Enabling Floating-Point Arithmetic in the Coq Proof Assistant

Érik Martin-Dorel[1], Guillaume Melquiond[2], Pierre Roux[3]

[1]IRIT, Université de Toulouse, CNRS, Toulouse INP, UT3, Toulouse, France.
[2]Université Paris Saclay, CNRS, ENS Paris Saclay, Inria, LMF, Gif-sur-Yvette, France.
[3]ONERA/DTIS, Université de Toulouse, Toulouse, France.

Contributing authors: erik.martin-dorel@irit.fr; guillaume.melquiond@inria.fr; pierre.roux@onera.fr;

Abstract

Floating-point arithmetic is a well-known and extremely efficient way of performing approximate computations over the real numbers. Although it requires some careful considerations, floating-point numbers are nowadays routinely used to prove mathematical theorems. Numerical computations have been applied in the context of formal proofs too, as illustrated by the CoqInterval library. But these computations do not benefit from the powerful floating-point units available in modern processors, since they are emulated inside the logic of the formal system. This paper experiments with the use of hardware floating-point numbers for numerically intensive proofs verified by the Coq proof assistant. This gives rise to various questions regarding the formalization, the implementation, the usability, and the level of trust. This approach has been applied to the CoqInterval and ValidSDP libraries, which demonstrates a speedup of at least one order of magnitude.

Keywords: formal proof, floating-point arithmetic, proof by computation

1 Introduction

Efficient and reproducible floating-point computations are widely available nowadays, from embedded processors to supercomputers, thanks to the internationally recognized

IEEE-754 standard [1]. The primary use of floating-point arithmetic is to perform approximate computations over real numbers. Due to the limited precision and range of floating-point numbers, various numerical issues arise: rounding errors, overflows, underflows, loss of algebraic properties, and so on.

These issues did not stop people from using floating-point arithmetic for serious applications. In fact, given sufficient care in the implementation, intensive floating-point computations can even be used in proofs of mathematical theorems, *e.g.*, the existence of a strange attractor for Lorenz' equations [2]. To do so, the algorithms not only compute a floating-point approximation of the ideal real result but also a bound on the approximation error. This bound might be computed dynamically, as is the case with interval arithmetic [3]. This approach is easy to use and correct by construction but it is more expensive than traditional floating-point arithmetic. Another approach consists in mathematically bounding the approximation error by performing a comprehensive floating-point analysis. Correctness becomes much more tedious to guarantee (though tools for static analysis might help in some specific cases), but this offers a wide range of efficient yet rigorous algorithms [4].

Both approaches have been used to formally prove theorems with the Coq proof assistant. On the one hand, the use of interval arithmetic can be illustrated by the CoqInterval library [5], which automatically and formally proves properties on real expressions by computing their floating-point enclosures. To do so, it specifies, implements, and verifies an adhoc floating-point arithmetic in Coq: arbitrary radix, arbitrary precision, unbounded exponent range, neither infinities nor signed zeros. On the other hand, the use of precomputed error bounds can be illustrated by the formal verification of the rigorous variant of Cholesky's decomposition [6]. This time, the underlying arithmetic complies with the IEEE-754 standard, so one can use the reference implementation provided by the Flocq library to perform the floating-point computations [7]. In either case, computations on floating-point numbers are emulated inside the logic of Coq using integer arithmetic, and thus they do not benefit from the highly efficient floating-point unit of the processor running the proof assistant.

To improve on this unfortunate state of affairs, support for hardware floating-point numbers was added to Coq [8], in a way reminiscent of the support for machine integers [9, 10]. In both cases, the process was as follows. One first declares an abstract type as well as some operations over it: addition, multiplication, and so on. Then, one provides some dedicated reduction rules, which delegate the operations to the arithmetic unit of the processor. This makes it possible to formally prove an equality such as $1.0 + 2.0 = 2.0 + 1.0$ by reducing it to $3.0 = 3.0$. But short of enumerating all the finitely-many pairs of floating-point numbers (which is practically impossible), this is not sufficient to formally prove $\forall x, y,\ x + y = y + x$. So, one also has to relate these abstract operations to some concrete definitions that are much slower but exhibit suitable mathematical properties.

Section 2 focuses on the latter part, that is, how these concrete definitions are formalized in Coq. A peculiarity of this specification of floating-point arithmetic is that it is complete but hardly useful in isolation, as it explains how the numbers are computed but not what their properties are. Our first contribution was to extend this specification using the Flocq library to obtain a meaningful formalization that bridges the gap between the hardware floating-point numbers provided by Coq and the real numbers.

Section 3 then focuses on the actual implementation. It quickly reminds how the various conversion and reduction engines of Coq were modified to support hardware floating-point computations [8]. It then details various improvements we have contributed since then, to make this support more useful and usable. In particular, it explains some design choices related to rounding modes (which are critical for interval arithmetic) and to parsing and printing. Finally, it discusses the issue of trust, as both the specification and implementation amount to adding a large numbers of axioms. In particular, this article analyzes all the soundness bugs that were found (and fixed) during the four years that followed the original implementation.

Even once a consistent system has been recovered, relying on the hardware floating-point unit in a proof assistant would be pointless if it required too much of a proof effort or if the performance gain was too small. So, we have converted two preexisting, representative, Coq developments, so as to evaluate the costs and the benefits. Section 4 describes what kind of work this conversion entailed. It also benchmarks how proofs relying on hardware floating-point arithmetic compare to those based on emulated computations, performance-wise.

2 Specification: Coq and Flocq

From the point of view of the logic of Coq, the type `float` of floating-point numbers is completely abstract. Similarly, basic operations on these numbers are declared as axioms. Except for some hardcoded reduction rules for these operations (which are delegated to the floating-point unit of the processor), no property is known. Thus, a specification describing this arithmetic is needed. Its role is twofold.

First, it should precisely characterize what the floating-point operations compute. In other words, one should be able to prove properties about what is being computed without performing the computation, *e.g.*, commutativity of addition. The axioms describing this part of the specification are shipped with Coq's standard library. It provides an inductive data type `spec_float` representing floating-point numbers, as well as conversion functions from and to the abstract type `float`. It also provides some naive implementation of the arithmetic operations over `spec_float`, and then states as axioms that the conversion function is a morphism from `float` to `spec_float`. Section 2.1 gives more details about this specification.

This first specification is complete but useless in practice, as it is purely operational. It is no different from a software floating-point emulator such as SoftFloat.[1] One should not have to look at the implementation to make use of floating-point numbers inside proofs. So, we need higher-level properties about floating-point operations. In particular, the IEEE-754 standard states that an arithmetic operation shall be performed as if it first computed the result with infinite precision and then rounded it to the target floating-point format [1]. By "infinite precision", the standard simply means that, except for the exceptional values, floating-point numbers are just real numbers, and operations behave the same, rounding notwithstanding. With such a specification,

[1] http://www.jhauser.us/arithmetic/SoftFloat.html

it becomes possible to perform floating-point computations to prove properties about real numbers. This higher-level specification is provided by the Flocq library [7]. If not for the large and intricate proofs that relate both specifications, the Flocq specification could also have been shipped with Coq's standard library. Section 2.2 gives more details about it.

2.1 On Coq's side

Support for hardware floating-point arithmetic in Coq was inspired by two libraries: the Flocq library [7], which gathers an IEEE-754-compliant executable formalization, formerly in the module `IEEE754.Binary`,[2] and the formalization of primitive 63-bit integers [10], now part of the standard library as module `UInt63`.[3] The former provides a precise specification in the form of a reference implementation of floating-point operators, while the latter guided the implementation methodology.

Flocq defines an inductive type `full_float` that represents signed zeros, signed infinities, unbounded floating-point numbers with any integer mantissa `m` and exponent `e`, and NaNs (Not-a-Number) with a payload `m`. The payload distinguishes the different binary encodings of NaNs.[4]

```
Variant full_float :=
  | F754_zero (s : bool)
  | F754_infinity (s : bool)
  | F754_nan (s : bool) (m : positive)
  | F754_finite (s : bool) (m : positive) (e : Z).
```

Our aim was to extract a subset of this Flocq theory sufficient to completely specify floating-point numbers and operators, so that Coq does not depend on Flocq at compilation time. We started by porting this definition, just removing the NaN payloads because they are not fully specified by the IEEE-754 standard, including the distinction between signaling and quiet NaNs, which can lead to implementation discrepancies between hardware vendors. Thus, ignoring these payloads in Coq is paramount to guarantee the portability of computations and proofs performed with the same Coq script on different processors.

```
Variant spec_float : Set :=
  | S754_zero (s : bool)
  | S754_infinity (s : bool)
  | S754_nan (* no payload *)
  | S754_finite : (s : bool) (m : positive) (e : Z).
```

Values of type `spec_float` are not normalized, since no bound is enforced on mantissas and exponents. But in practice, all the operators will make sure that the exponent `e` is bounded, and that the mantissa `m` contains 53 bits for normalized numbers, and at most 52 for subnormal numbers, thus ensuring that the number belongs to the `binary64` format. This property is denoted by the `valid_binary` predicate.

[2] https://gitlab.inria.fr/flocq/flocq/-/blob/flocq-3.2.1/src/IEEE754/Binary.v
[3] https://coq.github.io/doc/V8.17.0/stdlib/Coq.Numbers.Cyclic.Int63.Uint63.html
[4] For instance, the `binary64` format makes room for $2^{53} - 2$ different NaN values.

Note that this definition actually matches Flocq's original formalization of the IEEE-754 standard [11]. But Flocq was later extended to make it suitable for the semantics used in the CompCert C compiler [12]. Indeed, even if nothing can be specified about NaN payloads, they can still be distinguished by a C program and thus need to exist. As part of this work, the original formalization was added back to Flocq, in the module `IEEE754.BinarySingleNaN`.

Next, following the same methodology as that of the `Uint63` formalization, we have declared an abstract type and some arithmetic operations:

```
Primitive float := #float64_type.
Primitive add := #float64_add. (* and so on *)
```

Here, the `Primitive` vernacular amounts to introducing `Parameters` (*i.e.*, axiomatic symbols) that the kernel maps to hardware operations whenever they are fully applied to concrete floating-point values, as explained in Section 3. This is different from the original "retroknowledge" mechanism [13], which would have first defined some concrete data types and operations over it, and then hardcoded some reduction rules for these operations (thus sidestepping their original definitions).

After declaring various axiomatic symbols to manipulate `float` values (*e.g.*, `is_nan`, `of_uint63`), both types `float` and `spec_float` are endowed with two mappings `Prim2SF` and `SF2Prim`, implemented as regular definitions:

```
Definition Prim2SF (f : float) : spec_float := (* body omitted *).
Definition SF2Prim (f : spec_float) : float := (* body omitted *).
```

Unlike `Prim2SF` which is injective, the function `SF2Prim` will typically map many `spec_float` to the same `float`, since `spec_float` enforces no constraint at all on the range of the mantissa and exponents. We thus have the following axioms:

```
Axiom SF2Prim_Prim2SF : ∀ p : float, SF2Prim (Prim2SF p) = p.
Axiom Prim2SF_valid : ∀ p : float, valid_binary (Prim2SF p).
Axiom Prim2SF_SF2Prim :
  ∀ s : spec_float, valid_binary s -> Prim2SF (SF2Prim s) = s.
```

Finally, the computational content of all the operators is axiomatized with respect to reference implementations of the algorithms over the `spec_float` type. For example, in the case of the addition, we first define a naive[5] implementation `SFadd` for any precision `prec` and maximal exponent `emax`:

```
Definition SFadd (prec emax : Z) (x y : spec_float) :=
  match x, y with
  | S754_nan, _ | _, S754_nan => S754_nan
  | S754_infinity sx, S754_infinity sy =>
    if Bool.eqb sx sy then x else S754_nan
  | S754_infinity _, _ => x | _, S754_infinity _ => y
  | S754_zero sx, S754_zero sy =>
    if Bool.eqb sx sy then x else S754_zero false
```

[5]The naivety of the approach can be seen in that both mantissas `mx` and `my` are aligned by shifting them left according to the smallest exponent `min ex ey`. A fast implementation would instead shift them right according to the largest exponent, as done in hardware. This is much trickier to get right and the improved performance is not required for our specification purpose.

```
  | S754_zero _, _ ⇒ y | _, S754_zero _ ⇒ x
  | S754_finite sx mx ex, S754_finite sy my ey ⇒
    (* let ez be the minimum of ex and ey                    *)
    (* shift mx and my to the left, i.e., mx := mx * 2^(ex - ez) *)
    (* take signs into account:  mx := (-1)^sx * mx; idem for my *)
    (* round the result (mx + my) * 2^ez to nearest             *)
  end.
```

Then, an axiom relates the axiomatic symbol `add` to this reference implementation instantiated for the `binary64` format (`prec` = 53 and `emax` = 1024):

```
Axiom add_spec :
  ∀ x y, Prim2SF (add x y) = SFadd 53 1024 (Prim2SF x) (Prim2SF y).
```

Users wanting to load all floating-point operations and axioms just need to `Require Import Floats.` In addition, to enjoy decimal literals and notations without explicit quoting, users can `Open Scope float_scope.`

The implementation provides the following floating-point primitives. First, come some arithmetic operations: "`+`", "`-`" (opposite and subtraction), "`*`", "`/`", `abs`, and `sqrt`. All these operations are rounded to nearest, ties breaking to even.

Second, some comparison functions return a boolean result: "`=?`", "`<?`", "`<=?`". These comparison functions comply with the IEEE-754 standard, that is, comparing to a NaN is always false. In addition, there is a generic comparison function "`?=`" that returns a four-valued result: `FEq`, `FLt`, `FGt`, or `FNotComparable` (in case one or both inputs are NaN). There is also a `classify` function that tells whether a number is NaN, zero, infinite, normal, or subnormal, and except for NaN, what its sign is. Note that there is a lot of redundancy between the comparison functions "`=?`", "`<?`", "`<=?`" and "`?=`" as most could be emulated using the others. It is difficult to choose which ones are more useful than others, so we ended up providing hardcoded reduction rules for all of them.

Finally, there are some dedicated operations to manipulate numbers. The function `of_int63` rounds a 63-bit unsigned integer to the nearest floating-point number, ties breaking to even. Conversely, `normfr_mantissa` takes a number in $[0.5, 1.0)$ and returns its integral mantissa. There are also functions `frshiftexp` and `ldshiftexp` that behave like `frexp` and `ldexp` from the standard library of the C language. The first one decomposes a floating-point number into a mantissa (*i.e.*, a floating-point number f such that $0.5 \leq |f| < 1$) and an integer exponent e. The second one is the converse operation which computes $f \cdot 2^e$. There is a small peculiarity though, as Coq's hardware integers were unsigned at the time hardware floating-point numbers were implemented (signed integers have since been added). So, the exponent e is represented with a bias, to make it non-negative. The functions `next_up` and `next_down` return the successor and predecessor of a given floating-point value.

The reference implementation amounts to about 500 lines of Coq code in the standard library. But let us recall that the material provided in the `Floats` module is purely algorithmic. It does not axiomatize nor prove any meaningful floating-point properties and is thus basically useless in isolation.

2.2 On Flocq's side

Comprehensive formalizations of floating-point arithmetic exist for several proof assistants, *e.g.*, HOL Light [14] and PVS [15, 16]. In the case of Coq, the largest formalization is provided by the Flocq library [7]. A whole hierarchy of formats is provided, ranging from real numbers with bounded mantissas but unbounded exponents to computable numbers with all the floating-point special values: signed zeros, infinities, and NaNs. Along with these formats and the links between them, the library contains many classical results about roundings and error-free transformations.

When verifying properties of floating-point algorithms, two families of formats are commonly encountered:

- Numbers with an unbounded exponent range, *i.e.*, without underflow nor overflow. Although unrealistic, this model is attractive for its simplicity and commonly used for error bounds [17].
- Numbers with an exponent range only lower bounded, *i.e.*, with underflow but without overflow. This is slightly more realistic, since overflows can often be studied separately, while this is usually much harder for underflows [6].

To make Coq's basic floating-point specification useful, we need to establish a link with one of Flocq's formats, namely the `binary_float` type. This is basically a dependent product of `spec_float` and a proof that the mantissa and exponent are bounded:

```
Inductive binary_float :=
  | B754_zero (s : bool)
  | B754_infinity (s : bool)
  | B754_nan : binary_float
  | B754_finite (s : bool) (m : positive) (e : Z) :
    valid_binary (S754_finite s m e) -> binary_float.
```

The Coq theory `Flocq.IEEE754.PrimFloat`[6] provides two functions `Prim2B : float -> binary_float` and `B2Prim : binary_float -> float` that convert back and forth between values of Coq's abstract type `float` and Flocq's concrete type `binary_float`. These conversion functions act as morphisms, as illustrated by the following theorem.

```
Theorem add_equiv :
  ∀ x y, Prim2B (x + y) = Bplus mode_NE (Prim2B x) (Prim2B y).
```

In the above theorem, the `+` operator on the left stands for Coq's hardware `float` addition whereas `Bplus mode_NE` on the right stands for Flocq's addition rounded to nearest, with ties breaking to even, which is the default rounding mode of the IEEE-754 standard.

The main purpose of these morphisms is to give access to Flocq's theorems which state that floating-point operations amount to rounding the corresponding operation in the real field $\mathbb{R}$, as mandated by the IEEE-754 standard. For instance, Flocq provides a formal proof of the following theorem:

[6] https://gitlab.inria.fr/flocq/flocq/-/blob/flocq-4.1.1/src/IEEE754/PrimFloat.v

```
Theorem Bplus_correct : ∀ m x y, is_finite x -> is_finite y ->
  let z : R := round radix2 fexp (round_mode m) (B2R x + B2R y) in
  if Rlt_bool (Rabs z) (bpow radix2 emax) then
    B2R (Bplus m x y) = z     (* no overflow *)
  else
    B2SF (Bplus m x y) = binary_overflow m (Bsign x)
```

This theorem mainly states that, if `x` and `y` are finite floating-point numbers, the real value of their floating-point sum `(Bplus m x y)` is exactly the rounding `z` of the mathematical addition `(B2R x + B2R y)` of `x` and `y` seen as real numbers, assuming the addition did not overflow. The actual theorem in Flocq also gives the sign of the result, which is useful to distinguish $+0$ and -0, but it is omitted here for the sake of conciseness.

By relating the addition of real numbers with the addition of floating-point numbers, this theorem brings confidence in the correctness of the non trivial bit-level specification of floating-point operations described in Section 2.1, at least for finite inputs. For infinite and NaN inputs, exhaustive testing is achievable [12]. Moreover, this theorem gives access to the extensive set of results proved in the Flocq library. This includes cases where floating-point operations are exact or where the round-off error is represented as a floating-point number or bounded. The latter enables the use of the standard model of floating-point arithmetic to derive bounds on errors of elaborate expressions or algorithms [6, 18]. This will be used in Section 4.1 to perform efficient proofs by reflection, combining floating-point computations and such proofs bounding their round-off errors.

Building this link between Flocq's formalization and Coq's specification of hardware floating-point numbers has provided the opportunity to add to the `IEEE754` layer of Flocq several new functions, such as the Boolean comparisons `Beqb`, `Bltb`, and `Bleb`, which complement the already available and more general `Bcompare` function. Some other added functions provide ways to precisely craft or destruct floating-point values from their integer mantissa and exponent: `Bnormfr_mantissa`, `Bldexp`, and `Bfrexp`. Of particular interest for interval arithmetic are the predecessor and successor functions `Bpred` and `Bsucc` as well as the unit in the last place `Bulp`. Finally, the two constants `Bone` and `Bmax_float` are provided for convenience.

In terms of implementation, all the above changes to Flocq required adding 4900 lines and removing 1600 lines.

3 Implementation

While the reference implementations described in Section 2.1 can effectively perform floating-point operations, they are excruciatingly slow. So, we want to delegate them to hardware units instead. Section 3.1 shows how the kernel of Coq was extended to make it possible.

To minimize the trusted computing base, only the default rounding direction of the IEEE-754 standard is supported by Coq, as it remains the most portable one. Unfortunately, applications such as interval arithmetic depend on the availability of directed rounding modes. Section 3.2 explains how the functions predecessor and successor can be used instead, and how the performance overhead was mitigated.

While floating-point numbers are usually hidden deep inside proofs of theorems about real numbers, it might happen that the user wants to directly manipulate floating-point numbers. To cover this use case, our implementation also provides some facilities for parsing and printing numbers (Section 3.3).

Finally, this whole work would be moot if the implementation did not match the specification described in Section 2.1. So, Section 3.4 discusses the issue of trust and what has been done to offer the highest level of guarantee.

3.1 Reduction engines

As explained in Section 2.1, Coq provides an abstract type `float` as well as operations over it, and Coq's version of λ-calculus is extended with dedicated reduction rules for these operations. This means that these rules have to be implemented in the software. Unfortunately, Coq supports various engines, each one with its own implementation of the rules of Coq's calculus.

First, there is the conversion engine, which is responsible of checking that two λ-terms are equivalent according to the rules of the calculus. The conversion engine works great in general, but it falls short when performing a proof by computational reflection. Indeed, in that case, the archetypal goal to prove is $f(x) = \mathit{true}$ for some given x. This is a proof by reflexivity, which means that Coq has to check that the term $f(x) = \mathit{true}$ is equivalent to the term $\mathit{true} = \mathit{true}$. Since true is already in normal form, this amounts to computing the normal form of $f(x)$. But the conversion engine tries hard to never normalize terms, as the size of the normal form explodes in the general case. So, it is unable to handle this use case efficiently.

That is why Coq provides two reduction engines that are solely designed to compute normal forms. The first one, invoked by the `vm_compute` tactic, compiles the λ-term to bytecode and then evaluates it using an interpreter derived from the bytecode interpreter for the OCaml language [19]. The second one, invoked by the `native_compute` tactic, follows a similar approach. It turns the λ-term into an OCaml function, compiles it to machine code using the OCaml compiler, and then loads it and executes it [20].

For all three engines, the underlying runtime comes from the OCaml language. As a consequence, we chose to represent floating-point numbers the same way as OCaml, if only to please the garbage collector. Thus, floating-point numbers in Coq are boxed, that is, they are represented by a pointer to an allocated memory cell containing a single floating-point number.

The implementation of the arithmetic operations, however, is not as straightforward, as we cannot just follow OCaml's guidance. Indeed, contrarily to standard OCaml functions, Coq λ-terms can contain free variables, which are irreducible. In turn, even if the type of a λ-term is `float`, it does not mean that its reduced value is a floating-point number, it might be an arbitrary irreducible expression. So, we follow the same approach as in previous works for hardware integer support [9, 10]. The implementation first checks if the inputs are actual floating-point numbers. Thanks to the boxing format, this information is readily available. If the inputs are numbers, the floating-point operation is performed. Otherwise, an irreducible term is built from the inputs and the operation.

This causes a discrepancy with the OCaml runtime. Indeed, while floating-point numbers are usually boxed, the runtime optimizes the representation of floating-point arrays, so that they directly store numbers rather than pointers to boxes. This is problematic for Coq, since irreducible terms of floating-point types are not numbers and thus cannot be stored as unboxed values. The original implementation failed to circumvent this runtime optimization, which led to an inconsistency, as explained in Section 3.4.

3.2 Rounding directions

As mentioned above, only rounding to nearest is supported by our implementation. Yet, the IEEE-754 standard specifies some other rounding directions, some of which are especially useful for proofs by computation. Let us illustrate this with interval arithmetic. This arithmetic reliably approximates a real expression x with a pair $(\underline{x}, \overline{x})$ of floating-point numbers that enclose it [3]. Given some enclosures of u and v, an enclosure of $u + v$ is then represented by the pair $(\underline{u} + \underline{v}, \overline{u} + \overline{v})$. To make sure this is an actual enclosure, the lower bound $\underline{u} + \underline{v}$ is computed using a floating-point addition rounded toward $-\infty$, while the upper bound is rounded toward $+\infty$. Indeed, simply rounding to nearest would be unsound; for instance, $1 \in [0; 1]$ and $2^{-80} \in \left[0; 2^{-80}\right]$ whereas the real sum $1 + 2^{-80}$ does not lie in $\left[0 \oplus 0; 1 \oplus 2^{-80}\right] = [0; 1]$, denoting by $\oplus$ the floating-point addition rounded to nearest. A similar approach is used for multiplication, division, and so on.

Unfortunately, while rounding to nearest is readily available in floating-point units, this is not the case for directed rounding. And even if it was, there is no foolproof way of performing floating-point operations with directed rounding in programming languages such as OCaml and C. For instance, as of writing this article, the GCC compiler still does not handle dynamically changing the rounding mode in a safe way, let alone an efficient one.[7] So, to ensure portability, we had to stop at rounding to nearest.

But since interval arithmetic is such an ubiquitous paradigm when it comes to proving properties about real numbers, we have provided two more primitives to ease its implementation: predecessor and successor. Indeed, interval arithmetic does not really care whether the bounds of the enclosures are correctly rounded toward $-\infty$ and $+\infty$. If the lower bound is even smaller (or the upper bound larger), the enclosure is still valid, though less tight. In other words, this might cause some proofs to fail, but cannot lead to unsound results. Thus, rather than rounding $\underline{u} + \underline{v}$ toward $-\infty$ to compute the lower bound, an interval library can instead round $\underline{u} + \underline{v}$ to nearest and then take its floating-point predecessor.

The original implementation of these two primitives was a trivial wrapper over the `nextafter` function of the C standard library [8]. Unfortunately, neither LLVM nor GCC properly optimize it, even if the second operand is an infinite constant. So, we

[7] https://gcc.gnu.org/bugzilla/show_bug.cgi?id=34678

replaced `nextafter` by our own specific implementation,[8] which brought a noticeable speedup.

Still, performance of the bytecode interpreter (`vm_compute`) was poor, due to the boxing of floating-point numbers. Indeed, any arithmetic operation would cause two memory allocations in a row, one for the result of the floating-point operation rounded to nearest, and another one for the predecessor or successor. A simple way to fix it would have been to provide variants of all the arithmetic operations composed either with the predecessor or the successor, that is, ten more floating-point primitives. The C implementation of these new primitives would then be able to elide the first allocation.

But adding ten more primitives to Coq just for the sake of one single library felt wasteful. So, we explored a different approach. Instead of eliding the first allocation, we elide the second one in some specific cases. Indeed, if the first allocation is no longer needed once the predecessor/successor has been computed (and thus would be collected by the garbage collector), these functions can reuse it to store their own result. This case can easily be detected by the peephole optimizer of the bytecode compiler.[9] As a consequence, immediately computing the predecessor/successor of a floating-point result is now so cheap that removing the calls to these functions from CoqInterval (which would be unsound) does not bring any noticeable speedup, and thus neither would having some dedicated primitives.

3.3 Parsing and printing

Parsing and particularly printing the floating-point constants in Coq appeared to be a non-trivial point. Coq basically offers two levels for printing terms. The most commonly used level applies library- and user-defined notations to display concise terms that are as readable as possible. At the lower level, terms are displayed in a raw form to ensure that no details are hidden by some notation. The user can switch between the two levels with the `Set` and `Unset Printing All` commands. Printing of floating-point constants now fully supports both levels.

As suggested by its name, the `binary64` floating-point format, implemented by processors and wired into Coq's hardware floating-point numbers, is a binary format. This means that its finite values are the rational numbers $m \times 2^e$ for bounded integers m and e. In the `binary64` format, the mantissa m is encoded on 53 bits while the exponent e is encoded on 11 bits. Thus, all finite values can be exactly encoded in the standard hexadecimal format `[+-]0x⟨m⟩p[+-]⟨e⟩` where `⟨m⟩` is an hexadecimal encoding of m, spreading on at most 14 characters, and `⟨e⟩` a decimal encoding of e, taking at most 4 characters. This is the way floating-point values are displayed as raw terms, offering an exact and compact display, with at most 23 characters.

Unfortunately, this hexadecimal printing lacks readability for humans, used to work with decimal representations. It is worth noting that 10 being a multiple of 2, any binary value can be exactly represented as a decimal one. Indeed, when $e \geq 0$, the number $m \times 2^e$ is an integer and when $e < 0$, we have $m \times 2^e = (m \times 5^{-e}) \times 10^e$ that is a decimal value since $m \times 5^{-e}$ is an integer. For the `binary64` format, $|m| \leq 2^{53} - 1$ and $-1074 \leq e \leq 971$, which means that, in the first case, the integer

[8] https://github.com/coq/coq/pull/12959/commits/ef3ec53e4f74f32a705489b332b037569680d28e
[9] https://github.com/coq/coq/pull/12959/commits/5c7f63fa7a88cf2cb9b6837eb2797268c5843030

$m \times 2^e$ can be represented with at most 309 digits and in the second case, the integer $m \times 5^{-e}$ can be represented with at most 767 digits. Thus, although an exact decimal representation exists, using up to nearly 800 characters to display numeric constants is utterly unpractical. However, it is known that printing values in a decimal format with at least 17 significant digits and implementing parsing as a rounding to nearest guarantees that no information is lost [21, Table 3.16]. More precisely, the following theorem holds:

Theorem 1 (17 digits is enough)**.** *For any floating-point value x in the binary floating-point format with 53 bits of precision and gradual underflows at some minimal exponent e_{min}, then for any $x' \in \mathbb{R}$ such that*

$$|x - x'| \leq \frac{10^{\lceil \log_{10}(x) \rceil - 17}}{2},$$

we have $\circ(x') = x$ with $\circ$ any rounding to nearest in the format of x.

Note that any rounding to nearest x' of any $x \in \mathbb{R}$ with 17 significant digits satisfies the above hypothesis. We prove this theorem in Coq as follows.[10]

```
(* if [printed_x] is a rounding to nearest of binary64 value [x] with
   at least 17 digits, then parsing [printed_x] with a radix2 rounding
   to nearest yields back [x]. *)
Theorem print17 : ∀ emin tie,
  ∀ x, 0 < x -> generic_format radix2 (FLT_exp emin prec) x ->
  ∀ printed_x,
  Rabs (x - printed_x) <= /2 * bpow radix10 (mag radix10 x - 17) ->
  round radix2 (FLT_exp emin prec) (Znearest tie) printed_x = x.
```

This is in fact a corollary of the following more generic lemma.

Lemma 1. *Let $\mathbb{F}$ be the floating-point format with gradual underflow, radix β, minimal exponent e_{min}, and precision p. For any radix β' and any precision p' such that*

$$\beta^p < \beta'^{p'-1}$$

then for any floating-point value $x \in \mathbb{F}$ and any $x' \in \mathbb{R}$ such that

$$|x - x'| \leq \frac{\beta'^{\lceil \log_{\beta'}(x) \rceil - p'}}{2},$$

we have $\circ(x') = x$ with $\circ$ any rounding to nearest in $\mathbb{F}$.

Note that 17 is the smallest integer p' such that $2^{53} < 10^{p'-1}$. Printing with 17 decimal digits is thus the choice made in the default printing mode. This means that one can verify the following script:

```
Goal (0.99999999999999999 = 1)%float.
Proof. reflexivity. Qed.
```

[10] https://gitlab.inria.fr/flocq/flocq/-/blob/flocq-4.1.1/examples/Print17.v

Indeed, the constants `0.99999999999999999` and `1` are the same, as clearly seen when displaying the goal under its raw form: `@eq float 0x1p+0%float 0x1p+0%float`. To avoid any surprise to the user, a warning is displayed by Coq: *The constant 0.99999999999999999 is not a binary64 floating-point value. A closest value 0x1p+0 will be used and unambiguously printed 1.*

Regarding the implementation, as of Coq 8.14, the parsing relies on the OCaml function `float_of_string` and the printing on `printf "%.17g"` for decimal and `printf "%h"` for hexadecimal display. All these OCaml functions are themselves mostly wrappers on the corresponding libc functions. An alternative, that may be preferred in the future, would be to use Coq's `Number Notation` mechanism which provides a way to write parsers and printers for numeric constants directly as Coq functions. This would enable to perform some correctness proofs on those parsing and printing functions, at the cost of a slower parsing and printing since interpreting Coq functions is many times slower than executing natively compiled OCaml code. As a nice side effect, such Coq implemented parser and printer could provide a generic implementation that would work for other formats than `binary64`, for instance `binary32`. Andrew Appel offered a proof of concept[11] of such an implementation.

3.4 Soundness

When we originally reported on the implementation of hardware floating-point numbers in Coq [8], we had identified three main potential threats to soundness. We remind them below, along with a discussion of the few soundness bugs that were discovered since the merge of the feature in Coq in November 2019. All known bugs are now fixed. It is worth noting that all these bugs pertain more to the usual implementation mishaps than fundamental issues and that barring such implementation bugs, the approach is theoretically sound.

Specification issues

A mismatch with respect to the implementation would break the soundness. Of course, such errors in the specification can only be harmful when the offending axioms are used (we recall that all the axioms used in a proved theorem explicitly appear in the result of the Coq command `Print Assumptions`). So far, two such bugs have been reported and fixed:

- incorrect specification of `PrimFloat.leb`,[12]
- inconsistent classification of zeros.[13]

Both bugs were due to some typo in the specification, and it should be noted that the former would now be automatically spotted, thanks to a new warning raised by Coq for unused variables in pattern-matching.

As seen in Section 2.1, our Coq specification happens to be executable (although this can be pretty slow). This allowed us to add to Coq's test-suite some consistency

[11] https://github.com/coq/coq/issues/14782#issuecomment-906480643
[12] https://github.com/coq/coq/issues/12483
[13] https://github.com/coq/coq/issues/16096

checks between the specification and the implementation. These checks, however, can obviously not be exhaustive.

Incompatible implementations

If evaluation tactics (`native_compute`, `vm_compute`, `compute`, `simpl`, `hnf`, and so on) were to evaluate a same term to different results depending on the hardware, this would lead to a proof of `False`. In particular, it happens that the payload of NaNs is not fully specified by the IEEE-754 standard (different hardwares can produce different NaNs for a same computation), so we have chosen to consider all NaNs as equal and not distinguish them. Thus incompatible bit-level implementations remain compatible at the logical level.

Double roundings due to the legacy x87 unit on old 32-bit architectures could also be harmful [22]. The OCaml compiler systematically relies on it when it is available, which led us to implement all floating-point operators in C for 32-bit architectures, and use the appropriate compiler flags. To double-check the absence of double roundings, we have also added a runtime test[14] to prevent Coq from running in case of miscompilation.

As with the specification, some tests of consistency between the various evaluation mechanisms have been added to Coq's test-suite.

Incorrect convertibility test

Distinguishing two values that should not be distinguished, or vice versa, would also be a threat. In particular, implementing the convertibility test using the equality test on floating-point values (as defined in the IEEE-754 standard) would be wrong, as not only NaNs cause issues, but also signed zeroes. Indeed, the standard equates -0 and $+0$, while $1 \div (-0) = -\infty \neq 1 \div (+0) = +\infty$. Fortunately enough, a very simple implementation is feasible; it amounts to the following OCaml code:

```
let equal f1 f2 =
  if f1 = f2 then f1 <> 0. || sign f1 = sign f2
  else is_nan f1 && is_nan f2
```

When `f1` and `f2` compare equal in the guard, they are either nonzero, and are then the same floating-point value, or they can be $+0$ or -0 which are distinguished by the `then` branch. Otherwise, `f1` and `f2` are different floating-point values unless they are both NaNs, since `nan = nan` is false by the IEEE-754 equality. The `else` branch thus checks for that case.

Interaction with other primitive types in Coq

This can also be a source of soundness issues. To be more precise, unsigned hardware integers were completely reworked in February 2019 (9 months before floating-point numbers); primitive persistent arrays were added in July 2020; finally, signed hardware integers were added in February 2021. First, the fact that the hardware integers used in the specification of hardware floating-point numbers were unsigned did require some care. Next, the way OCaml optimizes arrays of floating-point values did raise a

[14]https://github.com/coq/coq/blob/V8.17.0/kernel/float64_63.ml#L28-L35

few issues during the development, although it seemed unlikely that such bugs could lead to a proof of `False`. This nonetheless happened with the introduction of primitive persistent arrays that were developed concurrently to hardware floating-point numbers, and whose interactions had not been properly tested before then.[15] Lastly, a similar bug involving the OCaml binary representation of floating-point arrays and the `native_compute` mechanism, but independent from primitive arrays, was discovered and fixed.[16]

4 Applications

A few Coq libraries rely on intensive floating-point computations to formally prove properties involving real numbers. We have adapted two of them to take advantage of the hardware support for floating-point arithmetic. The first one is ValidSDP [23], which uses floating-point numbers to verify positivity certificates. As detailed in Section 4.1, the adaptation was rather straightforward, since ValidSDP was already formalized in a way that made it robust against underflow and overflow.

This was not the case for the second library, CoqInterval [5], which uses pairs of floating-point numbers to enclose real numbers. Indeed, CoqInterval was performing computations using a floating-point format with unbounded exponents and, in numerous places, the proofs were implicitly relying on some of its non-standard properties. Making this formalization compatible with IEEE-754 floating-point numbers required some large and intrusive changes, as explained in Section 4.2.

4.1 ValidSDP

The ValidSDP library implements a reflexive tactic to automatically prove inequalities involving multivariate polynomials with real coefficients [23]. It follows the so-called skeptical (or certifying) approach, calling external Semi-Definite Programming (SDP) solvers and formally proving the considered polynomial positivity claims by relying on the witness provided by the solvers.

The implementation relies on the following theorem, formally verified in Coq, where the constants ε and η bound the errors when rounding to nearest. More precisely, the error of a floating-point operator is bounded by the relative bound ε when its result is a normal value and by the absolute bound η when it is a subnormal value.

Theorem 2 (Corollary 2.4 in [24])**.** *For $A \in \mathbb{R}^{n\times n}$, let c be*

$$\frac{(n+1)\varepsilon}{1-(n+1)\varepsilon}\,\mathrm{tr}(A) + 4n\left(2(n+1) + \max_i A_{ii}\right)\eta.$$

If the Cholesky decomposition of $A - c \cdot I_n$ succeeds with floating-point numbers (i.e., without taking the square root of a negative value), then the Cholesky decomposition of A with real numbers would succeed as well, which implies that A is positive definite.

[15] https://github.com/coq/coq/issues/15070
[16] https://github.com/coq/coq/issues/17871

The ValidSDP tactic reifies the positivity goal to obtain a multivariate polynomial p, then calls a SDP solver to obtain a matrix A such that when A is positive definite, $p \geq 0$. The above theorem is then used to check that A is definite positive [23].

The implementation thus requires a floating-point kernel, and unlike the proof goals discharged by the CoqInterval tactic, the accuracy of the floating-point operators involved is important, as shown by the use of ε and η in Theorem 2. This floating-point kernel was initially the emulated one of CoqInterval, providing an effective implementation of floating-point operators, using rounding-to-nearest most of the time, and occasionally, rounding to $+\infty$.

The ValidSDP formalization gathers several slightly different definitions of the type of floating-point numbers, seen either as a subtype of the real numbers `R` (an axiomatized type from Coq's standard library) or as a subtype of rational numbers (a concrete type allowing for effective computation).

• `FS_of`: a dependent record gathering every $v \in \texttt{R}$ satisfying $\texttt{format}(v)$, for a given function `format` of type $\texttt{R} \to \texttt{bool}$.
• `Float_spec`: a dependent record gathering every format satisfying the standard floating-point model without overflow. It includes a rounding operator $\circ : \texttt{R} \to \texttt{FS\_of format}$ and the bounds $\varepsilon \in \texttt{R}$ and $\eta \in \texttt{R}$. For instance, the axiom for the multiplication reads:

$$\forall x, y : \texttt{FS\_of format},\ \exists d, \exists e, \begin{cases} |d| \leq \frac{\varepsilon}{1+\varepsilon}, \\ |e| \leq \eta, \\ de = 0, \\ \circ(xy) = (1+d)xy + e. \end{cases}$$

• `Float_infnan_spec`: a dependent record gathering an instance of the previous structure, a concrete type `FIS`—that can contain NaNs and infinities—a projection `FIS2FS` from `FIS` to `FS_of format`, a Boolean predicate `finite` that holds for finite floating-point numbers of `FIS`, and concrete arithmetic operators over `FIS` with their specification, linking them to the axiomatized operators for finite values. Thus, this structure formalizes overflow handling as well as effective computation.
• `Float_round_up_infnan_spec`: a dependent record gathering an instance of the previous structure, along with `fieps`, `fieta` (overapproximations of ε and η) and addition, multiplication, division with upward rounding.

Once these structures are defined, the following instances are provided:

• `flx64`: an instance of `Float_spec` with $\eta = 0$ matching Flocq's $\texttt{FLX}(\texttt{prec}{=}53)$ format, corresponding to 53-bit floating-point numbers without NaNs, overflows, nor underflows.
• `binary64`: an instance of `Float_spec` matching Flocq's $\texttt{FLT}(\texttt{prec}{=}53, e_{\min}{=}-1074)$ format, corresponding to the IEEE 754 Binary64 format, still without NaNs nor overflows.
• `binary64_infnan`: an instance of `Float_infnan_spec` extending `binary64` with an effective implementation taken from Flocq's `IEEE754.BinarySingleNaN`.

• `coqinterval_infnan`: an instance of `Float_infnan_spec` extending `flx64` with an efficient implementation relying on CoqInterval's `Float.Specific_ops`.
• `coqinterval_round_up_infnan`: an instance of `Float_round_up_infnan_spec`, also relying on CoqInterval.

It happens that the already formalized structures were precise enough to account for underflows, overflows, and NaNs. This made the porting to hardware floating-point numbers smooth, since we just needed to implement the following two instances, without needing to alter the `Float_*_spec` structures:

• `primitive_floats_infnan`: an instance of `Float_infnan_spec` extending `binary64` with an efficient implementation relying on from `Coq.Floats.Floats`.[17]
• `primitive_float_round_up_infnan`: an instance of `Float_round_up_infnan_spec`, also relying on `Coq.Floats.Floats`.

4.2 CoqInterval

Since version 4.0, the CoqInterval library uses hardware floating-point numbers by default. This implementation of interval arithmetic lives alongside the original implementation, which emulates floating-point numbers using arbitrary-precision integers `bigZ` from the bignums library [25]. Two such integers are used, one for the mantissa and the other one for the exponent. A special value NaN is provided, but infinities are not (thanks to the unbounded exponent, neither overflow nor underflow are possible). Every operation is performed at a given target precision, *i.e.*, number of figures in the resulting mantissa. Let us consider addition as an illustration. The original interface of the operation was as follows in the `FloatOps` module type.

```
Parameter add : rounding_mode -> precision -> type -> type -> type.
```

The argument of type `rounding_mode` is chosen among one of four rounding modes: `rnd_UP`, `rnd_DN`, `rnd_ZR`, and `rnd_NE`. In the same module type, the addition was equipped with an exact specification:

```
Parameter add_correct : ∀ mode p x y,
  toX (add mode p x y)
  = Xround radix mode (prec p) (Xadd (toX x) (toX y)).
```

The function `toX` is an injection from the type `type` of floating-point numbers into a type of real numbers extended with an exceptional value `Xnan`.

Interval arithmetic relies mostly on the `rnd_DN` and `rnd_UP` rounding modes to compute respectively lower and upper bounds of intervals. The exact rounding of the bounds, however, is not required for correctness. Thus, as explained in Section 3.2, we can instead round to nearest and then pick the predecessor `next_down` or successor `next_up` on floating-point numbers to approximate directed rounding. The specification `add_correct`, however, no longer holds, as the results might be off by one. So, the `add` function above has been dropped and replaced in the `FloatOps` module type by two separate functions:

[17]The authors would like to thank Guillaume Bertholon who implemented the original version of `primitive_float_infnan`.

```
Parameter add_UP : precision -> type -> type -> type.
Parameter add_DN : precision -> type -> type -> type.
```

The precision argument has been kept, as it still governs the operations on emulated floating-point numbers. But for hardware floating-point numbers, it is ignored and always assumed to be 53.

We now need to specify those operators. The extended real numbers $\mathbb{R} \cup \{$`Xnan`$\}$ are kept for specification, in order to minimize changes. Thus, `toX` maps $-\infty$ and $+\infty$ to `Xnan` which remains interpreted as $-\infty$ or $+\infty$, depending on whether it is used as the lower or upper bound of an interval. One would like to write the following straightforward specification, where `le_upper` is essentially a “less than or equal to” operator on extended reals.

```
Parameter add_UP_correct :
  ∀ p x y, le_upper (Xadd (toX x) (toX y)) (toX (add_UP p x y))).
```

However, `add_UP`$(-\infty)(-\infty) = -\Omega$ (with Ω the largest finite floating-point number) is not an upper bound of `Xadd`(`toX`$(-\infty)$)(`toX`$(-\infty)$) = `Xadd Xnan Xnan` = `Xnan`, which plays the role of $+\infty$ when used as an upper bound. We thus need to ensure that `add_UP` is never called on $-\infty$ (and `add_DN` is never called on $+\infty$). Under this assumption, one can specify, and later prove, that `add_UP` yields an upper bound. We take the opportunity to specify that it never returns $-\infty$ and the specification finally reads as follows.

```
Parameter add_UP_correct :
  ∀ p x y, valid_ub x = true -> valid_ub y = true
  -> (valid_ub (add_UP p x y) = true
      ∧ le_upper (Xadd (toX x) (toX y)) (toX (add_UP p x y))).
```

The predicate `valid_ub x` means that `x` can be used as the upper bound of an interval, *i.e.*, that it is not $-\infty$. The other operators, `sub`, `mul`, `div`, and `sqrt`, receive a similar specification. Adapting the implementations of this `FloatOps` module type required some easy but tedious work as many proofs needed to be updated.

In order to guarantee that `add_UP` is never called on $-\infty$, one could test `valid_ub` before every call but that would have a performance impact. Fortunately, `add_UP` is only used to compute upper bounds of intervals (and symmetrically, `add_DN` is used for lower bounds). So, we just need to forbid “ill-formed” intervals $[+\infty; x]$ or $[x; -\infty]$. This could be done using dependent types, by adding to every interval a proof that it is well formed. Instead, we chose to modify the interpretation of intervals, which was originally as follows, where `Inan` contains all the extended real numbers, including `Xnan`, whereas `Ibnd l u` encodes the interval $[l; u] \subseteq \mathbb{R}$.

```
Definition convert xi :=
  match xi with
  | Inan => Interval.Inan
  | Ibnd l u => Interval.Ibnd (toX l) (toX u)
  end
```

The new interpretation gives every ill-formed interval, like $[42; -\infty]$ or $[+\infty; 42]$, the semantics of an empty interval:

```
Definition convert xi :=
  match xi with
  | Inan => Interval.Inan
  | Ibnd l u =>
    if F.valid_lb l && F.valid_ub u then Interval.Ibnd (toX l) (toX u)
    else Interval.Ibnd (Xreal 1) (Xreal 0)  (* empty interval [1;0] *)
  end
```

As a consequence, correctness theorems are now vacuously true when ill-formed intervals are passed to interval functions. Although this required many modifications to the `FloatInterval` module implementing basic interval operators, only minimal changes had to be applied in the higher layers of the library.

Finally, a few other adaptations were needed as some functions were incompatible with overflows, or simply with the fixed precision of hardware floating-point numbers:

- Function `fromZ` was originally an exact injection from $\mathbb{Z}$ into floating-point numbers. This is no longer possible, so most of its uses have been replaced with calls to the new functions `fromZ_DN` and `fromZ_UP`. The original function `fromZ` is still available, so as to easily compute constants in algorithms, but it is now specified only on the small interval $[-256; 256]$.
- The `scale` function was specified as performing exact multiplication or divisions by powers of 2 since neither overflow nor underflow could happen with unbounded exponents. This was mostly used in heuristics, so the function was kept with a best-effort implementation and without any specification. A `div2` function was added for the few cases where exact divisions by 2 were needed, but it is specified only for inputs larger than $\frac{1}{256}$. Finally, a `midpoint` function was added such that `midpoint x y` returns a value in the interval $[x; y]$. In practice, the result is close to the middle of the interval, but this property is left unspecified as it is not used in correctness proofs.
- Exact operations (addition, subtraction, and multiplication), using as much precision as needed to exactly encode their result, were used mostly to compute some constants (typically integers). So, the latter were replaced with tight intervals containing the constants.

With this new coarser `FloatOps` interface, it was relatively easy to implement hardware floating-point numbers in CoqInterval. Regarding the implementation, 9700 lines were added to the library (including 3300 for the hardware-based implementation of the interface `FloatOps`) while 4300 were removed.

The use of hardware floating-point numbers provides a speedup of about one order of magnitude, as illustrated by the following proof of

$$\left|\int_0^8 \sin(t + \exp t)dt - 0.3474\right| \le 0.1.$$

```
From Coq Require Import Reals.
From Coquelicot Require Import Coquelicot.
From Interval Require Import Tactic.
```

```
Open Scope R_scope.

Goal Rabs ((RInt (fun t => sin (t + exp t)) 0 8) - 0.3474) <= 0.1.
Proof.
Time integral with (i_fuel 600, i_prec 30).
(* emulated 30-bit FP computations: 32.7s *)
Undo.
Time integral with (i_fuel 600).
(* hardware FP computations: 3.7s; 9x faster! *)
Qed.
```

The main drawback stems from the fact that we do not use directed rounding modes but instead approximate them by composing rounding to nearest with successor or predecessor, as explained in Section 3.2. Thus, proofs that relies on correct directed rounding cannot be performed with hardware operations, as illustrated by the following example.

```
Goal 1 + 1 <= 2.
interval with (i_prec 1).
(* exact rounding with emulated FP works (even with a single bit) *)
Undo.
Fail interval.
(* but hardware FP fails as next_up(1 + 1) is larger than 2 *)
```

This happened to yield a disastrous effect on proofs involving square roots such as follows.

```
Goal ∀ x, -1 <= x <= 1 -> sqrt (1 - x) <= 3/2.
Proof. intros; interval. Qed.
```

Indeed, since the interval computed for `1 - x` now contains negative values (*e.g.*, the predecessor of zero), the interval version of `sqrt` started returning `Inan`, which is the interval containing all the real numbers as well as `Xnan`. We have alleviated this issue by modifying the interval square root so that it ignores negative inputs. This is made possible because the `sqrt` function in Coq's standard library defines `sqrt x = 0` for any negative `x`, rather than some arbitrary result.

Many proofs, however, do not rely on exact rounding and work just as well with hardware floating-point numbers as with emulated ones. For the few cases where exact rounding is required, users can tell the tactics to fall back to emulated numbers by providing an explicit precision with the `i_prec` argument.

5 Benchmarks

We have evaluated the benefit of implementing reduction rules using hardware floating-point arithmetic for the applications ValidSDP presented in Section 4.1 and CoqInterval presented in Section 4.2.

The experimental results of the upcoming sections have been obtained using a Debian GNU/Linux workstation based on an Intel Core i7-7700 CPU clocked at

Table 1 Proof time for the reflexive tactic `posdef_check` using `vm_compute`. Every test is run 5 times. The table indicates the average and relative variability among the timings of 5 runs.

Source	Emulated	Hardware	Speedup
mat0050	0.228s ±9.5%	0.013s ±16.7%	17.3×
mat0100	1.451s ±2.4%	0.113s ±8.2%	12.9×
mat0150	4.572s ±6.8%	0.276s ±13.0%	16.6×
mat0200	10.724s ±3.4%	0.557s ±9.8%	19.2×
mat0250	21.839s ±1.2%	1.032s ±3.5%	21.2×
mat0300	37.706s ±1.6%	1.810s ±4.7%	20.8×
mat0350	60.616s ±1.5%	2.802s ±4.1%	21.6×
mat0400	89.343s ±1.5%	4.110s ±0.9%	21.7×

Table 2 Proof time for the reflexive tactic `posdef_check` using `native_compute`. Every test is run 5 times. The table indicates the average and relative variability among the timings of the 5 runs.

Source	Emulated	Hardware	Speedup
mat0050	0.702s ±1.4%	0.602s ±4.2%	1.2×
mat0100	1.034s ±2.3%	0.674s ±2.5%	1.5×
mat0150	1.735s ±2.7%	0.694s ±2.6%	2.5×
mat0200	3.207s ±5.6%	0.836s ±4.5%	3.8×
mat0250	5.624s ±4.2%	0.924s ±2.2%	6.1×
mat0300	9.359s ±4.1%	1.080s ±3.3%	8.7×
mat0350	14.524s ±3.6%	1.307s ±3.8%	11.1×
mat0400	21.650s ±3.0%	1.641s ±5.3%	13.2×

3.60 GHz, with 16 GB of RAM. All the benchmarks have been executed sequentially (namely, without the `-j` option of `make`).

Our benchmarks rely on a large set of dependencies that can take about an hour to compile. For greater convenience, we devised some Docker images containing the benchmark environments, based on Debian 11, opam 2 (the OCaml package manager), and OCaml 4.07.1. The source code of all the benchmarks as well as guidelines to run them are gathered at the following URL: https://github.com/validsdp/benchs-primitive-floats.

5.1 Benchmark with ValidSDP 1.0.1 and Coq 8.15

We first run the `posdef_check` tactic on ValidSDP's test-suite and compare its execution time between emulated and hardware floating-point numbers. The results are displayed in Table 1 for `vm_compute` and Table 2 for `native_compute`.

Notice that the measured speedups are far from the three order of magnitudes separating emulated floating-point operations from equivalent OCaml implementations. From the above results, it appears that arithmetic operators constitute most of the computation time when using emulated numbers (at least 95% with `vm_compute`) but

Table 3 Computation time for individual operations obtained by subtracting the CPU time of a normal execution from that of a modified execution where the specified operation is performed twice (resp. 1001 times). Every test is run 5 times. The table indicates the average and relative variability among the timings of the 5 runs.

Op	compute	Emulated CPU times (Op×2−Op)	Hardware CPU times (Op×1001−Op)	Speedup
add	vm	101.54±1.6% – 77.91±1.2%	163.50±0.5% – 4.12±0.9%	148×
mul	vm	116.68±1.5% – 77.91±1.2%	163.54±0.5% – 4.12±0.9%	243×
add	native	25.08±2.0% – 20.10±4.8%	88.67±2.2% – 1.66±0.9%	57×
mul	native	29.13±1.2% – 20.10±4.8%	92.79±1.7% – 1.66±0.9%	99×

nothing tells us this is still the case with hardware numbers. In fact, with hardware numbers, most of the computation time is spent on manipulating lists as our matrices are implemented using them [26]. This could be improved using primitive "persistent arrays" now that they have been integrated in Coq.

To get an idea of the time actually devoted to floating-point arithmetic in the total proof time of our reflexive tactic, we use the following simple methodology: replace every arithmetic operator over emulated floating-point numbers with a version that uselessly performs the computation twice, then measure the execution time of both the original program (denoted "Op" in Table 3) and the modified program (denoted "Op×2"). The difference between both timings gives the cost of performing only the arithmetic operations. In the case of hardware operations, they are performed 1001 times instead, as the difference would otherwise be of the same order of magnitude as the variability of the measured timings.

Note that the redundant computations involved in the modified program are not implemented with a mere additional let-in such as

```
let m1 := mul a b in let m2 := mul a b in m2
```

because both Coq's bytecode compiler and the OCaml native compiler optimize away the unused local definition. Fortunately, this doubling trick can be enabled by going through an extra function call `select m1 m2` with

```
Definition select (a b : F.type) := a.
```

The results are given in Table 3 for `vm_compute` and `native_compute`. The tested operations are addition and multiplication, as they constitute the vast majority of the arithmetic computations performed during a Cholesky decomposition.

It is worth noting that these speedups should be taken as coarse orders of magnitude rather than precise measurements. Indeed, the time difference "Op×N−Op" also includes the cost of the duplication machinery itself, *i.e.*, some extra function calls such as `select`. As a consequence, the speedups are presumably largely underestimated, as this extra cost is far from negligible in the case of hardware floating-point arithmetic ("Op×1001−Op").

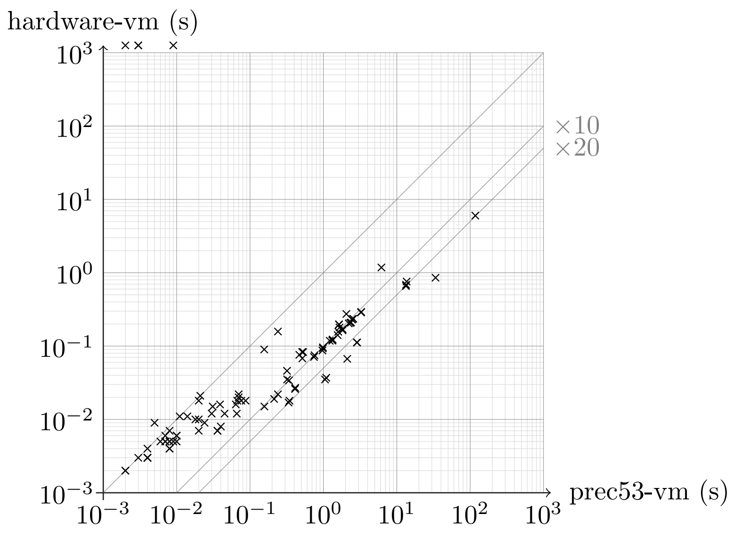

Fig. 1 Comparison of proof times between hardware and emulated 53-bit floating-point arithmetic using `vm_compute`. The graph uses a log-log scale, so diagonals represent equivalent speedups. Out of the 101 examples, 4 proofs fail with hardware numbers due to the pessimistic outward rounding, as explained at the end of Section 4.2. The corresponding points appear at the top of the graph.

5.2 Benchmark with CoqInterval 4.5.2 and Coq 8.15

The second benchmark comprises 101 mathematical properties verified using the CoqInterval library. Fig. 1 shows that, except for the shortest examples, hardware floating-point numbers usually offer a 10× to 20× speedup over emulated computations when the latter are performed at the corresponding precision, *i.e.*, 53 bits.

Earlier versions of CoqInterval, however, were using a default precision of 30 bits, which is sufficient for a large number of examples from the benchmark. As shown in Fig. 2, this made proofs up to twice faster in that case, but nowhere near the speedup achieved using hardware floating-point numbers.

Fig. 3 shows that using the `native_compute` reduction mechanism instead of `vm_compute` can bring a 3× speedup, but only for the longer examples. Indeed, `native_compute` performs an invocation of the OCaml compiler, which incurs a systematic latency. In particular, hardware floating-point numbers bring such a large speedup over emulated ones that using `native_compute` is often detrimental, except for a handful of examples, as shown in Fig. 4.

Finally, Fig. 5 shows that the 10× to 20× speedup from hardware floating-point numbers observed in Fig. 1 also holds when using `native_compute` instead of `vm_compute`, but only for the longest examples. For shorter examples, the native OCaml compilation dominates the timings, so any speedup brought by hardware floating-point numbers goes unnoticed.

In addition to the figures described above, Appendix A provides the raw data for all the timings involved.

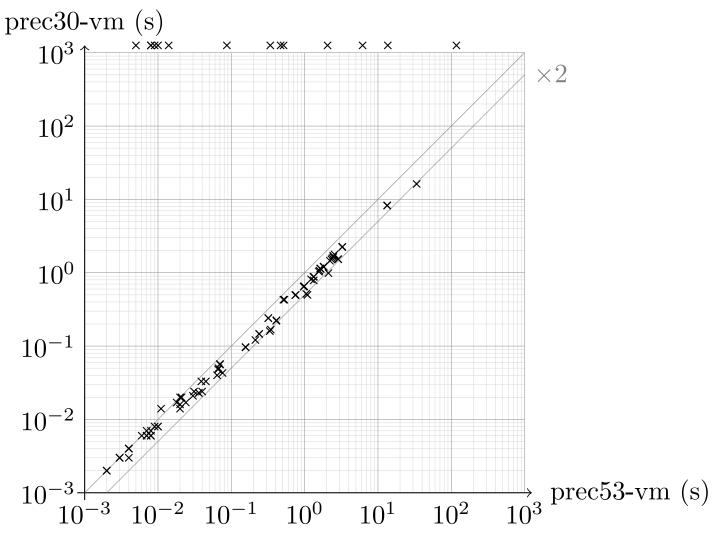

Fig. 2 Comparison of proof times between emulated 53-bit and 30-bit floating-point arithmetic using `vm_compute`. Out of the 101 examples, 14 proofs fail with the reduced precision.

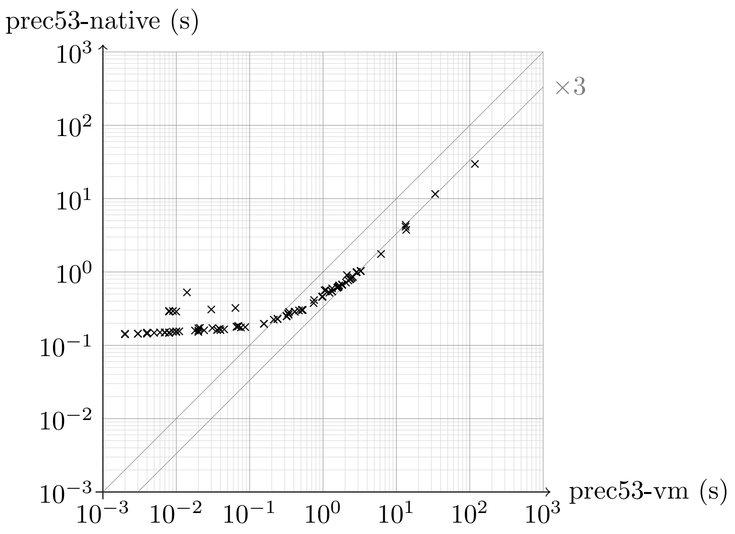

Fig. 3 Comparison of proof times for emulated 53-bit floating-point arithmetic between `vm_compute` and `native_compute`.

6 Conclusion

The work described in this article explains how support for hardware floating-point numbers was added to the Coq proof assistant. Formally proving properties of real numbers using floating-point computations is nothing new, but up to now, these computations were slowly emulated in the logic of Coq [5]. Given that modern processors

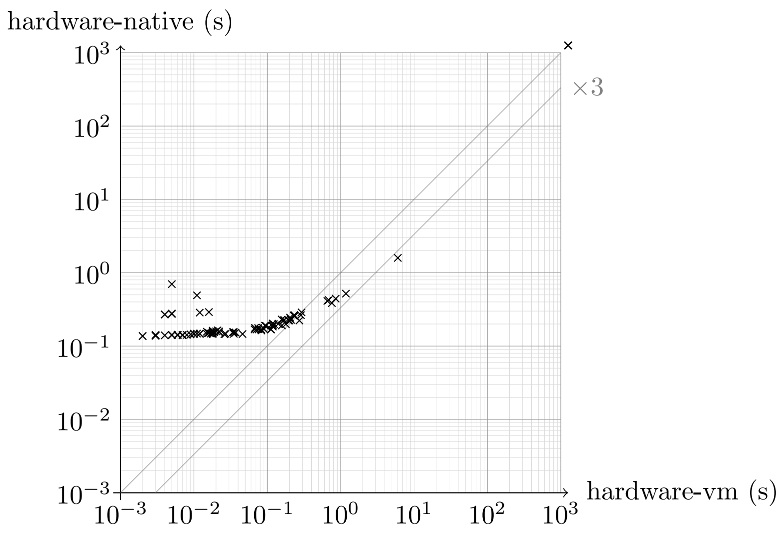

Fig. 4 Comparison of proof times for hardware floating-point arithmetic between `vm_compute` and `native_compute`. The 4 proofs that fail in both cases appear as a cluster at the top right of the graph.

come with a floating-point unit whose semantics are specified by the IEEE-754 standard [1], such an emulation is a waste of computational resources. The same motivation had already led to delegating arithmetic on 31-bit integers (and later 63-bit integers) to hardware units [9]. This work follows a similar approach for floating-point computations: the three conversion/reduction engines of Coq have been extended, so as to use the processor whenever floating-point inputs are not open terms.

While the approach is similar on the implementation side, there is a large difference on the specification side. Indeed, while both integer and floating-point computations are axiomatized using their operational semantics, floating-point arithmetic is so peculiar that one should not blindly believe that the semantics expressed in the logic of Coq matches the behavior of the floating-point hardware. To restore the trust in the formal system, this operational semantics has been proved equivalent to that of the Flocq library, which had already been proved to comply with the IEEE-754 standard [7, 12]. But as usual with any software implementation, a few bugs were introduced along the way. Those are now fixed and, barring such implementation bugs, the approach is theoretically sound and does not allow any incorrect proof.

Since the IEEE-754 standard relates floating-point computations to infinitely precise ones, *i.e.*, real numbers, the theorems from Flocq make it easy to use hardware floating-point numbers to formally prove properties on real numbers. There are two main ways to do so. One is to formalize a careful error analysis of floating-point computations, as in the ValidSDP library [6]. The other is to use directed rounding, as in the CoqInterval library [5]. Our work accommodates both approaches. But in the latter case, directed rounding is only approximate, which has forced us to rewrite large parts of CoqInterval to enable the use of hardware floating-point numbers.

Thanks to this work, some proofs by computational reflection have been sped up by a factor 10. While they would have necessarily been run offline before, some of

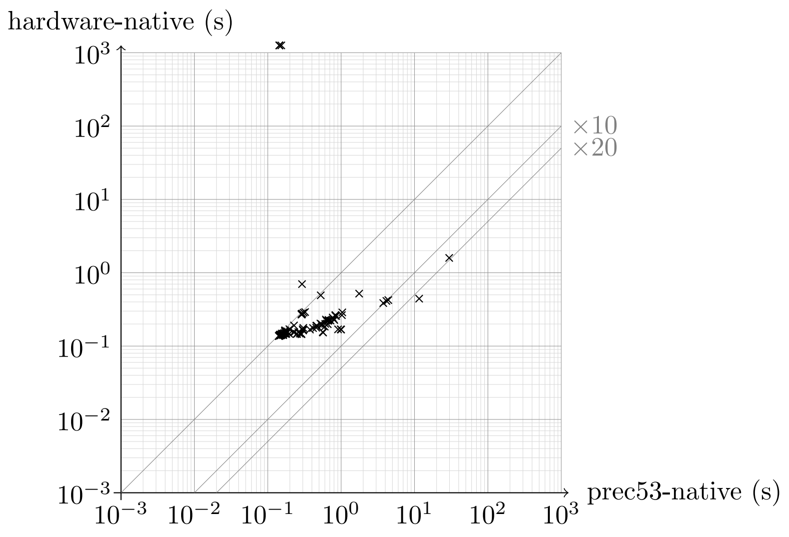

Fig. 5 Comparison of proof times between hardware and 53-bit emulated floating-point arithmetic using `native_compute`. The 4 proofs that fail with hardware numbers appear as a cluster at the top of the graph.

them can now be performed in an interactive setting. This makes for a much friendlier user experience when tactics fail earlier. This speedup comes at the expense of a larger trusted computing base, but the opposite could also be said of this work. Indeed, the speedup is so large that the point of the `native_compute` mechanism becomes largely moot for this use case, thus making it possible to greatly reduce the trusted computing base.

Hardware floating-point arithmetic is not a panacea though, as the numbers are constrained to a bounded range of exponents and, more importantly, a precision of 53 bits. Whenever a higher precision is needed, the tactics provided by, *e.g.*, CoqInterval have to fall back to emulating floating-point arithmetic. But this problem is nothing new, and whichever solution the scientific computing community comes with, we hope it can be adapted to a proof assistant.

Acknowledgements

We are grateful to Guillaume Bertholon for his early work on extending the kernel of Coq with hardware floating-point numbers. This work was partly supported by the European Research Council (ERC) under the European Union's Horizon 2020 Research and Innovation Programme – Grant Agreement n°101001995. This work was partly supported by the NuSCAP project (ANR-20-CE48-0014) of the French national research agency (ANR).

References

[1] IEEE Computer Society: IEEE standard for floating-point arithmetic. Technical

Report 754-2008, IEEE (August 2008). https://doi.org/10.1109/IEEESTD.2008.4610935

[2] Tucker, W.: A rigorous ODE solver and Smale's 14th problem. Foundations of Computational Mathematics **2**, 53–117 (2002) https://doi.org/10.1007/s002080010018

[3] Moore, R.E.: Interval Analysis. Prentice-Hall, Englewood Cliffs, NJ, USA (1963)

[4] Rump, S.M.: Verification methods: Rigorous results using floating-point arithmetic. Acta Numerica **19**, 287–449 (2010) https://doi.org/10.1017/S096249291000005X

[5] Martin-Dorel, É., Melquiond, G.: Proving tight bounds on univariate expressions with elementary functions in Coq. Journal of Automated Reasoning **57**(3), 187–217 (2016) https://doi.org/10.1007/s10817-015-9350-4

[6] Roux, P.: Formal proofs of rounding error bounds – with application to an automatic positive definiteness check. Journal of Automated Reasoning **57**(2), 135–156 (2016) https://doi.org/10.1007/s10817-015-9339-z

[7] Boldo, S., Melquiond, G.: Flocq: A unified library for proving floating-point algorithms in Coq. In: Antelo, E., Hough, D., Ienne, P. (eds.) 20th IEEE Symposium on Computer Arithmetic, Tübingen, Germany, pp. 243–252 (2011). https://doi.org/10.1109/ARITH.2011.40

[8] Bertholon, G., Martin-Dorel, É., Roux, P.: Primitive floats in Coq. In: Harrison, J., O'Leary, J., Tolmach, A. (eds.) 10th International Conference on Interactive Theorem Proving. Leibniz International Proceedings in Informatics, vol. 141, pp. 7–1720. Portland, OR, USA (2019). https://doi.org/10.4230/LIPIcs.ITP.2019.7

[9] Armand, M., Grégoire, B., Spiwack, A., Théry, L.: Extending Coq with imperative features and its application to SAT verification. In: Kaufmann, M., Paulson, L.C. (eds.) 1st International Conference on Interactive Theorem Proving. Lecture Notes in Computer Science, vol. 6172, pp. 83–98. Edinburgh, UK (2010). https://doi.org/10.1007/978-3-642-14052-5_8

[10] Dénès, M.: Towards primitive data types for Coq 63-bits integers and persistent arrays. In: 5th Coq Workshop, Rennes, France (2013). https://coq.inria.fr/files/coq5_submission_2.pdf

[11] Boldo, S., Jourdan, J.-H., Leroy, X., Melquiond, G.: A formally-verified C compiler supporting floating-point arithmetic. In: Nannarelli, A., Seidel, P.-M., Tang, P.T.P. (eds.) 21st IEEE Symposium on Computer Arithmetic, Austin, TX, USA, pp. 107–115 (2013). https://doi.org/10.1109/ARITH.2013.30

[12] Boldo, S., Jourdan, J.-H., Leroy, X., Melquiond, G.: Verified compilation of

floating-point computations. Journal of Automated Reasoning **54**(2), 135–163 (2015) https://doi.org/10.1007/s10817-014-9317-x

[13] Spiwack, A.: Verified computing in homological algebra. PhD thesis, École Polytechnique, Palaiseau, France (2011). https://tel.archives-ouvertes.fr/pastel-00605836

[14] Harrison, J.: A machine-checked theory of floating point arithmetic. In: Bertot, Y., Dowek, G., Hirschowitz, A., Paulin-Mohring, C., Théry, L. (eds.) 12th International Conference in Theorem Proving in Higher Order Logics. Lecture Notes in Computer Science, vol. 1690, pp. 113–130. Nice, France (1999). https://doi.org/10.1007/3-540-48256-3_9

[15] Miner, P.: Defining the IEEE-854 floating-point standard in PVS. Technical Report 19950023402, NASA, Langley Research Center, Hampton, VA, USA (1995)

[16] Boldo, S., Munoz, C.: A high-level formalization of floating-point number in PVS. Technical Report 20070003560, NASA, National Institute of Aerospace, Hampton, VA, USA (2006)

[17] Higham, N.: Accuracy and Stability of Numerical Algorithms. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (1996)

[18] Jeannerod, C., Rump, S.M.: On relative errors of floating-point operations: Optimal bounds and applications. Mathematics of Computations **87**(310), 803–819 (2018) https://doi.org/10.1090/mcom/3234

[19] Grégoire, B., Leroy, X.: A compiled implementation of strong reduction. In: 7th ACM SIGPLAN International Conference on Functional Programming, Pittsburgh, PA, USA, pp. 235–246 (2002). https://doi.org/10.1145/581478.581501

[20] Boespflug, M., Dénès, M., Grégoire, B.: Full reduction at full throttle. In: 1st International Conference on Certified Programs and Proofs, Kenting, Taiwan, pp. 362–377 (2011). https://doi.org/10.1007/978-3-642-25379-9_26

[21] Muller, J.-M., Brunie, N., Dinechin, F., Jeannerod, C.-P., Joldes, M., Lefèvre, V., Melquiond, G., Revol, N., Torres, S.: Handbook of Floating-Point Arithmetic, 2nd edn. Birkhäuser, Basel (2018). https://doi.org/10.1007/978-3-319-76526-6

[22] Monniaux, D.: The pitfalls of verifying floating-point computations. ACM Transactions on Programming Languages and Systems **30**(3), 12–11241 (2008) https://doi.org/10.1145/1353445.1353446

[23] Martin-Dorel, É., Roux, P.: A reflexive tactic for polynomial positivity using numerical solvers and floating-point computations. In: Bertot, Y., Vafeiadis, V. (eds.) 6th ACM SIGPLAN Conference on Certified Programs and Proofs, Paris,

France, pp. 90–99 (2017). https://doi.org/10.1145/3018610.3018622

[24] Rump, S.M.: Verification of positive definiteness. BIT Numerical Mathematics **46**, 433–452 (2006)

[25] Grégoire, B., Théry, L.: A purely functional library for modular arithmetic and its application to certifying large prime numbers. In: Furbach, U., Shankar, N. (eds.) 3rd International Joint Conference on Automated Reasoning. Lecture Notes in Computer Science, vol. 4130, pp. 423–437. Seattle, WA, USA (2006). https://doi.org/10.1007/11814771_36

[26] Cohen, C., Dénès, M., Mörtberg, A.: Refinements for free! In: Gonthier, G., Norrish, M. (eds.) 3rd International Conference on Certified Programs and Proofs. Lecture Notes in Computer Science, vol. 8307, pp. 147–162. Melbourne, Australia (2013). https://doi.org/10.1007/978-3-319-03545-1_10

Appendix A Full Data for CoqInterval Benchmarks

In the table below, the string `452` stands for the CoqInterval version (*i.e.*, 4.5.2). The source code for these benchmarks, along with guidelines to rerun them, can be found at the following URL: https://github.com/validsdp/benchs-primitive-floats.

Problems	452-bigz-prec53	452-bigz-prec30	speedup	452-primfloat	speedup	452-bigz-prec53-native	speedup	452-bigz-prec30-native	speedup	452-primfloat-native	speedup
Lemma	time (s)	time (s)		time (s)		time (s)		time (s)		time (s)	
Chebyshev	6.173	N/A		1.179	5.2x	1.76	3.5x	N/A		0.519	11.9x
MT1	2.112	0.988	2.1x	0.067	31.5x	0.908	2.3x	0.55	3.8x	0.169	12.5x
MT10	1.052	0.514	2.0x	0.035	30.1x	0.561	1.9x	0.359	2.9x	0.155	6.8x
MT11	1.094	0.499	2.2x	0.037	29.6x	0.568	1.9x	0.356	3.1x	0.153	7.2x
MT12	1.657	1.133	1.5x	0.179	9.3x	0.657	2.5x	0.498	3.3x	0.199	8.3x
MT13	33.727	16.222	2.1x	0.855	39.4x	11.569	2.9x	5.992	5.6x	0.443	76.1x
MT14	0.02	0.014	1.4x	0.007	2.9x	0.152	0.1x	0.145	0.1x	0.141	0.1x
MT15	0.076	0.043	1.8x	0.018	4.2x	0.175	0.4x	0.155	0.5x	0.146	0.5x
MT16	0.04	0.024	1.7x	0.008	5.0x	0.161	0.2x	0.148	0.3x	0.142	0.3x
MT16__1	0.008	0.007	1.1x	0.007	1.1x	0.147	0.1x	0.15	0.1x	0.141	0.1x
MT16__2	0.006	0.006	1.0x	0.005	1.2x	0.15	0.0x	0.142	0.0x	0.141	0.0x
MT16__3	0.007	0.007	1.0x	0.006	1.2x	0.15	0.0x	0.141	0.0x	0.141	0.0x
MT16__4	0.007	0.006	1.2x	0.005	1.4x	0.15	0.0x	0.142	0.0x	0.142	0.0x
MT17	0.036	0.023	1.6x	0.007	5.1x	0.162	0.2x	0.146	0.2x	0.142	0.3x
MT17'	0.036	0.023	1.6x	0.007	5.1x	0.162	0.2x	0.146	0.2x	0.141	0.3x
MT18	0.413	0.225	1.8x	0.027	15.3x	0.287	1.4x	0.214	1.9x	0.147	2.8x
MT18'	0.41	0.221	1.9x	0.026	15.8x	0.288	1.4x	0.212	1.9x	0.145	2.8x
MT19	0.24	0.146	1.6x	0.158	1.5x	0.228	1.1x	0.189	1.3x	0.191	1.3x
MT19'	0.24	0.147	1.6x	0.022	10.9x	0.23	1.0x	0.194	1.2x	0.154	1.6x
MT2	1.33	0.786	1.7x	0.119	11.2x	0.598	2.2x	0.425	3.1x	0.184	7.2x

Problems Lemma	452-bigz-prec53 time (s)	452-bigz-prec30 time (s)	speedup	452-primfloat time (s)	speedup	452-bigz-prec53-native time (s)	speedup	452-bigz-prec30-native time (s)	speedup	452-primfloat-native time (s)	speedup
MT20	2.881	1.529	1.9x	0.112	25.7x	1.001	2.9x	0.622	4.6x	0.168	17.1x
MT20'	2.853	1.529	1.9x	0.112	25.5x	0.987	2.9x	0.619	4.6x	0.169	16.9x
MT21	0.157	0.096	1.6x	0.015	10.5x	0.196	0.8x	0.174	0.9x	0.147	1.1x
MT21'	0.156	0.097	1.6x	0.09	1.7x	0.197	0.8x	0.174	0.9x	0.17	0.9x
MT22	0.319	0.24	1.3x	0.046	6.9x	0.247	1.3x	0.213	1.5x	0.146	2.2x
MT22'	0.32	0.241	1.3x	0.035	9.1x	0.251	1.3x	0.212	1.5x	0.147	2.2x
MT23	0.513	0.429	1.2x	0.083	6.2x	0.303	1.7x	0.269	1.9x	0.163	3.1x
MT23'	0.53	0.429	1.2x	0.083	6.4x	0.303	1.7x	0.268	2.0x	0.167	3.2x
MT24	0.008	0.006	1.3x	0.005	1.6x	0.15	0.1x	0.142	0.1x	0.142	0.1x
MT24'	0.007	0.006	1.2x	0.005	1.4x	0.149	0.0x	0.141	0.0x	0.141	0.0x
MT25	0.031	0.024	1.3x	0.015	2.1x	0.172	0.2x	0.161	0.2x	0.157	0.2x
MT3	0.335	0.16	2.1x	0.017	19.7x	0.273	1.2x	0.206	1.6x	0.148	2.3x
MT4	0.345	0.168	2.1x	0.018	19.2x	0.285	1.2x	0.21	1.6x	0.148	2.3x
MT5	0.03	0.021	1.4x	0.012	2.5x	0.308	0.1x	0.295	0.1x	0.286	0.1x
MT6	0.064	0.04	1.6x	0.016	4.0x	0.323	0.2x	0.3	0.2x	0.291	0.2x
MT7	0.01	0.008	1.2x	0.006	1.7x	0.152	0.1x	0.142	0.1x	0.141	0.1x
MT8	0.34	N/A		0.034	10.0x	0.261	1.3x	N/A		0.155	2.2x
MT9	0.47	N/A		0.076	6.2x	0.299	1.6x	N/A		0.169	2.8x
RD	0.014	N/A		0.011	1.3x	0.527	0.0x	N/A		0.492	0.0x
Rump_Tucker	117.45	N/A		6.012	19.5x	29.703	4.0x	N/A		1.593	73.7x
abs_err_atan	0.213	0.121	1.8x	0.019	11.2x	0.224	1.0x	0.187	1.1x	0.155	1.4x
adaptiveLV	0.011	0.014	0.8x	0.011	1.0x	0.155	0.1x	0.524	0.0x	0.148	0.1x
arctan_0_1	0.039	0.033	1.2x	0.016	2.4x	0.168	0.2x	0.154	0.3x	0.151	0.3x
bissect	2.054	N/A		0.274	7.5x	0.717	2.9x	N/A		0.223	9.2x

Problems Lemma	452-bigz-prec53 time (s)	452-bigz-prec30 time (s)	speedup	452-primfloat time (s)	speedup	452-bigz-prec53-native time (s)	speedup	452-bigz-prec30-native time (s)	speedup	452-primfloat-native time (s)	speedup
bug20120927	0.01	N/A		0.005	2.0x	0.288	0.0x	N/A		0.274	0.0x
bug20140723_1	0.008	N/A		0.004	2.0x	0.288	0.0x	N/A		0.269	0.0x
bug20140723_2	0.008	N/A		0.004	2.0x	0.292	0.0x	N/A		0.27	0.0x
bug20140728	0.002	0.002	1.0x	0.002	1.0x	0.142	0.0x	0.139	0.0x	0.137	0.0x
bug20150924	0.002	0.002	1.0x	N/A		0.144	0.0x	0.139	0.0x	N/A	
bug20150925	0.003	0.003	1.0x	N/A		0.144	0.0x	0.138	0.0x	N/A	
butcher	0.02	0.02	1.0x	0.018	1.1x	0.171	0.1x	0.162	0.1x	0.162	0.1x
circle	0.018	0.017	1.1x	0.01	1.8x	0.159	0.1x	0.149	0.1x	0.148	0.1x
cos_cos_d2	0.97	0.644	1.5x	0.087	11.1x	0.458	2.1x	0.362	2.7x	0.178	5.4x
cos_cos_d2__1	0.744	0.495	1.5x	0.071	10.5x	0.376	2.0x	0.306	2.4x	0.168	4.4x
cos_cos_d2__2	0.756	0.499	1.5x	0.075	10.1x	0.414	1.8x	0.318	2.4x	0.176	4.3x
cos_cos_d3	1.319	0.876	1.5x	0.119	11.1x	0.545	2.4x	0.449	2.9x	0.192	6.9x
cos_cos_d3__1	0.981	0.652	1.5x	0.093	10.5x	0.465	2.1x	0.373	2.6x	0.19	5.2x
cos_cos_d3__2	0.985	0.655	1.5x	0.096	10.3x	0.458	2.2x	0.373	2.6x	0.189	5.2x
cos_cos_d4	1.562	1.036	1.5x	0.141	11.1x	0.607	2.6x	0.478	3.3x	0.201	7.8x
cos_cos_d4__1	1.337	0.884	1.5x	0.123	10.9x	0.541	2.5x	0.445	3.0x	0.199	6.7x
cos_cos_d4__2	1.222	0.814	1.5x	0.118	10.4x	0.522	2.3x	0.412	3.0x	0.203	6.0x
cos_cos_d5	2.383	1.578	1.5x	0.21	11.3x	0.806	3.0x	0.623	3.8x	0.227	10.5x
cos_cos_d5__1	1.823	1.197	1.5x	0.164	11.1x	0.666	2.7x	0.53	3.4x	0.221	8.2x
cos_cos_d5__2	1.811	1.208	1.5x	0.169	10.7x	0.674	2.7x	0.531	3.4x	0.225	8.0x
cos_cos_d6	2.322	1.52	1.5x	0.207	11.2x	0.792	2.9x	0.611	3.8x	0.233	10.0x
cos_cos_d6__1	1.826	1.21	1.5x	0.17	10.7x	0.674	2.7x	0.53	3.4x	0.227	8.0x
cos_cos_d6__2	1.597	1.068	1.5x	0.157	10.2x	0.62	2.6x	0.497	3.2x	0.23	6.9x
cos_cos_d7	3.252	2.237	1.5x	0.287	11.3x	1.024	3.2x	0.8	4.1x	0.264	12.3x
cos_cos_d7__1	2.211	1.458	1.5x	0.204	10.8x	0.766	2.9x	0.597	3.7x	0.245	9.0x
cos_cos_d7__2	2.56	1.776	1.4x	0.236	10.8x	0.855	3.0x	0.685	3.7x	0.257	10.0x

Problems Lemma	452-bigz-prec53 time (s)	452-bigz-prec30 time (s)	speedup	452-primfloat time (s)	speedup	452-bigz-prec53-native time (s)	speedup	452-bigz-prec30-native time (s)	speedup	452-primfloat-native time (s)	speedup
cos_cos_d8	3.274	2.26	1.4x	0.293	11.2x	1.037	3.2x	0.809	4.0x	0.288	11.4x
cos_cos_d8__1	2.45	1.708	1.4x	0.23	10.7x	0.833	2.9x	0.669	3.7x	0.259	9.5x
cos_cos_d8__2	2.462	1.632	1.5x	0.233	10.6x	0.836	2.9x	0.649	3.8x	0.265	9.3x
example20071016_1	0.004	0.003	1.3x	0.003	1.3x	0.145	0.0x	0.141	0.0x	0.14	0.0x
example20071016_2	0.004	0.004	1.0x	0.004	1.0x	0.149	0.0x	0.14	0.0x	0.14	0.0x
example20071016_3	0.009	N/A		0.005	1.8x	0.293	0.0x	N/A		0.7	0.0x
example20071016_3’	0.005	N/A		0.009	0.6x	0.148	0.0x	N/A		0.144	0.0x
example20071016_4	0.066	0.048	1.4x	0.012	5.5x	0.179	0.4x	0.16	0.4x	0.147	0.4x
example20071016_5	0.07	0.056	1.2x	0.022	3.2x	0.181	0.4x	0.17	0.4x	0.158	0.4x
example20071016_6	0.004	0.004	1.0x	0.003	1.3x	0.144	0.0x	0.139	0.0x	0.139	0.0x
example20071016_7	0.067	0.05	1.3x	0.018	3.7x	0.179	0.4x	0.167	0.4x	0.157	0.4x
example20071016_8	0.009	0.008	1.1x	N/A		0.153	0.1x	0.144	0.1x	N/A	
example20120205_1	0.002	0.002	1.0x	0.002	1.0x	0.141	0.0x	0.14	0.0x	0.137	0.0x
example20120205_2	0.003	0.003	1.0x	N/A		0.144	0.0x	0.139	0.0x	N/A	
example20120205_3	0.003	0.003	1.0x	0.003	1.0x	0.144	0.0x	0.141	0.0x	0.138	0.0x
example20120205_4	0.004	0.004	1.0x	0.003	1.3x	0.145	0.0x	0.139	0.0x	0.142	0.0x
example20140221_1	13.612	N/A		0.759	17.9x	3.752	3.6x	N/A		0.386	35.3x
example20140221_2	0.087	N/A		0.018	4.8x	0.177	0.5x	N/A		0.15	0.6x
example20140610_1	13.331	8.219	1.6x	0.658	20.3x	4.138	3.2x	2.674	5.0x	0.413	32.3x
example20140610_2	1.631	1.142	1.4x	0.196	8.3x	0.629	2.6x	0.514	3.2x	0.222	7.3x
example20150105	0.516	N/A		0.068	7.6x	0.305	1.7x	N/A		0.177	2.9x
example_ln_1	0.004	0.004	1.0x	0.003	1.3x	0.149	0.0x	0.14	0.0x	0.14	0.0x
example_ln_2	0.008	0.006	1.3x	0.005	1.6x	0.295	0.0x	0.281	0.0x	0.278	0.0x
exp_0_3	0.024	0.017	1.4x	0.009	2.7x	0.16	0.1x	0.161	0.1x	0.144	0.2x
exp_cos_0_1	0.045	0.033	1.4x	0.012	3.8x	0.165	0.3x	0.155	0.3x	0.148	0.3x
h_54_ln_2	0.01	0.008	1.2x	0.006	1.7x	0.155	0.1x	0.143	0.1x	0.143	0.1x

Problems Lemma	452-bigz-prec53 time (s)	452-bigz-prec30 time (s)	speedup	452-primfloat time (s)	speedup	452-bigz-prec53-native time (s)	speedup	452-bigz-prec30-native time (s)	speedup	452-primfloat-native time (s)	speedup
magnetism	0.021	0.02	1.1x	0.021	1.0x	0.173	0.1x	0.166	0.1x	0.164	0.1x
rel_err_geodesic	13.374	8.293	1.6x	0.687	19.5x	4.395	3.0x	2.691	5.0x	0.424	31.5x
rel_err_geodesic'	1.63	1.141	1.4x	0.198	8.2x	0.634	2.6x	0.519	3.1x	0.224	7.3x
remez_sqrt	0.07	0.057	1.2x	0.02	3.5x	0.184	0.4x	0.171	0.4x	0.159	0.4x
x_ln1p_0_1	0.02	0.016	1.2x	0.01	2.0x	0.159	0.1x	0.148	0.1x	0.146	0.1x