_s64}(n - 2^{63}) \oplus 2^{63})\end{aligned} \tag{23}$$

and likewise for `f32_s64` and `f32_u64`, replacing the final `f64_f80` rounding by `f32_f80`. Since the 80-bit extended-precision FP format of the x87 has a 64-bit significand, it can exactly represent any integer in the range $(-2^{64}, 2^{64})$. Hence, all FP computations in the formulas above are exact, except the final `f64_f80` or `f32_f80` conversions, which perform the correct rounding.

If the target architecture provides only conversions from 32-bit integers, it is always possible to convert a 64-bit integer by splitting it in two 32-bit halves, converting them, and combining the results. Writing $n = 2^{32}h + l$, where h and l are 32-bit integers and l is unsigned, we have

$$\texttt{f64\_s64}(n) = \texttt{f64\_s32}(h) \otimes 2^{32} \oplus \texttt{f64\_u32}(l) \tag{24}$$

$$\texttt{f64\_u64}(n) = \texttt{f64\_u32}(h) \otimes 2^{32} \oplus \texttt{f64\_u32}(l) \tag{25}$$

All operations are exact except the final FP addition, which performs the correct rounding. For the same reason, a fused multiply-add instruction can be used if available, without changing the result.

On PowerPC 32 bits, we can combine implementations (20) and (25), obtaining

$$\begin{aligned}\texttt{f64\_u64}(n) = (&\texttt{f64make}(\texttt{0x43300000}, h) \ominus 2^{52}) \otimes 2^{32} \\ &\oplus (\texttt{f64make}(\texttt{0x43300000}, l) \ominus 2^{52})\end{aligned} \tag{26}$$

A first improvement is to fold the multiplication by 2^{32} with the first `f64make` FP construction:

$$\begin{aligned}\texttt{f64\_u64}(n) = (&\texttt{f64make}(\texttt{0x45300000}, h) \ominus 2^{84}) \\ &\oplus (\texttt{f64make}(\texttt{0x43300000}, l) \ominus 2^{52})\end{aligned} \tag{27}$$

Indeed, just like $\texttt{f64make}(\texttt{0x43300000}, n) = 2^{52} + n$ for all 32-bit unsigned integers n, it is also the case that $\texttt{f64make}(\texttt{0x45300000}, n) = 2^{84} + n \times 2^{32}$.

One further improvement is possible:

$$\begin{aligned}\texttt{f64\_u64}(n) = (&\texttt{f64make}(\texttt{0x45300000}, h) \ominus (2^{84} + 2^{52})) \\ &\oplus \texttt{f64make}(\texttt{0x43300000}, l)\end{aligned} \tag{28}$$

Indeed, `f64make`$(\texttt{0x45300000}, h)$ ranges over $[2^{84}, 2^{84}+2^{64})$, so it is within a factor of two of $2^{84}+2^{52}$, hence the FP subtraction is exact. This leaves only the outer FP addition that correctly rounds to the final result. A similar analysis for the signed integer case gives:

$$\begin{aligned}\texttt{f64\_s64}(n) = (&\texttt{f64make}(\texttt{0x45300000}, h + 2^{31}) \ominus (2^{84} + 2^{63} + 2^{52})) \\ &\oplus \texttt{f64make}(\texttt{0x43300000}, l)\end{aligned} \tag{29}$$

Many of the implementation schemes for 32-bit integer to FP conversions listed in Section 7.1 do not extend straightforwardly to the 64-bit case, because double rounding can occur. For instance, assuming the `f64_s64` conversion is available, it is not correct to define its unsigned counterpart `f64_u64` in the style of implementation (13):

$$\begin{aligned}\texttt{f64\_u64}(n) \neq\ &\texttt{if}\ \ n < 2^{63} \\ &\texttt{then}\ \texttt{f64\_s64}(n) \\ &\texttt{else}\ \texttt{f64\_s64}(n - 2^{63}) \oplus 2^{63}\end{aligned}$$

Indeed, for some values of $n > 2^{63} + 2^{52}$, both the conversion $A = \texttt{f64\_s64}(n - 2^{63})$ and the FP addition $A \oplus 2^{63}$ are inexact, and the two consecutive roundings produce a result different from the correct single rounding of n to `binary64` format.

Looking at the assembly code generated by GCC 4 for the PowerPC 64-bit architecture, we observe an elegant workaround for this issue:

$$\begin{aligned}\texttt{f64\_u64}(n) = \ &\texttt{if}\ \ n < 2^{63} \\ &\texttt{then}\ \texttt{f64\_s64}(n) \\ &\texttt{else}\ 2 \otimes \texttt{f64\_s64}((n \texttt{ >> } 1) \texttt{ | } (n \texttt{ \& } 1))\end{aligned} \tag{30}$$

This is an instance of the *round-to-odd* technique presented in Section 5.4. The computation $n' = (n \texttt{ >> } 1) \texttt{ | } (n \texttt{ \& } 1)$ has the effect of rounding $n/2$ to odd. Indeed, looking at the two low-order bits of n, we have

n	n'	
$4k$	$2k$	(even, but an exact quotient)
$4k+1$	$2k+1$	(quotient rounded up)
$4k+2$	$2k+1$	(exact quotient)
$4k+3$	$2k+1$	(quotient rounded down)

Therefore, the `else` case of implementation (30) is actually computing

$$\begin{aligned}2 \otimes \texttt{f64\_s64}(n') &= 2 \otimes \circ_{53}(\square_{63}^{\texttt{odd}}(n/2)) \\ &= 2 \otimes \circ_{53}(n/2) \\ &= \circ_{53}(n)\end{aligned}$$

The second equality follows from Theorem 3 with $p = 53$ and $p + k = 63$. The third equality follows from $n \geq 2^{63}$.

Another situation where double rounding rears its ugly head is converting 64-bit integers to `binary32` FP numbers. Again, it is not correct to proceed as in the 32-bit case, simply converting to `binary64` then rounding to `binary32`:

$$\begin{aligned}\texttt{f32\_s64}(n) &\neq \texttt{f32\_f64}(\texttt{f64\_s64}(n)) \\ \texttt{f32\_u64}(n) &\neq \texttt{f32\_f64}(\texttt{f64\_u64}(n))\end{aligned}$$

For large enough values of n, the conversion to `f64` is inexact, causing a double rounding error in conjunction with the subsequent `f32_f64` rounding.

Looking once more at the code generated by GCC 4 for `f32_u64` on PowerPC 64 bits, we observe another clever use of the round-to-odd technique:

$$\begin{array}{l}\texttt{f32\_u64}(n) = \texttt{f32\_f64}(\texttt{f64\_u64}(\texttt{if } n < 2^{53} \texttt{ then } n \texttt{ else } n')) \\ \text{where } n' = (n \mid ((n \,\&\, \texttt{0x7FF}) + \texttt{0x7FF})) \,\&\, \texttt{\~{}0x7FF}\end{array} \tag{31}$$

In the $n < 2^{53}$ case, the result of $\texttt{f64\_u64}(n)$ is exact and therefore a single rounding to `f32` occurs. In the other case, unraveling the computation of n', we see that the low 11 bits of n' are 0; the high 52 bits of n' are identical to those of n; and bit number 11 of n' is equal to bit number 11 of n if all low 11 bits of n are 0, and is forced to 1 otherwise. Therefore, n' is n rounded to $q = \lceil \log_2 n \rceil - 11$ significant bits using round-to-odd mode. Since n' has only $q \leq 53$ significant bits, its conversion $\texttt{f64\_u64}(n')$ is exact. The correctness of implementation (31), then, follows from Theorem 3, with $p = 24$ and $p + k = q \in [42, 53]$.

The same trick also applies to the signed conversion `f32_s64`:

$$\begin{array}{l}\texttt{f32\_s64}(n) = \texttt{f32\_f64}(\texttt{f64\_s64}(\texttt{if } |n| < 2^{53} \texttt{ then } n \texttt{ else } n')) \\ \text{where } n' \text{ is computed from } n \text{ as in (31)}\end{array} \tag{32}$$

Owing to two's-complement representation of integers, the logical and arithmetic operations defining n' from n perform round-to-odd even if n is negative. Note that the `then` path is also correct if $|n| = 2^{53}$. GCC 4 takes advantage of this fact by testing whether $-2^{53} \leq n < 2^{53}$, which can be done with only one conditional jump.

7.3 From FP Numbers to Integers

Conversions from FP numbers to integers are more straightforward than the conversions described in Sections 7.1 and 7.2. The general specification for FP-to-integer conversions, as given in the ISO C standards, is that they must round the given FP number f towards zero to obtain an integer. If the resulting integer falls outside the range of representable values for the target integer type (*e.g.* $[-2^{31}, 2^{31})$ for target type `s32`), or if the FP argument is *NaN*, the conversion has undefined behavior: it can produce an arbitrary integer result, but it can also abort the program.

All our target architectures of interest provide an instruction converting `binary64` FP numbers to signed 32-bit integers, rounding towards zero (the `s32_f64` conversion). The behaviors of these instructions differ in the overflow case, but this does not matter because such overflow behavior is undefined by ISO C.

Conversion to unsigned 32-bit integers can be obtained from the signed conversion `s32_f64` plus a case analysis:

$$\begin{array}{l}\texttt{u32\_f64}(f) = \texttt{if } f < 2^{31} \\ \qquad\qquad \texttt{then s32\_f64}(f) \\ \qquad\qquad \texttt{else s32\_f64}(f \ominus 2^{31}) + 2^{31}\end{array} \tag{33}$$

The conversion `u32_f64`(f) is defined only if $f \in [0, 2^{32})$. In this case, the FP subtraction $f \ominus 2^{31}$ in the `else` branch is exact, and in either branch `s32_f64` is applied to an argument in the $[0, 2^{31})$ range, where it is defined.

The same construction applies in the case of 64-bit integers:

$$\begin{aligned} \texttt{u64\_f64}(f) = \texttt{if } & f < 2^{63} \\ & \texttt{then s64\_f64}(f) \\ & \texttt{else s64\_f64}(f \ominus 2^{63}) + 2^{63} \end{aligned} \tag{34}$$

CompCert uses this implementation for the x86 platform, where `s64_f64`(f) is implemented using the x87 80-bit FP operations as `s64_f80`(`f80_f64`(f)). The only caveat is that the `s64_f80` instruction of x87 rounds using the current rounding mode (to nearest even, by default); therefore, the rounding mode must be temporarily changed to round-towards-zero, which is costly.

ARM and PowerPC 32 bits provide no conversion instructions that produce 64-bit integers. We considered various implementations for `s64_f64` and `u64_f64`, including one based on the ExtractScalar algorithm [33], then finally settled on rather pedestrian implementations that extract the integer significand and shift it appropriately based on the exponent. We show pseudocode for `u64_f64`.

```
u64 u64_f64(f64 f)
{
  int s = bit<63>(f);             // extract sign and
  int e = bits<62:52>(f) - 1023; // unbiased exponent
  if (s != 0 || e >= 64)          // f<0 or f>=2^64 ?
    return OVERFLOW;              // arbitrary result
  if (e < 0)                      // f<1 ?
    return 0;                     // it converts to 0
  u64 m = bits<51:0>(f) | 1<<52; // extract mantissa
  if (e >= 52)                    // and shift it
    return m << (e - 52);
  else
    return m >> (52 - e);
}
```

8 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 CompCert since version 1.12, available at `http://compcert.inria.fr/`. This work required to add about 5,000 lines of new Coq proofs to both CompCert and Flocq.

This approach gives a correct and predictable compiler that conforms to the IEEE-754 standard. This means that, among the several possibilities allowed by the ISO C standard, we have chosen a single way to compile and we have formally proved its correctness. This compilation choice can be discussed: for example all

intermediate results are computed with a precision matching their static C type, therefore with (usually) less accuracy than with extended registers. The first reason is that this is sorely needed to be able to prove algorithms or programs. The second reason is that we favored reproducibility over possible higher accuracy. 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 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.

The integration of Flocq made it possible to enrich CompCert with a few optimizations specific to FP arithmetic, as shown in Section 6.2, and to prove them correct. For the semantic preservation theorem to remain valid, however, only algebraic identities that hold for all representable FP numbers (including the payload of *NaN*) can be used, which severely restricts the amount of optimization that can be performed. Exploiting the results of a static analysis over FP variables could enable a few additional optimizations. However, aggressive loop optimizations such as vectorization, which often entail reassociating FP operations, cannot be supported in a verified compiler such as CompCert, since they 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 [19].

Acknowledgments

This work was supported by the Verasco project (ANR-11-INSE-003) of Agence Nationale de la Recherche. We thank the anonymous reviewers and Paul Zimmermann for their insightful comments on this paper.

References

1. Ayad, A., Marché, C.: Multi-prover verification of floating-point programs. In: J. Giesl, R. Hähnle (eds.) 5th International Joint Conference on Automated Reasoning (IJCAR), *Lecture Notes in Computer Science*, vol. 6173. Springer, Edinburgh, Scotland (2010)
2. Boldo, S.: Preuves formelles en arithmétiques à virgule flottante. Ph.D. thesis, École Normale Supérieure de Lyon (2004)

3. Boldo, S., Filliâtre, J.C.: Formal verification of floating-point programs. In: P. Kornerup, J.M. Muller (eds.) 18th IEEE International Symposium on Computer Arithmetic (ARITH), pp. 187–194. IEEE, Montpellier, France (2007)
4. Boldo, S., Melquiond, G.: Emulation of FMA and correctly-rounded sums: Proved algorithms using rounding to odd. IEEE Transactions on Computers **57**(4), 462–471 (2008)
5. Boldo, S., Melquiond, G.: Flocq: A unified library for proving floating-point algorithms in Coq. In: E. Antelo, D. Hough, P. Ienne (eds.) 20th IEEE Symposium on Computer Arithmetic, pp. 243–252. IEEE, Tübingen, Germany (2011)
6. Boldo, S., Nguyen, T.M.T.: Proofs of numerical programs when the compiler optimizes. Innovations in Systems and Software Engineering **7**, 151–160 (2011)
7. Brisebarre, N., Muller, J.M., Raina, S.K.: Accelerating correctly rounded floating-point division when the divisor is known in advance. IEEE Transactions on Computers **53**(8), 1069–1072 (2004)
8. Carreño, V.A., Miner, P.S.: Specification of the IEEE-854 floating-point standard in HOL and PVS. In: 8th International Workshop on Higher-Order Logic Theorem Proving and Its Applications (HOL95). Aspen Grove, UT (1995)
9. Clinger, W.D.: How to read floating-point numbers accurately. In: Programming Language Design and Implementation (PLDI'90), pp. 92–101. ACM (1990)
10. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTRÉE analyzer. In: 14th European Symposium on Programming (ESOP), *Lecture Notes in Computer Science*, vol. 3444, pp. 21–30. Springer (2005)
11. Dekker, T.J.: A floating point technique for extending the available precision. Numerische Mathematik **18**(3), 224–242 (1971)
12. Delmas, D., Goubault, E., Putot, S., Souyris, J., Tekkal, K., Védrine, F.: Towards an industrial use of FLUCTUAT on safety-critical avionics software. In: Formal Methods for Industrial Critical Systems (FMICS), *Lecture Notes in Computer Science*, vol. 5825, pp. 53–69. Springer (2009)
13. Figueroa, S.A.: When is double rounding innocuous? SIGNUM Newsletter **30**(3), 21–26 (1995)
14. Goldberg, D.: What every computer scientist should know about floating point arithmetic. ACM Computing Surveys **23**(1), 5–47 (1991)
15. Granlund, T., Montgomery, P.L.: Division by invariant integers using multiplication. In: Programming Language Design and Implementation (PLDI'94), pp. 61–72. ACM (1994)
16. Harrison, J.: A machine-checked theory of floating point arithmetic. In: Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, L. Théry (eds.) 12th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'99), *Lecture Notes in Computer Science*, vol. 1690, pp. 113–130. Springer, Nice, France (1999)
17. Harrison, J.: Formal verification of floating point trigonometric functions. In: 3rd International Conference on Formal Methods in Computer-Aided Design (FMCAD), *Lecture Notes in Computer Science*, vol. 1954, pp. 217–233. Springer, Austin, Texas (2000)
18. IBM: The PowerPC Compiler Writer's Guide. Warthman Associates (1996)
19. Ioualalen, A., Martel, M.: A new abstract domain for the representation of mathematically equivalent expressions. In: 19th International Symposium on Static Analysis, *Lecture Notes in Computer Science*, vol. 7460, pp. 75–93. Springer (2012)
20. ISO: International standard ISO/IEC 9899:2011, Programming languages – C (2011)
21. Leavens, G.T.: Not a number of floating point problems. Journal of Object Technology **5**(2), 75–83 (2006)
22. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM **52**(7), 107–115 (2009)
23. Leroy, X.: The CompCert verified compiler, software and commented proof. Available at `http://compcert.inria.fr/` (2014)
24. Li, G., Owens, S., Slind, K.: Structure of a proof-producing compiler for a subset of higher order logic. In: R.D. Nicola (ed.) 16th European Symposium on Programming (ESOP), *Lecture Notes in Computer Science*, vol. 4421, pp. 205–219. Springer, Braga, Portugal (2007)
25. Microprocessor Standards Subcommittee: IEEE Standard for Floating-Point Arithmetic. IEEE Std. 754-2008 pp. 1–58 (2008)
26. Milner, R., Weyhrauch, R.: Proving compiler correctness in a mechanized logic. In: B. Meltzer, D. Michie (eds.) 7th Annual Machine Intelligence Workshop, *Machine Intelligence*, vol. 7, pp. 51–72. Edinburgh University Press (1972)

27. Monniaux, D.: The pitfalls of verifying floating-point computations. ACM Transactions on Programming Languages and Systems **30**(3), 1–41 (2008)
28. Moore, J.S.: A mechanically verified language implementation. Journal of Automated Reasoning **5**(4), 461–492 (1989)
29. Muller, J.M., Brisebarre, N., de Dinechin, F., Jeannerod, C.P., Lefèvre, V., Melquiond, G., Revol, N., Stehlé, D., Torres, S.: Handbook of Floating-Point Arithmetic. Birkhäuser (2010)
30. Myreen, M.O.: Formal verification of machine-code programs. Ph.D. thesis, University of Cambridge (2008)
31. Nguyen, T.M.T., Marché, C.: Hardware-dependent proofs of numerical programs. In: J.P. Jouannaud, Z. Shao (eds.) International Conference on Certified Programs and Proofs (CPP), *Lecture Notes in Computer Science*, vol. 7086, pp. 314–329. Springer (2011)
32. Nickolls, J., Dally, W.: The GPU computing era. IEEE Micro **30**(2), 56–69 (2010)
33. Rump, S., Ogita, T., Oishi, S.: Accurate floating-point summation Part I: Faithful rounding. SIAM Journal of Scientific Computing **31**(1), 189–224 (2008)
34. Russinoff, D.M.: 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 **1**, 148–200 (1998)
35. Sterbenz, P.H.: Floating point computation. Prentice Hall (1974)
36. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in C compilers. In: 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 283–294. ACM Press (2011)