VCFloat2: Floating-Point Error Analysis in Coq

Andrew W. Appel
Princeton University
USA
appel@princeton.edu

Ariel E. Kellison
Cornell University
USA
ak2485@cornell.edu

Abstract

The development of sound and efficient tools that automatically perform floating-point round-off error analysis is an active area of research with applications to embedded systems and scientific computing. In this paper we describe VCFloat2, a novel extension to the VCFloat tool for verifying floating-point C programs in Coq. Like VCFloat1, VCFloat2 soundly and automatically computes round-off error bounds on floating-point expressions, but does so to higher accuracy; with better performance; with more generality for nonstandard number formats; with the ability to reason about external (user-defined or library) functions; and with improved modularity for interfacing with other program verification tools in Coq. We evaluate the performance of VCFloat2 using common benchmarks; compared to other state-of-the art tools, VCFloat2 computes competitive error bounds and transparent certificates that require less time for verification.

CCS Concepts: • **Software and its engineering → Formal software verification**; • **Mathematics of computing → Numerical analysis**.

Keywords: floating point, round-off error analysis

ACM Reference Format:
Andrew W. Appel and Ariel E. Kellison. 2024. VCFloat2: Floating-Point Error Analysis in Coq. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '24), January 15–16, 2024, London, UK*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3636501.3636953

1 Introduction

We describe VCFloat2, an open-source[1] tool for automated floating-point round-off error analysis, foundationally verified sound in Coq, with a new functional modeling language for better integration with extended formal analyses of numerical programs, a new core representation of number formats that allows users to derive error bounds for formats not currently defined by the IEEE 754 standard, a new efficient optimizer for multinomial floating-point expressions, and a new analysis for obtaining tighter error bounds. VCFloat2 is available at github.com/VeriNum/vcfloat, or by opam in the coq-released repository as coq-vcfloat.

Developing machine-checked proofs that programs correctly and accurately approximate the solution to continuous mathematical problems is challenging. First, there is the challenge of developing machine-checked proofs of accuracy: while high-level mathematical specifications use real numbers and assume operations are exact, computer programs operate on floating-point numbers and perform inexact operations that introduce round-off error. Practically useful proofs of accuracy provide tight bounds on this round-off error without compromising soundness. Second, there is the challenge of composing proofs of correctness and accuracy—without introducing logical gaps—inside of a single mechanized logical framework. Using VCFloat2, we address these challenges by decomposing the problem as follows.

- Write a **floating-point functional model** —a floating-point-valued specification of the program—using the functional-modeling language provided by VCFloat2.
- Prove that a program written in a low-level language such as C correctly implements the floating-point functional model using a program logic, specified in Coq, for the low-level language. To help automate this proof, use a program logic verification tool, like the Verified Software Toolchain (VST) [2].
- Prove that the floating-point functional model approximates a **real-valued functional model**, to within a certain accuracy; that is, perform a *round-off error analysis,* using VCFloat2 for automation.
- Prove that the real-valued functional model finds a solution to the mathematical problem of interest, with a given accuracy bound. For this, use a Coq library for real analysis, such as Coquelicot [11] or Mathematical Components [33].

CPP '24, January 15–16, 2024, London, UK
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0488-8/24/01
https://doi.org/10.1145/3636501.3636953

[1]VCFloat2 is open source with the LGPL license, for which we thank: DARPA who funded the development of VCFloat1 and and encouraged the products of its sponsored research to be open-sourced; Jonathan M. Smith, who started DARPA on this open-source path in 2005 as a program manager; Ramananandro *et al.* [40] who open-sourced VCFloat1 in compliance with this policy; and Qualcomm, who (after acquiring the company that had developed VCFloat1) in 2022 released its residual rights. Development of VCFloat2 is supported in part by National Science Foundation grant CCF-2219757 and by the U.S. Department of Energy, Office of Advanced Scientific Computing Research, Computational Science Graduate Fellowship Award Number DE-SC0021110.

Using VCFloat2 as both a specification language and a tool for automating floating-point round-off error analysis, the proofs outlined above can be composed into a single correctness-and-accuracy theorem in Coq, with no logical gaps compromising soundness. While there are several tools that perform floating-point round-off error analysis, not all of them produce machine-checkable proofs that can be so seamlessly composed with other theorems about program correctness and real analysis; a detailed review of related work is provided in Section 15.

VCFloat2 is an extension of the original VCFloat tool, developed by Ramananandro *et al.* [40], with the following improvements:

Functional-modeling language: VCFloat1 took as input expressions from C programs parsed by the front end of the CompCert C compiler [10, 30]. In VCFloat2, we have added a new front end that can *reify*—that is, turn a formula into an abstract syntax tree for symbolic analysis—directly from functional models written in a general and natural style entirely independent of C or CompCert (see §5). In effect, this is a *functional modeling language* shallow-embedded in Coq for defining floating-point algorithms (§3).

Any-precision literals: To support VCFloat2's functional models, we added new floating-point literal notations within Coq that work at any floating-point precision—single (binary32), double (binary64), half, quad, or arbitrary user-specified (§3).

Modeling library: VCFloat2 provides a library containing definitions, lemmas, and congruences for reasoning about functional models and their equivalences.

User-defined operators: While VCFloat1 could only handle a specific set of built-in operators ($+, -, \times, \div, \sqrt{\ }$), VCFloat2 supports user-defined operators of any arity and precision, requiring only user-supplied proofs (or axioms) about their rounding behavior.

Non-IEEE formats: While VCFloat1 could handle any IEEE 754 format (with arbitrary mantissa and exponent sizes), VCFloat2 additionally supports user-supplied number formats, so long as the operations on them have defined (possibly zero) relative and absolute error bounds. We provide an example of this functionality using binary64 double-words ("double-doubles" [25]) where the operation under consideration sums a double-double number and a binary64 number.

Exact division: While VCFloat1 recognized that floating-point multiplication by nonnegative powers of 2 is exact in the absence of overflow, VCFloat2 obtains tighter error bounds by recognizing that floating-point division by powers of 2 (or multiplication by powers of 1/2) can also be exact, or in the case of underflow, highly accurate (§7).

Efficient simplifier: VCFloat1 workflow generated verification conditions which were discharged by Coq's standard solvers for polynomial and rational equations (e.g., the field_simplify tactic) followed by a call to Coq's Interval package [13, §4.2]. Even on standard floating-point benchmarks, these standard solvers generated exponentially large terms that caused Coq to run out of memory. For VCFloat2, we built an efficient, accurate, and proved correct (§9) interval-goal optimizer in Coq's term language (§8).

Decomposition tactic: To obtain tight error bounds using Coq's interval package on expressions generated by VCFloat, we implemented a tactic that decomposes subexpressions in order to mitigate the dependency effect in the interval analysis. With this decomposition tactic, VCFloat2 can provide tight bounds on benchmarks in instances where VCFloat1 failed to produce a useful bound.

Taken together, each of the contributions outlined above make VCFloat2 a flexible tool for building end-to-end machine-checked proofs of accuracy and correctness for floating-point programs. This is illustrated by several verification projects where VCFloat2 has played an integral role.

Applications. VCFloat2 has been used in several applications, demonstrating its effectiveness as a tool for building end-to-end proofs of accuracy and correctness.

Differential equations: Kellison and Appel [26] proved that a C program correctly and accurately integrates an ordinary differential equation (ODE). Using VST, the authors proved that a C program correctly implements a floating-point functional model. VCFloat2 was used to prove that the functional model accurately approximates a discrete-time-step real-valued functional model. Finally, Coquelicot was used to prove that the real-valued functional model finds an accurate solution to the equation with bounded error after N time-steps; all proofs and accuracy bounds were composed into a single Coq theorem.

Matrix-vector operations: The LAProof library [27] contains formal proofs of the accuracy of linear algebra operations described by the basic linear algebra subprograms (BLAS) specification, and provides a proof of correctness of a C implementation of sparse matrix-vector multiplication. These Coq theorems use VCFLoat2's functional-modeling language and modeling library, but do not use its automated error analysis.

Jacobi method: Tekriwal *et al.* [43] proved correctness, accuracy, and convergence of a stationary iterative solver for linear systems of equations. A sparse-matrix C program implementing the Jacobi method was proved correct using VST and using VCFloat2's modeling library; proofs of accuracy and convergence used the

Mathematical Components library and VCFloat2's modeling library.

VSTlib: C's standard math library is now axiomatized as a VST library using VCFloat2's functional-modeling language with the library functions (sin, cos, etc.) treated as VCFLoat2 user-defined operators [3]; VCFloat2 therefore supports automatic round-off-error analysis for computations using those functions.

In the remainder of this paper, VCFLoat2 is described in detail; the significance of VCFloat2 with respect to related work is described in Section 15. In the following section, we provide a review of round-off error analysis in order to set the stage.

2 Review of round-off error analysis

We briefly review the fundamentals of round-off error analysis and floating-point arithmetic, and point readers to more detailed expositions by Overton [38] and Muller et al. [35] for further reference.

A floating-point number may be 0, *normal finite*, *subnormal finite*, $+\infty$, $-\infty$, or *NaN* (not a number). A *normal* number is representable as $\pm 1.dddddd \cdot 2^e$, where the $dddddd$ represents a string of binary digits of length M (the mantissa size) and $e_{\min} \le e \le e_{\max}$ is the exponent. A *subnormal* number is representable as $\pm 0.dddddd \cdot 2^{e_{\min}}$, where there may be several leading zeros. A *finite* number is either ± 0, normal, or subnormal.

Every finite floating-point number x *exactly* represents a real number $\mathrm{R}(x)$. But the floating-point calculation $x + y$ or $x \cdot y$ is usually not exact: often $\mathrm{R}(x) + \mathrm{R}(y) \ne \mathrm{R}(x + y)$. That is, the result of a floating-point operation may have more mantissa bits than fit into the representation, so it will be *rounded off.* In general, if the result of evaluating x op y for op $\in \{+, -, \times, \div\}$ is a *normal* number, then we know that $\mathrm{R}(x \text{ op } y) = (\mathrm{R}(x) \text{ op } \mathrm{R}(y))(1 + \delta)$ for some δ such that $|\delta| \le 2^{-M}$; i.e., there is a *relative error bound.* If (x op y) is *subnormal*, then $\mathrm{R}(x \text{ op } y) = (\mathrm{R}(x) \text{ op } \mathrm{R}(y)) + \epsilon$, where $|\epsilon| \le 2^{-E}$ and $E = e_{\max} + M - 2$. For single precision $M = 24$ and $e_{\max} = 128$, so $|\delta| \le 2^{-24}$ and $|\epsilon| \le 2^{-150}$. If we know (x op y) will be finite but don't know whether it will be normal or subnormal, then $\mathrm{R}(x \text{ op } y) = (\mathrm{R}(x) \text{ op } \mathrm{R}(y))(1 + \delta) + \epsilon$ for some δ and ϵ bounded as above.

There are some special cases: if x and y have the same binary order of magnitude ($\frac{1}{2} \le x/y < 2$) then $\mathrm{R}(x - y) = \mathrm{R}(x) - \mathrm{R}(y)$ exactly; this is called *Sterbenz subtraction.* If $y = 2^k$ for $0 \le k$, $\mathrm{R}(x \cdot y) = \mathrm{R}(x) \cdot \mathrm{R}(y)$ exactly, so long as it doesn't overflow. If $y = 2^{-k}$, then $\mathrm{R}(x \cdot y) = \mathrm{R}(x) \cdot \mathrm{R}(y)$ exactly, so long as the result is normal; or $\mathrm{R}(x \cdot y) = \mathrm{R}(x) \cdot \mathrm{R}(y) + \epsilon$ if $x \cdot y$ is subnormal. Finally, if x and y are subnormal, then $\mathrm{R}(x+y) = \mathrm{R}(x)+\mathrm{R}(y)$ exactly, but currently VCFloat2 does not handle this special case, approximating as $\mathrm{R}(x + y) = \mathrm{R}(x) + \mathrm{R}(y) + \epsilon$.

3 Overview of VCFloat2

We will use this formula as a running example:

$$x + h \cdot (v + (h/2) \cdot (3 - x))$$

which arises in the simulation of a harmonic oscillator by leapfrog integration with time-step $h = 1/32$. Assume $2 \le x \le 4$ and $-2 \le v \le +2$, and we compute this in single-precision floating point.

VCFloat's job is to soundly derive an *absolute round-off error bound* for the formula, by inserting deltas and epsilons as described in Section 2, taking into account of all the special cases (known-normal numbers, known-subnormal numbers, multiplication by powers of two, Sterbenz subtraction, etc.).

We will illustrate the user's workflow using VCFloat2's new annotation system. The user writes in Coq,

```
Definition h := (1/32)%F32.
Definition F(x: ftype Tsingle) : ftype Tsingle :=
                Sterbenz(3.0−x)%F32.
Definition step (x v: ftype Tsingle) :=
                (Norm(x + h*(v+(h/2)*F(x))))%F32.
```

Our functional-modeling language is just Coq formulas and Coq functions over variables of type ftype(T), where T:type is any floating-point precision. (Don't confuse type, meaning a floating-point format, with Type, which is the notion of type in Coq's logic.) We predefine Tsingle:type and Tdouble:type, but the user can define any binary IEEE 754 floating-point format (e.g., half-precision or quad precision). So: Tsingle is a floating-point precision description, and the inhabitants of ftype Tsingle are single-precision floats.

VCFloat2's %F32 tag means that constants such as 1, 3.0, 38.571e-2 are to be parsed as single-precision (32-bit) floating-point, with + meaning single-precision addition. We also permit %F64 and the user can easily define, parametrically, constant notations and operators for any desired precision. Formulas may mix different precisions: ((1/2) * cast Tdouble (h * h)%F64)%F32 computes $h \cdot h$ in double precision and the rest in single precision.

VCFloat2 comes with these exciting new functions:

```
Definition Norm {A}(x: A) := x.
Definition Denorm {A}(x: A) := x.
Definition Sterbenz {A}(x: A) := x.
```

But these are hardly exciting at all: each one is just the identity function. Actually these are *annotations* for the analysis.[2] Norm(x+y) suggests the sum x+y is going to be a normal (not a subnormal) number. Denorm suggests a subnormal number. Sterbenz(x−y) suggests that x and y will have the same binary order of magnitude. These suggestions will have to be proved—at the point marked *(*B*)* below—but then they

[2]The idea of using annotations for reification that are semantically equivalent to identity functions was developed independently by Jason Gross [21, 23] (although those papers don't describe the technique so explicitly).

will lead to a better round-off error analysis. If these verification conditions can't be proved, then the whole round-off error theorem is unproved; one might need to remove the corresponding annotations.

Our running example is annotated with Norm and Sterbenz at specific places where it is expected that these conditions will be met. In future work we expect to automatically calculate many of these annotations, so users can write annotations only where they want to insist on the condition.

In general, an error analysis is valid only for a particular range of input values: the assumptions $2 \le x \le 4$ and $-2 \le y \le 2$ are important. The VCFloat2 user encodes these into a *boundsmap* that maps *identifiers* (denoted here as _x and _v) for the free variables of the expression (x and v in our example) to user-specified lower and upper bounds:

```
Definition _x : ident := 2%positive.
Definition _v : ident := 3%positive.
Definition step_bmap_list : list varinfo :=
  [ Build_varinfo Tsingle _x 2 4 ; Build_varinfo Tsingle _v (−2) 2 ].
Definition step_bmap : boundsmap :=
           a line of boilerplate mentioning step_bmap_list.
```

The theorem the user wants to prove: The round-off error of the step function is the maximum difference, for all x and v in bounds, between the floating-point evaluation of step x v and the evaluation of the same formula on the real numbers. To state this theorem, we use the notion of a valmap, a computable mapping from variable-identifiers to floating-point numbers. For example, given floats x and v we can build a valmap representing $\{\_x \mapsto x, \_v \mapsto v\}$.

We will prove that the round-off error is less than one in four million, for any valmap.[3] To prepare a functional-model formula like step for analysis, one reifies it (using some Coq boilerplate to invoke the reifier within a definition):

```
Definition step' := ltac:(
  let e' := HO_reify_float_expr constr:([_x; _v]) step in exact e').
```

Proving this theorem for the user is the main purpose of the tool:

```
Lemma prove_roundoff_bound_step:
  ∀ vmap : valmap,
    prove_roundoff_bound step_bmap vmap step' (1/4000000).
 (* Theorem statement: for all x and v in the bounds of step_bmap,
   the round–off error in computing [step x v] is less than 1/4000000. *)
Proof.
intros.
(*A*) prove_roundoff_bound.   This tactic leaves two subgoals
– prove_rndval.                    First subgoal
(*B*) all: interval.
– prove_roundoff_bound2.          Second subgoal
(*C*) optimize_for_interval.
(*D*) interval.
Qed.
```

To illustrate what VCFloat2 does and how it differs from VCFloat1, we will show the proof state at points A, B, C, D.

Point A. By now we have already reified the formula step into a deep-embedded tree step'. Unlike VCFloat1, we reified from a functional-model formula such as step, instead of from a C program; see section 5.

Point B. By this point, VCFloat's core algorithm has calculated several verification conditions. In this case there are 5 conditions. Number 3 of 5, for example, arises from the claim that $(x + h \cdot v)$ is a normal number. Here is this verification condition (cleaned up a bit); all operations are in the reals:

$$\begin{array}{l} -2 \le v \le 2 \qquad 2 \le x \le 4 \qquad |\delta_1| \le 2^{-24} \\ |\epsilon_0| \le 2^{-150} \qquad |\epsilon_1| \le 2^{-150} \qquad |\epsilon_3| \le 2^{-150} \\ \hline 2^{-126} \le |x + (\frac{1}{32}(v + (\frac{1}{32}(3-x) + \epsilon_1))(1+\delta_1) + \epsilon_3) + \epsilon_0| \end{array}$$

This is a claim that $x + h(v + (h/2)(3-x))$, after rounding, is not smaller than 2^{-126}: it is not subnormal. The line all:interval (at Point B in the proof) indicates that these 5 goals are easily dispatched by the interval tactic [13, §4.2].

Point C. By this point VCFloat has inserted deltas and epsilons using the improved analysis of VCFloat2 as described in Section 7, and now one must prove that the resulting difference formula is within the error bound:

$$\begin{array}{l} -2 \le v \le 2 \qquad 2 \le x \le 4 \qquad |\delta_1| \le 2^{-24} \qquad |\delta_2| \le 2^{-24} \\ |\epsilon_0| \le 2^{-150} \qquad |\epsilon_1| \le 2^{-150} \qquad |\epsilon_3| \le 2^{-150} \\ \hline |(x + (\frac{1}{32}((v + (\frac{1}{64}(3-x) + \epsilon_1))(1+\delta_1) + \epsilon_3) + \epsilon_0))(1+\delta_2) \\ \quad - (x + \frac{1}{32}(v + (\frac{1}{32}/2)(3-x)))| \qquad \le \frac{1}{4{,}000{,}000} \end{array}$$

In general this is a difficult formula to analyze, because of nested epsilons and deltas and the implicit subtraction of $x - x$. If we give this directly to the interval package, it will derive a very weak bound, much worse than the desired one. Here we use our new special-purpose simplifier, which (efficiently and soundly) transforms the goal at Point C into the proof goal at Point D.

Point D. $|\,\delta_2 x + \frac{1}{32}\delta_1 v + \frac{1}{32}\delta_2 v| \le \frac{1}{4{,}000{,}000} - 4.07453 \cdot 10^{-10}$ This goal is easily solved by the interval tactic.

On some "Point C" goals we also use a (new) tactic that recursively decomposes the expression for absolute floating-point error into smaller subexpressions of related terms that are easier for our simplifier (and Coq Interval package); see §10. The Gappa tool uses a similar technique.

[3]If a bound is not known in advance, VCFloat2 can calculate and produce a proof of a bound at the same time. We don't illustrate that here.

4 Floating point types, operations, and notation

VCFloat1 defined a floating point type as (basically),

```
Record type: Type :=
  TYPE {fprec: Z; femax: Z; prec_range: 1 < fprec < femax}.
```

That is, fprec is the number of bits in the mantissa, femax is the maximum exponent value. The proof fprec_range enforces (via dependent types) that the precision (number of mantissa bits) must be less than the maximum exponent.[4]

This was sufficient to describe any IEEE 754 format (including half-precision, double-precision, quad-precision, etc.). But in VCFloat2 we support exotic floating-point types that don't precisely fit the IEEE 754 format. In particular, each value of the synthetic type double-double [20] has fprec *at least (or exceeding)* 106 bits, and femax=1024. Double-double generally respects the round-off bounds for $\langle 106, 1024\rangle$ but is not exactly described by any VCFloat1 "type."

Therefore in VCFloat2 we define,

```
Record type: Type :=
  GTYPE {fprec: Z; femax: Z; prec_range: 1 < fprec < femax;
          nonstd: option (nonstdtype fprec femax prec_range)}.
Definition TYPE fprecp femax prec_range :=
        GTYPE fprecp femax fprec_lt_femax prec_range None.
```

When nonstd=None, this is a standard IEEE 754 format; when nonstd=Some(nt), then nt is a record describing an abstract data type and its interpretation.

The new definition TYPE allows backward compatibility for defining IEEE 754 formats, such as single-precision and double-precision:

```
Definition Tsingle : type := TYPE 24 128  ltac:(simpl;lia).
Definition Tdouble: type := TYPE 53 1024  ltac:(simpl;lia).
```

We define precisions Tsingle and Tdouble, but the user can easily add more. The ltac:(simpl;lia) finds a proof that $1 < 24 < 128$ or $1 < 53 < 1024$.

For the underlying semantics of floating-point numbers and operations, we rely on Coq's Flocq floating-point library [12, 13]. For example, subtraction:

```
Bminus (prec: Z) (emax: Z): prec>0 → prec<emax → mode →
        binary_float prec emax →
        binary_float prec emax → binary_float prec emax.
```

which operates on binary IEEE 754 floating-point numbers at any precision. The rounding mode may be round-to-nearest-even, round-toward-zero, round-down, etc. The last two arguments and the result are floats of the given precision.

Given any format-description (ty: type), we can define ftype(ty), the Coq type of *floating-point values* belonging to that format:

[4]In the actual implementation, the name and statement of fprec_range are slightly different.

```
Definition ftype (ty: type) : Type :=
 match nonstd ty with
 | None ⇒ binary_float (fprec ty) (femax ty)
 | Some nt ⇒ nonstd_rep nt
 end.
```

That is, if ty is a standard type, then the Coq type of its values is simply Flocq's binary_float type (of the appropriate precision). But if it's a nonstandard type, then we use the user-supplied representation type from the nt package.

When t is a standard type (nonstd(t)=None), we can use dependent types to convert between binary_float (fprec t) (femax t) and ftype(t):

```
Definition float_of_ftype {t:type} {STD: is_standard t} :
                    ftype t → binary_float (fprec t) (femax t).
Definition ftype_of_float {t:type} {STD: is_standard t} :
                    binary_float (fprec t) (femax t) → ftype t.
```

Now, for any Flocq-standard arithmetic operator such as Bplus, Bminus, Bmult, etc. that is parametrized by prec and emax, we can define

```
Definition BINOP op ty `{STD: is_standard ty} :
                              ftype ty → ftype ty → ftype ty :=
   ftype_of_float (op _ _ . . . (float_of_ftype x) (float_of_ftype y)).
   (* . . . indicates some parts of this definition are elided *)
Definition BPLUS := BINOP Bplus.
Definition BMINUS := BINOP Bminus.
```

Therefore, @BPLUS Tsingle has type
ftype Tsingle → ftype Tsingle → ftype Tsingle;
it is the single-precision floating-point add operator.

Many users would rather write x+y than BPLUS x y, so in VCFloat2 we provide Coq *notations*:

```
Notation "x + y" := (@BPLUS Tsingle x y)(level...): float32_scope.
Notation "x + y" := (@BPLUS Tdouble x y)(level...): float64_scope.
```

so for example ((1/2) * (h * h))%F32 is an expression using @BMULT Tsingle and with the constants 1 and 2 parsed as single-precision floating-point numbers.

Notation parser/pretty-printer for literals. Coq has 64-bit floats built-in, with notation parsers and printers for the usual notation (e.g., 1.36e+7). But we want to parse and pretty-print floats in any precision, and not into built-in floats but into Flocq's deep-embedded *description* of floats. So we implemented an entire scientific-notation parser and pretty-printer in Coq, and plugged it in using Coq's customizable Number Notation feature.[5]

[5]https://coq.inria.fr/doc/v8.17/refman/user-extensions/syntax-extensions.html#number-notations. This feature, new in Coq 8.14 (released 2021), is rather intricate to use but (with the help of a Coq wizard) we were able to instantiate it for any specific set of exponent and mantissa sizes, with about 50 lines of copy-pasted (and edited) Coq code needed per instantiation, and 390 lines of Coq programming for the generic semantics of accurate floating-point parsing and printing.

It is well known that printing floating-point numbers accurately and concisely is nontrivial [14]. There are four kinds of solutions:

1. Incorrect (in some cases, print a number that does not read back as the same number).
2. Correct but unrounded, i.e. print 0.15 as 1.49999996e-1 which reads back correctly but does not look nice.
3. Correct and concise by validation, i.e., print 0.15 as 0.15 or 1.5e-1 by trying twice (rounding down, rounding up), and then checking in which case the rounding was done correctly. This is the method we use.
4. Correct and concise by construction, i.e., sophisticated algorithms that get it right without needing to validate.

In programming languages without arbitrary-precision integers, all of this is more difficult, but in Coq we have the Z type that simplifies some issues.

5 Reification

To represent *in a logic* a function analyzing logical formulas of type τ, one cannot write a function with type $\tau \rightarrow$ Prop; one must operate on *syntactic representations* of formulas, such as our expr type. One can then define in the logic a reflect function of type expr $\rightarrow \tau$. The opposite process, *reification*, cannot be done within the logic. But we can (and do) program a reify function from τ to expr in the tactic language of the Coq proof assistant. One cannot prove reify correct, but we obtain a per-instance guarantee for each $f : \tau$ by checking that reflect(reify(f)) = f. This is *proof by reflection* [8, Ch. 16].

Reification is not a new concept, nor is the use of a tactic-based program to implement it. Where we innovate, compared to previous reifiers and compared to VCFloat1, is in the handling of *annotations* that seem to make no semantic difference—when reflected in the standard way—but become embodied in the reified term (abstract-syntax tree) so as to guide the proof of a theorem.

VCFloat1's inner workings are explained in Sections 3 and 4 of Ramananandro *et al.* [40]. First the term is reified into syntax trees. Our Listing 1 is similar to Ramananandro *et al.*'s [40] Figure 1, except that: The InvShift form of rounded_unop is new in VCFloat2; see §7. The Func form of expr is new; see §12. The notion of collection and the predicate is_standard are new; see §13.

VCFloat1 did not reify from Coq formulas; instead it translated C statements into expr terms by first parsing the C using CompCert's front end, then translating CompCert ASTs [30] into exprs. In VCFloat2 we reify from Coq formulas, not directly from C programs, even though we too are sometimes interested in proving the correctness of C programs. We reify from a *functional model* (such as the step function shown earlier), for several reasons:

```
Inductive rounded_binop: Type := PLUS | MINUS | MULT | DIV.
Inductive rounding_knowledge: Type := Normal | Denormal.
Inductive binop: Type :=
| Rounded2 (op: rounded_binop)
            (knowl: option rounding_knowledge)
| SterbenzMinus
| PlusZero (minus: bool) (zero_left: bool).

Inductive rounded_unop: Type :=
              SQRT | InvShift (pow: positive) (ltr: bool).
Inductive exact_unop: Type := Abs|Opp|Shift(pow:N)(ltr: bool).

Inductive unop: Type :=
| Rounded1 (op: rounded_unop) (knowl:
option rounding_knowledge)
| Exact1 (o: exact_unop)
| CastTo (ty: type) (knowl: option rounding_knowledge).

Inductive expr `{coll: collection} (ty: type) : Type :=
| Const (STD:is_standard ty) (f: binary_float(fprec ty)(femax ty))
| Var (IN: incollection ty) (i: V)
| Binop (STD: is_standard ty) (b: binop) (e1 e2: expr ty)
| Unop (STD: is_standard ty) (u: unop) (e1: expr ty)
| Cast (fromty: type) (STDto: is_standard ty)
         (STDfrom: is_standard fromty)
         (knowl: option rounding_knowledge) (e1: expr fromty)
| Func (f: floatfunc_package ty) (args: klist expr (ff_args f)).
```

Listing 1. Syntax of reified expressions.

- The functional model is an important artifact in its own right. It will be the subject of significant analysis, not only for floating-point round-off but for the function it calculates on the real numbers. We don't *only* want to prove that the C program accurately approximates some real-valued discrete algorithm, we want to prove that the real-valued algorithm accurately approximates the high-level goal, some real-valued function or relation. For that, we want a stable, cleanly written, human-readable functional model, not something automatically reified from a C program.
- The user of VCFloat might not be programming in C.
- Our functional modeling language (and VCFloat) can work at any floating-point precision, but CompCert C only defines 32-bit single precision and 64-bit double precision.

Our reifier is written in Coq's tactic language. Except for its treatment of annotations, it is fairly conventional. A few clauses are illustrated in Listing 2.

These four clauses reify differently annotated subtractions. Since Norm, Denorm, and Sterbenz are all identity functions, a program in Coq logic's core calculus could not distinguish them. But the tactic language can. In the Binop tree-node that it builds, different "rounding knowledge" is encoded into the syntax tree.

```
Ltac reify_float_expr E :=
 match E with
 | BMINUS _ ?a ?b ⇒
     let a' := reify_float_expr a in let b' := reify_float_expr b
      in constr:(Binop (Rounded2 MINUS None) a' b')
 | Norm (BMINUS _ ?a ?b) ⇒
     let a' := reify_float_expr a in let b' := reify_float_expr b
      in constr:(Binop (Rounded2 MINUS (Some Normal)) a' b')
 | Denorm (BMINUS _ ?a ?b) ⇒
     let a' := reify_float_expr a in let b' := reify_float_expr b
      in constr:(Binop (Rounded2 MINUS (Some Denormal)) a' b')
 | Sterbenz (BMINUS _ ?a ?b) ⇒
     let a' := reify_float_expr a in let b' := reify_float_expr b
      in constr:(Binop SterbenzMinus a' b')
 | . . .
```

Listing 2. Select clauses of the reifier.

6 The core of VCFloat

VCFloat's core algorithm is called rndval_with_cond: "compute rounded value with verification conditions." (In their paper [40, §4] it's called R.)

rndval_with_cond: expr ⟶ rexpr ∗ shiftmap ∗ list (environ ⟶ Prop).

In VCFloat2 we repackage it into a more user-friendly form, wrapping it with appropriate corollaries and adding automation tactics to help discharge the verification conditions. Suppose the user's formula is

$$(x + \frac{1}{32}v + \frac{1}{2}\frac{1}{32}\frac{1}{32}(3 - x))$$

which we reify into an expression e : expr (in the datatype of reified expressions, Listing 1). Then rndval_with_cond(e) produces results (r, m, vcs):

r : rexpr is a (reified) expression containing epsilons and deltas indexed by natural numbers, such as appears in the left-hand-side below the line at Point C (although there it appears in its reflected, not reified, form).

m : shiftmap is a map from those natural numbers to bounds-descriptors, sufficient to describe the bounds for δ_i and ϵ_j above the line at Point C.

vcs : list(environ ⟶ Prop) is a list of verification conditions, such as the one that appears (reflected, below the line) at Point B.

VCFloat's soundness theorem, rndval_with_cond_correct, is a machine-checked proof in Coq. It is presented as Theorem 3 by Ramananandro *et al.* [40]. Basically, it says that

- for any valmap ρ mapping reified variables (such as _x and _v in our example) to floating-point numbers,
- if each of the verification conditions vcs holds in environment ρ,
- **then** there exists an error-map σ from $\delta\epsilon$ indexes (natural numbers) to $\mathbb{R}$,
- such that every δ and ϵ (interpreted in σ) respects the bound in m,
- and the floating-point evaluation (in ρ) of e is finite (not an infinity or NaN),
- and the real-number interpretation of r (using ρ and σ) is exactly equal to the floating-point evaluation of e (in ρ).

VCFloat2's prove_roundoff_bound tactic (at Point A) extends this soundness theorem to handle user-supplied functions and nonstandard float types. VCFloat2's automation then applies the theorem as part of an end-to-end machine-checked proof in Coq about the floating-point round-off error of the given formula.

7 Optimizations and inverse shifts

VCFloat1 included some theorems about transformations on (reified) terms that would improve the analysis (for better error bounds). In VCFloat2 we have integrated these transformations so that they're automatically applied; to do so, we proved the necessary soundness corollaries. This is built in to the prove_roundoff_bound tactic used at Point A.

For example, multiplication by a power of 2 is exact in floating-point arithmetic, if the result stays finite. VCFloat's reified trees represent $64.0 \cdot x$ as Binop (Rounded2 MULT) 64 x, and represent $2^6 \cdot x$ as Unop (Exact1 (Shift 6)) x. Both of these "reflect" back to the same floating-point formula, but they are treated differently in the analysis: MULT introduces δ and ϵ, but Shift does not. We also use other such optimizing transformations such as constant-folding. The purpose is to optimize the analysis, not optimize the program that runs. The choice of what computation to actually run is specified by the user, in writing the functional model.

InvShift. We also implemented a new optimizing transformation: recognize division by powers of 2 (or multiplication by powers of 1/2). When $x \cdot 2^{-k}$ is a normal number, then $\mathrm{R}(x \cdot 2^{-k}) = \mathrm{R}(x) \cdot 2^{-k}$ exactly. When $x \cdot 2^{-k}$ is subnormal, then $\mathrm{R}(x \cdot 2^{-k}) = \mathrm{R}(x) \cdot 2^{-k} + \epsilon$, for $|\epsilon| \leq 2^{-E}$. We exploit this in VCFloat2's reified tree language with the InvShift operator. For example, our optimizer replaces Binop (Rounded2 DIV) x 64 with Unop (Rounded1 (InvShift 6)) x so that rndval_with_cond introduces the appropriate ϵ with its bound.[6]

[6]At present our specification of the InvShift optimization replaces, for example, $a/8$ with $a \times 2^{-3}$. These compute the same *except* that if a is a Not-a-Number (NaN) then the NaN-payloads of the two results may differ. We soundly account for this, in our proof, with an equivalence relation. Unfortunately, *that* clashes with a different feature of IEEE floating point: the behavior of fused multiply-add when one argument is negative zero. We have identified a solution to this problem, which is to model InvShift differently: replace $a/8$ with $a/2^3$. This will simplify many things, since we won't need the equivalence relation, and will permit the specification of fused multiply-add. We leave this for near-future work.

8 An efficient simplifier for interval goals

The Interval package [13, §4.2] is a procedure in Coq for proving goals of the form,

$$\frac{l_1 \le x_1 \le h_1 \qquad l_2 \le x_2 \le h_2 \quad \ldots \quad l_n \le x_n \le h_n}{l_0 \le E \le h_0}$$

where all the l_i and h_i are constants, the x_i are real-valued variables, and E is a real-valued expression over the variables. Any of the inequalities may be strict ($<$) rather than non-strict ($\le$), some of the inequalities may be missing, there may be several redundant constraints over any given x_i, and any of the inequalities may be expressed as $|x_i| \le h_i$. The l_0 and h_0 may be left unspecified, in which case Interval reports the best bounds that it can prove.

Interval uses floating-point interval arithmetic, being careful with floating-point rounding modes (round down on one side, up on the other). But that alone would provide very weak bounds, so Interval also uses higher-precision (synthetic) floating point, interval bisection, Taylor expansions, and automatic differentiation.

At Point C, just after prove_roundoff_bound2, the proof goal is in the form accepted by the Interval package. Unfortunately, Interval doesn't do a very good job on that goal. Multivariate Taylor expansion would work quite well [42], but Interval uses only univariate Taylor series.

The Interval mode that can work on our problem is (repeated) bisection of the interval. But even then, the nested expressions with deltas and epsilons are obstacles to good approximations. In particular, at Point C in Section 3 there is the formula $(x+\ldots)(1+\delta_2)-(x+\ldots)$, and subtracting $x-x$ using interval arithmetic leads to a severe over-estimation.

We found that it helps to use Coq's field_simplify tactic before calling Interval; this would turn the (Point C) goal $l_0 \le E \le h_0$ into,

$$\begin{aligned}|\,(&-x\delta_1\delta_2 - x\delta_1 + 2047x\delta_2 + 64v\delta_1\delta_2 + 64v\delta_1 + 64v\delta_2\\ &+64\epsilon_1\delta_1\delta_2 + 64\epsilon_1\delta_1 + 64\epsilon_1\delta_2 + 64\epsilon_1 + 3\delta_1\delta_2 + 3\delta_1\\ &+64\epsilon_3\delta_2 + 64\epsilon_3 + 2048\epsilon_0\delta_2 + 2048\epsilon_0 + 3\delta_2)/2048| \ \le\ \tfrac{1}{4{,}000{,}000}\end{aligned}$$

The two terms x and $-x$ have been *symbolically* canceled, which reduces the dependency effect in a subsequent interval analysis. On this new goal, the Interval tactic computes an excellent bound.

Repeatedly applying the distributive law to this multinomial has caused an exponential blow-up in the number of terms. For this small expression, "exponential" means only 17 terms, but if we apply VCFloat1 to more substantial examples, Coq runs out of memory.

The dependency effect. In interval arithmetic, expressions containing multiple occurrences of the same variable suffer from the *dependency effect*: rather than taking on a single value, each occurrence of the same variable represents a range of values, and the basic arithmetic operations assume the ranges of operands are independent [37, §1.3.5]. When interval arithmetic is used naively to bound the round-off error in floating-point computations, the dependency effect leads to a substantial over-estimation of the error. The simple example of $|x-x|$ suffices as a demonstration of the effect: if x lies in the interval $[0,2]$, then evaluating the expression in interval arithmetic without symbolic cancellation yields a worst-case error bound of 2.

A fast ring simplifier. We implemented a high-performance special-purpose simplifier, to clean up queries before asking Interval to solve them. It it creates *much* smaller formulas than does field_simplify. It works well for formulas that can (mostly) be described as multinomials. We expand the multinomial into sum-of-products form, then soundly and efficiently cancel terms while reducing the exponential blow-up in the number of terms.

A real-life numerical analyst might perhaps discard the higher-order terms, those in which more than one δ or ϵ are multiplied together. Another real-life alternative is to ignore the issue of underflow (denormalized numbers), in which case all the ϵ are discarded. Both of those methods work well most of the time. But neither one is *sound;* and we want proofs of our bounds!

Therefore, we implemented (and proved correct in Coq) an algorithm to efficiently and soundly simplify Interval goals:

Step 1: Apply *limited ring simplification*: the distributive law, and multiplication by 1 and by 0, division by 1, multiplication of constants together, limited simplification of rational constants.
Step 2: Discard *and bound the total of* insignificant terms.
Step 3: Cancel terms using an efficent balanced-binary-tree data structure.

The entire algorithm is implemented in Coq logic's functional programming language, which Coq can compile to byte-code or machine-code.

Reification. A program in Coq's logic must be applied to a *reified* term—an abstract-syntax tree—not to a "native" proof goal. For this component, we chose to use the reified-tree syntax from the Interval package, rather than VCFloat's own. This is because (1) our interval-goal simplifier should be usable by *any* user of the Interval package, not only in connection with VCFloat; and (2) we have no need of the "rounding knowledge" of VCFloat's tree syntax.

The distributive law. Step 1 of the algorithm doesn't need much explanation: it works in one pass over the tree with a recursive function.

Discarding negligible terms. Step 2, soundly discarding insignificant terms, works as follows. One might think, "let's discard any higher-order term, i.e., containing the product of two or more δ and ϵ." But some of those terms might be multiplied by very large coefficients or user-variables; and on the other hand, some terms containing only a single δ

or ϵ might be multiplied by tiny numbers and therefore be insignificant.

We will take advantage of the fact that *bounds are known for every variable*, both the original variables (x, v) and the δ and ϵ variables. So in a sum-of-products expression (resulting from limited ring simplification), we can bound every term. (Terms containing functions that we cannot bound in closed form, we handle as described below.)

The user supplies a *cutoff* such as 2^{-30} or 2^{-60} or whatever is appropriate. We preprocess all of the (already reified) bounds hypotheses (for variables $x, v, \delta_1, \epsilon_2$, etc.) into a single absolute-value bound for each variable: $|x_i| \le h_i$.

Then for each term $x_i x_j x_k$ we can look up the (reified) bound hypotheses to find a bound $h_i h_j h_k$ on the absolute value of the term. To "delete" $x_i x_j x_k$ we replace it by the constant $h_i h_j h_k$. We accumulate all those constants to produce the transformed goal,

$$\begin{aligned}&|x + x\delta_2 + \tfrac{1}{32}v + \tfrac{1}{32}v\delta_1 + \tfrac{3}{2048} - \tfrac{1}{2048}x + \tfrac{1}{32}v\delta_2\\ &- (x + \tfrac{1}{32}v + \tfrac{3}{2048} - \tfrac{1}{2048}x)| \ \le\ \tfrac{1}{4{,}000{,}000} - 4.0745386 \cdot 10^{-10}.\end{aligned}$$

Here, the term $4.0745386 \cdot 10^{-10}$ is the sum of bounds of the deleted terms. This goal implies the original goal, from Point C in Section 3.

The algorithm for deleting insignificant terms might encounter some terms whose bounds it cannot analyze because the terms are not simply products of variables and constants. Such *"residual"* terms it leaves unchanged.

Gathering similar terms and cancelling subtractions. At this point some terms could cancel (by subtraction). Step 3 cancels terms, efficiently, in the already-reified trees. We have a tree of additions of terms. Each term is *either* a product of constants and variables, *or* contains other operators; in the latter case we leave that term alone (as a *residual term*) and don't attempt to cancel it. An example of a (potentially) cancelable term is, $c_1 x_1 \delta_1 \epsilon_2 x_1 c_2 x_2$, where c_1, c_2 are rational constants, and $x_1, x_2, \delta_1, \epsilon_2$ are variables.

In our reified tree syntax, all variables are represented by natural numbers. So we can represent any product of (nonnegative integer) powers of these variables $x_0^{k_0} x_1^{k_1} x_2^{k_2}$ by a list of natural numbers $[k_0, k_1, k_2]$. The product of all the constants can be represented as a canonical-form rational number. For efficiency, we factor out all the powers of 2 from the numerator and denominator into a separate factor 2^e, where e may be positive, negative, or zero. That is, the canonical form of a (nonresidual) term is, $\overrightarrow{k} \cdot (\pm n)/d \cdot 2^e$, where where $\overrightarrow{k}$ is a list of natural numbers representing the polynomial $x_0^{k_0} x_1^{k_1} x_2^{k_2} \ldots$, n is an odd integer (or zero), d is an odd positive number, $\gcd(n, d) = 1$, and e is an integer.

We maintain a balanced binary search tree indexed by keys $\overrightarrow{k}$. At each key $\overrightarrow{k}$ we have a list of coefficients, each of the form $(\pm n)/d \cdot 2^e$. In walking the expression-tree, whenever we find $\overrightarrow{k} \cdot (\pm n)/d \cdot 2^e$ we look up k, and traverse the list: if we find the *negation* of $(\pm n)/d \cdot 2^e$ we delete it from the list, otherwise we cons $(\pm n)/d \cdot 2^e$ to the front of the list; then reinsert at key k. Meanwhile, we replace that term in the expression-tree with 0. What remains is:

- an expression-tree with residual terms that could not be represented in canonical form, and zeros where terms have been removed from the tree and added to the key-value map;
- a key-value map: for each key $\overrightarrow{k}$ a list of coefficients each of the form $\frac{n}{d} \cdot 2^e$.

After the key-value map is built, for each k mapped to a list of coefficients, we add all the coefficients together, as follows: we normalize all the elements to have the same exponent e (so that it is no longer true that every n and d is odd), then add all the rational numbers, to collapse the list into a single coefficient.

We convert each key-value pair back into an expression $x_0^{k_0} x_1^{k_1} x_2^{k_2} \ldots x_n^{k_n} (\pm n)/d \cdot 2^e$, and build a final expression tree with their sum, plus the residual terms previously accumulated. The result is shown at Point D.

Efficiency of the algorithm. Step 1 of the algorithm (distributive law) takes exponential time. Step 2 takes linear time and space (in the exponentially sized term produced by step 1). Step 3 takes $N \log N$ time and linear space (and tends to reduce the term back to small size).[7]

We tested this algorithm on a large expression that resulted from calculating position-change and momentum-change of a harmonic oscillator and then taking the sum of squares of position and momentum.

- The original expression-tree (Point C) had 242 nodes (constants, variables, operators).
- After step 1 (distributive law) there were 612,284 nodes, or 31,759 multinomial terms.
- After step 2 (delete insignificant terms) there were 1456 nodes, 244 terms.
- After step 3 (cancel) there were 219 nodes, 24 terms.

On this large expression with cutoff 2^{-30}, the entire algorithm runs in Coq in 1.8 seconds.[8] One might think, "simplifying 242 nodes into 219 nodes is not much of an accomplishment." But it is quite significant: the final expression has canceled the $x - x$ and $v - v$ that caused Interval to give horribly loose bounds.

[7]Since variables (x, δ, ϵ) are represented by natural numbers in unary, all these numbers must be multiplied by the number of variables. Using a binary representation would reduce this to a logarithmic factor.

[8]1.177 to 1.189 seconds using `vm_compute` on a MacBook Pro M2 laptop in Coq 8.17.1. It would probably be faster using `native_compute`. With cutoff 2^{-53} it takes 1.3 seconds. Smaller cutoffs are unlikely to be useful, but even so: cutoff 2^{-100} takes 3.4 seconds; 2^{-120} takes 7.4s; 2^{-140} takes 13.8s, 2^{-200} takes 36.4s. Coq's `ring_simplify` takes 3.365 seconds but cannot treat the divide operations; `field_simplify` takes 6.956 seconds; but in either case, Coq's stack overflows pretty-printing the result because there are so many terms.

A more efficient algorithm. It should be possible to discard negligible terms *interleaved* with the distributive law, so that the exponential blow-up in step 1 never occurs. To do so, one would first walk over the tree bounding the absolute value of every subexpression (using the bounds hypotheses). A second pass would walk the tree, such that in the context $A * B$, while processing B one would adjust the cutoff by the bound for A. We may implement this algorithm in the future. For now, the algorithm we have seems fast enough.

Floating-point interval arithmetic. In this section we have described analyses on real-valued formulas that contain integer and floating-point constants. Since one cannot efficiently compute on the real numbers, we perform our analyses in floating-point. Because the analyses must be sound even in the presence of round-off error, we compute in floating-point interval arithmetic, as provided by the Coq Interval package.

9 Soundness theorem for simplification

The algorithm described in the last section is proved correct in Coq, so applications of it are correct by reflection.

We use the Interval package's reify function to turn the user's functional model into a tree-term e of type expr. As usual in Coq, reify is written as a tactic program in the **Ltac** language, and we validate each reification by reflecting (see §5¶1).

The user chooses a (small, floating-point) cutoff value, and VCFloat2's tactic applies the following function:

```
Definition simplify_and_prune hyps e cutoff :=
(* Step 1 *) let e1 := ring_simp e in
(* Step 2 *) let '(e2,slop) := prune (map b_hyps hyps) e1 cutoff in
(* Step 3 *) let e3 := cancel_terms e2 in   (e3, slop).
```

The function simplify_and_prune embodies the three-part algorithm described in Section 8. Before stating the correctness theorem for simplify_and_prune, we must define the notion of equivalently evaluating expressions:

```
Inductive expr := ... (* from the Interval package *)
Definition environment := list ℝ. (* map variables,
                              represented as ℕ, to values *)
Definition eval : expr ⟶ environment ⟶ ℝ :=
              ... (* from the Interval package *)
Definition expr_equiv (a b: expr) : Prop :=
        ∀ env, eval a env = eval b env.
Infix "==" := expr_equiv (at level 70, no associativity).
```

The correctness theorems for ring_simp and cancel_terms is that they exactly preserve evaluation, in any environment.

```
Lemma ring_simp_correct: ∀ e, ring_simp e == e.
Lemma cancel_terms_correct: ∀ e, cancel_terms e == e.
```

The specification of prune is a bit more complicated, and therefore so is the specification of simplify_and_prune, shown in Listing 3. It says, suppose you wish to prove that $|e| \leq r$ with bounds-hypotheses hyps and variables vars that satisfy hyps. Suppose also that simplify_and_prune gives you a simplified expression e_1, and that the total of all deleted terms is bounded by s. Then it suffices to prove $|e_1| \leq r - s$.

```
Lemma simplify_and_prune_correct:
 ∀ hyps e cutoff e₁ s,
   simplify_and_prune hyps e cutoff = (e₁, s) ⟶
   F.real s = true ⟶
   ∀ (vars: list ℝ) (r : ℝ),
     length hyps = length vars ⟶
     eval_hyps hyps vars (Rabs (eval e₁ vars) ≤ r − R(s)) ⟶
       eval_hyps hyps vars (Rabs (eval e vars) ≤ r).
```

Listing 3. The specification of simplify_and_prune.

For example, if there are $\leq 10^{10}$ terms to delete and a cutoff of 10^{-8} is specified, then it is *inconceivable* that s will overflow, since 10^{10-8} is representable in double-precision. But just in case, hypothesis F.real s = true tests for overflow.

10 Decomposition tactic

VCFloat2 produces an optimized expression for the absolute error $|\tilde{y} - y|$ between the value $\tilde{y}$ resulting from the floating-point evaluation of an expression and the value y resulting from the evaluation of the same expression over the real numbers. For polynomial expressions, the simplifier described in Section 8 transforms the top-level interval goal $|\tilde{y} - y| \leq$ *bound* into a goal that the Interval tactic is more likely to prove a tight bound on. More generally, for rational expressions, the simplifier can be applied effectively to smaller subgoals that are generated by *decomposing* the top-level interval goal using some heuristics that reduce the dependency effect in the interval analysis.

The decomposition is packaged into a tactic in VCFloat2 called error_rewrites. It works by recursively approximating the left-hand side of interval goals for the absolute error. First, distributive and associative laws are applied at the top-level to the ϵ and δ error variables inserted by VCFloat as described in Section 2; for example, goals with left-hand sides of the form $|(\tilde{u}/\tilde{v})(1+\delta) + \epsilon - u/v|$ are rewritten as $|(\tilde{u}(1+\delta))/\tilde{v} - u/v + \epsilon|$. Then, the resulting left-hand side is recursively approximated using the triangle inequality (for ϵ terms) and the following inequalities, the first of which is particularly important for rational expressions.

$$\left|\frac{\tilde{u}}{\tilde{v}} - \frac{u}{v}\right| \leq \left(|\tilde{u} - u| + |\tilde{v} - v| \cdot \left|\frac{1}{v}\right| \cdot |u|\right) \cdot \left|\left(1 + \frac{\tilde{v} - v}{v}\right)^{-1}\right| \cdot \left|\frac{1}{v}\right|$$

$$|\tilde{u} \cdot \tilde{v} - u \cdot v| \leq |\tilde{u} - u| \cdot |v| + |\tilde{v} - v| \cdot |u| + |\tilde{u} - u| \cdot |\tilde{v} - v|$$

$$|(\tilde{u} + \tilde{v}) - (u + v)| \leq |\tilde{u} - u| + |\tilde{v} - v|$$

$$|(\tilde{u} - \tilde{v}) - (u - v)| \leq |\tilde{u} - u| + |\tilde{v} - v|.$$

Each of the above inequalities corresponds to a transformation used in Gappa, as described by Boldo and Melquiond [13, §4.3.2.2]. The decomposition tactic in VCFloat2 treats each subexpression in the right-hand side of the above inequalities as a separate interval goal, and each of these smaller interval goals generally contains fewer terms that suffer from the dependency effect than the original.

11 Proving C programs

The Verified Software Toolchain is a Coq library for proving the correctness of C programs with respect to specifications written in Coq's logic. Its specification language is higher-order separation logic with propositions that can use all of Coq's logic. Therefore its specification language and proof system is much more expressive than such systems as Dafny [29], Verifast [24], Frama-C [28]. Furthermore, because it is embedded in Coq, one can compose, *entirely within Coq*, a VST proof that a C program correctly implements a functional model, with a Coq proof that a functional model correctly implements some high-level specification [26].

VST includes a full treatment of C language floating point, using the Flocq model of the IEEE 754 standard. In VST one can (fairly easily) prove that a C program implements a floating-point functional model [4, 26, 27, 43]. But VST provides no help in reasoning *about* the functional model. That is what VCFloat2 can do.

VST describes float operations using CompCert's thin layer of definitions over Flocq's description of IEEE 754 single-precision and double-precision. VCFloat2's functional modeling language uses a different thin layer over the same Flocq types. VCFloat2's compatibility library FPCompCert bridges the small gap.

Fast-math, or not? Some compilers do "fast math" optimizations that change the semantics of floating-point operations; for example, combining $a \times b + c$ into a *fused multiply-add* (fma) which omits an internal rounding step. CompCert does not do any transformations that alter floating-point semantics. The CakeML verified ML compiler does [6], and its authors "argue that any compiler correctness result for fast-math optimizations should appeal to a real-valued semantics rather than the rigid IEEE 754 floating-point numbers." That argument is sound, but *should* a compiler perform such transformations? We suggest *no*, because it greatly complicates the *specification* to be proved about the low-level program. Our approach, as explained in this paper, is to prove (with VST or similar tools) that the low-level program *exactly* implements a floating-point functional model. That proof treats floating-point operations as uninterpreted functions, and does not even mention the real numbers. When we want fused multiply-add, we write it that way in the functional model, and in the C program, using VCFloat2's new mechanism that we will describe in the next section.

12 User-supplied functions

VCFloat2 allows the user to provide and use arbitrary functions (e.g., sin, cos). The user must provide proofs (or axioms) of their rounding behavior. Then VCFloat2 will compute and prove round-off error bounds for formulas containing calls to these functions (along with standard operators $+, -, \times, \div$, etc.). This is done by extending the reified expr syntax and extending the rndval_with_cond theorem to accommodate user-supplied functions.

Any function that takes 0 or more float arguments (not necessarily all of the same type) and produces a float result can be used in VCFloat2. The user provides a floatfunc_package to describe its characteristics:

```
Record floatfunc (args: list type) (result: type)
      (precond: klist bounds args)
      (realfunc: function_type (map RR args) R) :=
{ff_func: function_type (map ftype' args) (ftype' result);
 ff_rel: N;
 ff_abs: N;
 ff_acc: acc_prop args result ff_rel ff_abs
                    precond realfunc ff_func}.
```

```
Record floatfunc_package (ty: type) :=
 {ff_args: list type;
  ff_precond: klist bounds ff_args;
  ff_realfunc: function_type (map RR ff_args) R;
  ff_ff: floatfunc ff_args ty ff_precond ff_realfunc}.
```

Suppose p is floatfunc package, that is, p: floatfunc_package(ty). Then ff_args p is a list of argument types, and ty is the result type. For example, suppose ff_args p =[Tdouble,Tsingle] and ty=Tsingle, then the function f = ff_func(ff_ff p) has type f: ftype Tdouble → ftype Tsingle → ftype Tsingle, because function_type (map ftype' (ff_args p)) (ftype' ty) is convertible to ftype Tdouble → ftype Tsingle → ftype Tsingle.

The specification of this function is as follows: There is a real-valued function ff_realfunc(p): R → R → R; and ff_acc is the theorem that f approximates this function within relative error ff_rel p and absolute error ff_abs p, provided that the arguments to f are within the bounds specified by ff_precond.

Calls to such functions are reified into the Func constructor of the expr syntax (see §5).

```
Inductive klist (k : type → Type) : list type → Type :=
| Knil : klist k []
| Kcons {ty tys} : k ty → klist k tys → klist k (ty :: tys).
```

```
Inductive expr `{coll: collection} (ty: type) : Type :=
| Const (STD: is_standard ty) (f: binary_float(fprec ty)(femax ty))
| Binop (STD: is_standard ty) (b: binop) (e1 e2: expr ty)
. . .
| Func (f: floatfunc_package ty) (args: klist expr (ff_args f)).
```

Unlike VCFloat1's expr type, which was monotyped (in Coq) but each constructor labeled the intended type of its float values, VCFloat2's expr is dependently typed: expr(t) is the type of reified expressions that would evaluate to float-values of type ftype(t).

Func is *especially* dependently typed, in that klist forms a list of argument subexpressions each with a (potentially) different type, according to the list of types that is ff_args(f). The klist type constructor is for *heterogeneous lists* [16, §9.2].

From any given floatfunc_package (with its ff_acc theorem), VCFloat2 derives both the evaluation semantics and the (provable) rounding error of calls to these external functions.

VSTlib [3] is a Coq library for use in VST-verified C programs, with proofs or axiomatizations of the specifications of C libraries such as malloc/free, threads, and locks. VSTlib also provides an axiomatization of the GNU math library, including 58 standard Posix math functions such as sin, cos, fma (fused multiply-add), and so on. Each of these is specified to provide a certain level of floating-point accuracy *on each specific target architecture* as documented in the GNU C library manual [22]. Different architecture-specific implementations of the math library have been measured to have different accuracy guarantees. So if you install VST with target architecture AArch64, then you'll get a single-precision arctangent function accurate within 1.5 ulp (unit in last place), but on VST configured for x86-32 it'll be specified as accurate to 0.5 ulp. Accuracy specifications for floating-point functions are written using VCFloat2's floatfunc_package framework, as described earlier in this section.

13 Nonstandard floating-point-like types

Users may implement in software floating-point types that do not correspond exactly to any IEEE 754 format but that can be shown to respect round-off error bounds. VCFloat2 supports such user-defined types.

Consider the example of double-double [20]. A double-double number can be used to represent a floating-point number with at least 106 bits of mantissa using two double-precision floats whose 53-bit mantissas do not overlap (because their exponents differ by at least 53). One can add or multiply numbers in this data type using just a few ordinary double-precision operations, especially on machines that support fused multiply-add (fma).

Rounding double-double can be shown to be follow a model with relative error $|\delta| \leq 2^{-106}$ and absolute error $|\epsilon| \leq 2^{-1022}$. If the user can prove such a property of an abstract number type (of which double-double is just one example), then they can build a nonstdtype record in VCFloat2:

```
Record nonstdtype
      (fprec: Z) (femax: Z) (prec_range: 1 < fprec < femax) :=
 NONSTD
  { nonstd_rep: Type;
     nonstd_to_F: nonstd_rep → option (float radix2);
     . . . (* some fields omitted *) . . .
     nonstd_bounds: ∀ x: nonstd_rep,
          ( – (bpow radix2 femax – bpow radix2 (femax – fprec)) <=
            floatopt_to_real (nonstd_to_F x) <=
            bpow radix2 femax – bpow radix2 (femax – fprec) )%R }.
```

Here, nonstd_bounds is the user-supplied proof of such a rounding theorem, nonstd_to_F is the user-supplied definition of a function that maps nonstandard types to a floating-point representation, and floatopt_to_real is a function provided by VCFloat2 that maps a floating-point representation to a real number.

After building a nonstdtype record, the GTYPE constructor is used to build a new VCFloat2 type. Expressions in which some functions return values of this type and other functions take values of this type can now be reified, and VCFloat2 will automatically calculate and prove round-off error bounds.

In order to write such expressions, users of VCFloat2 must first define operations on the newly defined type. This is done by constructing a floatfunc record (whose type was presented in Section 12).

Standard vs. nonstandard types. One might wonder what is special about a "standard" type, that it cannot be just an instance of the general notion of nonstandard type. A standard type must support all the operations listed in Listing 1; nonstandard types support only whatever functions someone has supplied. The notion of an arbitrary "cast" from any IEEE precision to any other cannot be generically supported, and (for example) some nonstandard types do not support Sterbenz subtraction, or constant literals.

14 Double-doubles in VCFloat2

To demonstrate nonstandard types and the definition of a floatfunc over a nonstandard type, we use double-double. We instantiate a nonstdtype whose nonstd_rep is

{ a: ftype Tdouble * ftype Tdouble | dd_pred a}

where ddpred is a predicate describing a well-formed double-double. Based on this representation, we prove the required properties of a nonstdtype, such as nonstd_bounds.

As an example of a nonstandard operation on double-doubles constructed using a floatfunc, we use the operation DWPlusFP, shown in Listing 4, summing a double-double number and a double-precision IEEE 754 floating-point number. That is,

- we state the real-valued functional model as real-number addition;
- we define the floating-point valued functional model as the DWplusFP algorithm;
- we state its accuracy parameters (relative and absolute error corresponding to a hypothetical IEEE 754 type with 106-bit mantissa);
- and we prove the accprop, the accuracy property that relates the two models up to the stated accuracy. We adapt Muller and Rideau's Coq formalization of double-word arithmetic [36], extended to consider overflow and underflow.

We implemented the function in C, shown in Listing 5, and used VST to prove that it implements the floating-point

Table 1. Round-off error bounds for Gappa, PRECiSA, FPTaylor, and VCFloat2. The column labeled "Ratio" compares VCFloat2's error bound to the best performer (smaller is better). "Time" is shown as the sum x+y of the execution times (in seconds on a MacBook Pro M2) for Coq to (x) calculate and prove the error bound and (y) check the proof using the Qed command. "Vars" and "Ops" are the number of variables and operations in the formulas. Remarks: (a) uses our fast ring simplifier discarding negligible terms (b) uses our decomposition tactic (c) uses field_simplify (d) would benefit from let-bindings (future work) (–) no special preparation of the interval goal.

Benchmark	Gappa	FPTaylor	PRECiSA	VCFloat2	Ratio	Time	Vars	Ops	Remarks
carbonGas	2.7e-08	9.2e-09	7.2e-09	2.5e-08	3.5	10+9	1	11	a,b,c
doppler1	2.1e-13	1.6e-13	2.0e-13	4.5e-13	2.8	14+1	8	3	a,b
doppler2	4.0e-13	2.9e-13	3.8e-13	1.2e-12	4.1	12+1	8	3	a,b
doppler3	1.1e-13	8.3e-14	1.1e-13	2.0e-13	2.4	6+1	8	3	a,b
himmilbeau	1.1e-12	1.4e-12	1.0e-12	2.3e-12	2.3	2+.5	2	14	a
jetEngine	8.3e06	1.4e-11	1.6e-11	2.1e3	10^{14}	45+10	2	48	d
t_div_t1	1.0e03	5.8e-14	3.9e-15	4.4e-16	0.1	.7+.1	1	2	c
kepler0	1.3e-13	9.5e-14	1.1e-13	2.2e-13	2.3	2.5+.5	6	15	a
kepler1	5.4e-13	3.6e-13	3.9e-13	1.6e-12	4.6	6+2	4	24	a
kepler2	2.9e-12	2.0e-12	1.5e-12	6.2e-12	4.0	21+8	6	36	a
predprey	2.1e-16	1.9e-16	1.8e-16	3.1e-16	1.7	95+1	1	7	a,b,c
rigidBody1	3.0e-13	3.9e-13	3.0e-13	3.1e-13	1.0	1.5+.2	3	7	a
rigidBody2	3.7e-11	5.3e-11	3.6e-11	3.9e-11	1.1	4+1	3	14	a
verhulst	4.2e-16	3.3e-16	2.9e-16	2.3e-16	0.8	7+.3	1	4	–
turbine1	8.4e-14	2.4e-14	2.3e-14	7.9e-14	3.4	6+2	3	14	a,b,c
turbine2	1.3e-13	2.6e-14	3.1e-14	1.2e-13	4.6	5+1	3	10	a,b,c
turbine3	4.0e01	1.3e-14	1.7e-14	6.1e-14	4.7	7+2	3	14	a,b,c

```
Definition TwoSum (a b : ftype t) :=
let s := a+b in let a' := s−b in let b' := s− a' in let da := a−a' in
let db := b−b' in (a+b, da+db).

Definition Fast2Sum (a b : ftype t) :=
let s := a+b in let z := s−a in let t := b−z in (s, t).

Definition DWPlusFP (xh xl y : ftype t) :=
let (sh, sl) := TwoSum xh y in let v:= xl+sl in Fast2Sum sh v.
```

Listing 4. The DWPlusFP operation used in the construction of a nonstandard operation on double-doubles in VCFloat2.

```
void dw_plus_fp(struct dword *st,struct dword *x, double y) {
    double v; struct dword sh;
    two_sum(&sh,x→ s,y);
    v = x→ t + sh.t;
    fast_2sum(st,sh.s,v);
}
```

Listing 5. The double-word plus a floating-point number operation implemented in C.

functional model DWPlusFP. Now one can use DWPlusFP in VCFloat2 as a user-defined function operating on a user-defined type, and VCFloat2 can reason automatically about round-off error of programs that call this function.

Testing Type ***equality during reification.*** In order to reify such an expression, it's necessary to test whether the Type of a subexpression is equal the underlying representation Type of a nonstdtype (we capitalize Type to emphasize that these are Coq types, not our type data structure). But there is no decidable equality on Type. Since this test is being done by the reifier, which is implemented in the tactic language, we can get by with the ability of tactics to test exact identity (a stronger property than equality). But to do this, the reifier needs a list of candidate Types to test against. This we call a collection, and (before reifying anything) the user must "register" any nonstandard type by including it in a typeclass instance of class collection.

15 Related work and performance evaluation

Several tools perform round-off error analysis and generate machine-checkable proofs: Gappa [9] is implemented in C++ and generates Coq proof scripts; PRECiSA [34] is implemented in Haskell and C and generates proofs in PVS; FPTaylor [42] is implemented in OCaml and can generate proofs in HOL Light; Real2Float [32] is built on top of the NLCertify [31] verification system and generates proof certificates that can be checked in Coq, and Daisy [18] is implemented in Scala and generates proofs in Coq and HOL4 [7].

In comparison to these other tools, VCFloat2's formula language is shallow-embedded expressions in the proof assistant's own logic, so not only can VCFloat2 generate proofs (as Gappa, PRECiSA, and FPTaylor can), but it can operate directly on inputs "from the logic." Thus, VCFloat2 fits better into an integrated error analysis and correctness verification of a numerical program—of which floating-point round-off is only one component. We want to connect to other proofs (about program correctness, about discretization error, etc.) done in a proof assistant for a higher-order logic.

We assessed VCFloat2 on several benchmarks from FPBench [17] (see Table 1). We chose 17 benchmarks for which there are previously reported results for FPTaylor, Gappa, and PRECiSA. The column "Ratio" in Table 1 compares the error bounds produced by VCFloat2 to the best performer of PRECiSA, Gappa, and *verified* error bounds computed by FPTaylor (FPTaylor-f) [42]. VCFloat2 finds bounds within an order of magnitude of the best performer on 16 of the 17 problems. VCFloat2, like Gappa, fails to find a useful bound on *jetEngine*. Notably, the error bounds computed by VCFloat2 for the *doppler1-3* benchmarks are within the same order of magnitude of those obtained by FPTaylor, but took an average of 17 seconds each compared to an average of 37 minutes each (on a slower computer) for FPTaylor. Times for verifying the certificates generated by Gappa and PRECiSA were not reported.

On any benchmark *without* a or b in the "remarks," VCFloat1 would have gotten the same result, because (in these cases) VCFloat2 is using only VCFloat1 functionality. On the other benchmarks VCFloat1 would generally fail, unless the user did substantial ad-hoc tactical proofs equivalent to our automated tactics corresponding to remarks a or b.

VCFloat2 fails on *jetEngine* because it is a proper rational (not a multinomial) with a large number of operations (48 ops). VCFloat2's Prune tactic (§8) does multinomial pruning (where appropriate) and solves remaining subgoals using the Coq Interval package, which can do either multivariable first-order interval arithmetic or single variable Taylor models—but this formula is multivariate and not a multinomial. And the decomposition tactic provided by VCFloat2 works well on multivariate multinomials with a smaller number of operations (e.g., *turbine1-3*).

In each of the benchmarks we considered, all inputs are assumed to be exactly representable floating-point numbers. For modular verification efforts, where the round-off errors produced by one function might propagate to the inputs of another, it is particularly useful to be able to specify that some inputs are not exact; while this is possible in PRECiSA, Gappa, and FPTaylor, is not currently a feature of VCFloat2. Finally, modularity via the separate treatment of the propagation of input errors and the local introduction of round-off errors has enabled the development of scalable tools for rounding error analyses such as Hugo [1] and Satire [19]; this type of analysis has not yet been adopted in tools that produce proof certificates.

For each FPBench benchmark in Table 1, the VCFloat2 functional model is a Coq function over double-precision floats (ftype Tdouble). These functional models can therefore make use of features of Coq's core language, such as let-binders for sharing common subexpressions. But the syntax of reified terms (see Listing 1) does not support let-binders; common subexpressions are expanded upon reification and the resulting reified terms can be very large. This degrades both the efficiency and the quality of the analysis. For example, k occurrences of a subexpression that produces a single relative error term will cause VCFloat to introduce k independent δ error variables. In contrast, FPTaylor and PRECiSA maintain the dependence between round-off errors generated by common subexpressions.

Precision-tuning tools like FPTuner [15] and Precimonious [41] suggest, without proofs, which floating-point operations in a program can be done at double, single, or half-precision. The accuracy of these tuned programs could, in principle, be proved by VCFloat2. Herbie [39] discovers transformations of straight-line expressions that improve the floating-point round-off error; Becker et al. [5] used Daisy to produce sound upper bounds on the round-off error of rewrites produced by Herbie.

16 Conclusion

VCFloat2 permits automatic round-off error analysis to be done as part of a larger numerical analysis that treats algorithm correctness, discretization analysis, and low-level program correctness, *all in the same general-purpose logic* and with end-to-end composable, machine-checked proofs. For that purpose, it provides a language for writing floating-point functional models that clearly and simply relate to real-valued functional models. Below VCFloat2, VST (or other another tool) can reason about C program correctness and connect to VCFloat2's functional models. Above VCFloat2, tools for proofs in Coq about the properties of real-valued functional models are an exciting area for future research.

Future work. There are several interesting directions for future work: (1) Using multivariate Taylor models, perhaps as a form of optimize_for_interval at Point C of our process described in Section 3, would allow VCFloat2 to perform error analysis similar to FPTaylor. (2) Including an entire library of double-double operations in VCFloat2; we have only included a single operation to demonstrate the capability of our nonstandard-type mechanism. (3) Allowing input variables to come with their own error bounds (instead of being assumed exact) would give VCFloat2 modularity. (4) Allowing let-binders in expressions, or automatically handling common subexpressions, would improve overall accuracy bounds and would improve efficiency by reducing the number of error variables.

References

[1] Rosa Abbasi and Eva Darulova. 2023. Modular Optimization-Based Roundoff Error Analysis of Floating-Point Programs. In *Static Analysis*, Manuel V. Hermenegildo and José F. Morales (Eds.). Springer Nature Switzerland, 41–64. https://doi.org/10.1007/978-3-031-44245-2_4

[2] Andrew W. Appel. 2011. Verified Software Toolchain. In *ESOP'11: European Symposium on Programming*, Gilles Barthe (Ed.). LNCS, Vol. 6602. Springer, 1–17. https://doi.org/10.1007/978-3-642-19718-5_1

[3] Andrew W. Appel. 2023. VSTlib: Library Components for Verified C Programs. In *Coq Workshop 2023*, Yves Bertot and Enrico Tassi (Eds.). 4 pages. https://coq-workshop.gitlab.io/2023/abstracts/coq2023_vstlib.pdf

[4] Andrew W. Appel and Yves Bertot. 2020. C-language floating-point proofs layered with VST and Flocq. *Journal of Formalized Reasoning* 13, 1 (Dec. 2020), 1–16. https://doi.org/10.6092/issn.1972-5787/11442

[5] Heiko Becker, Pavel Panchekha, Eva Darulova, and Zachary Tatlock. 2018. Combining Tools for Optimization and Analysis of Floating-Point Computations. In *World Congress on Formal Methods (LNCS, Vol. 10951)*. Springer, 355–363. https://doi.org/10.1007/978-3-319-95582-7_21

[6] Heiko Becker, Robert Rabe, Eva Darulova, Magnus O Myreen, Zachary Tatlock, Ramana Kumar, Yong Kiam Tan, and Anthony Fox. 2022. Verified Compilation and Optimization of Floating-Point Programs in CakeML. In *36th European Conference on Object-Oriented Programming (ECOOP 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 1:1–1:28. https://doi.org/10.4230/LIPIcs.ECOOP.2022.1

[7] Heiko Becker, Nikita Zyuzin, Raphaël Monat, Eva Darulova, Magnus O. Myreen, and Anthony Fox. 2018. A Verified Certificate Checker for Finite-Precision Error Bounds in Coq and HOL4. In *2018 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 1–10. https://doi.org/10.23919/FMCAD.2018.8603019

[8] Yves Bertot and Pierre Casteran. 2004. *Interactive Theorem Proving and Program Development*. Springer. https://doi.org/10.1007/978-3-662-07964-5

[9] Sylvie Boldo, Jean-Christophe Filliâtre, and Guillaume Melquiond. 2009. Combining Coq and Gappa for certifying floating-point programs. In *International Conference on Intelligent Computer Mathematics*. Springer, 59–74. https://doi.org/10.1007/978-3-642-02614-0_10

[10] Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. 2013. A formally-verified C compiler supporting floating-point arithmetic. In *2013 IEEE 21st Symposium on Computer Arithmetic*. IEEE, 107–115. https://doi.org/10.1109/ARITH.2013.30

[11] Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. 2015. Coquelicot: A User-Friendly Library of Real Analysis for Coq. *Mathematics in Computer Science* 9 (2015), 41–62. https://doi.org/10.1007/s11786-014-0181-1

[12] Sylvie Boldo and Guillaume Melquiond. 2011. Flocq: A unified library for proving floating-point algorithms in Coq. In *2011 IEEE 20th Symposium on Computer Arithmetic*. IEEE, 243–252. https://doi.org/10.1109/ARITH.2011.40

[13] Sylvie Boldo and Guillaume Melquiond. 2017. *Computer Arithmetic and Formal Proofs: Verifying Floating-point Algorithms with the Coq System*. Elsevier. https://doi.org/10.1016/C2015-0-01301-6

[14] Robert G. Burger and R. Kent Dybvig. 1996. Printing Floating-Point Numbers Quickly and Accurately. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation (PLDI '96)*. Association for Computing Machinery, 108–116. https://doi.org/10.1145/231379.231397

[15] Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. 2017. Rigorous Floating-Point Mixed-Precision Tuning. In *POPL'17: 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. Association for Computing Machinery, New York, NY, USA, 300–315. https://doi.org/10.1145/3009837.3009846

[16] Adam Chlipala. 2013. *Certified Programming with Dependent Types*. MIT Press.

[17] Nasrine Damouche, Matthieu Martel, Pavel Panchekha, Jason Qiu, Alex Sanchez-Stern, and Zachary Tatlock. 2016. Toward a standard benchmark format and suite for floating-point analysis. In *Numerical Software Verification (NSV'16) (LNCS, Vol. 10152)*. Springer, 63–77. https://doi.org/10.1007/978-3-319-54292-8_6

[18] Eva Darulova, Anastasiia Izycheva, Fariha Nasir, Fabian Ritter, Heiko Becker, and Robert Bastian. 2018. Daisy – Framework for Analysis and Optimization of Numerical Programs (Tool Paper). In *Tools and Algorithms for the Construction and Analysis of Systems*, Dirk Beyer and Marieke Huisman (Eds.). Springer, 270–287. https://doi.org/10.1007/978-3-319-89960-2_15

[19] Arnab Das, Ian Briggs, Ganesh Gopalakrishnan, Sriram Krishnamoorthy, and Pavel Panchekha. 2020. Scalable yet Rigorous Floating-Point Error Analysis. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–14. https://doi.org/10.1109/SC41405.2020.00055

[20] T. J. Dekker. 1971. A Floating-Point Technique for Extending the Available Precision. *Numerical Mathematics* 18 (1971), 224–242. https://doi.org/10.1007/BF01397083

[21] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. 2019. Simple High-Level Code For Cryptographic Arithmetic: With Proofs, Without Compromises. In *2019 IEEE Symposium on Security and Privacy*. IEEE, 1202–1219. https://doi.org/10.1109/SP.2019.00005

[22] gnu 2023. GNU C Library, §19.7: Known Maximum Errors in Math Functions. (2023). //www.gnu.org/software/libc/manual/html_node/Errors-in-Math-Functions.html.

[23] Jason Gross, Andres Erbsen, Jade Philipoom, Miraya Poddar-Agrawal, and Adam Chlipala. 2022. Accelerating Verified-Compiler Development with a Verified Rewriting Engine. In *13th International Conference on Interactive Theorem Proving (ITP 2022) (LIPIcs, Vol. 237)*, June Andronick and Leonardo de Moura (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 17:1–17:18. https://doi.org/10.4230/LIPIcs.ITP.2022.17

[24] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *NASA Formal Methods Symposium*. Springer, 41–55. https://doi.org/10.1007/978-3-642-20398-5_4

[25] Mioara Joldes, Jean-Michel Muller, and Valentina Popescu. 2017. Tight and Rigorous Error Bounds for Basic Building Blocks of Double-Word Arithmetic. *ACM Trans. Math. Softw.* 44, 2, Article 15res (Oct. 2017), 27 pages. https://doi.org/10.1145/3121432

[26] Ariel E. Kellison and Andrew W. Appel. 2022. Verified Numerical Methods for Ordinary Differential Equations. In *15th International Workshop on Numerical Software Verification (NSV'22) (LNCS, Vol. 13466)*. Springer, 147–162. https://doi.org/10.1007/978-3-031-21222-2_9

[27] Ariel E. Kellison, Andrew W. Appel, Mohit Tekriwal, and David Bindel. 2023. LAProof: a Library of Formal Accuracy and Correctness Proofs for Sparse Linear Algebra Programs. In *30th IEEE International Symposium on Computer Arithmetic*. 8 pages.

[28] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2015. Frama-C: A software analysis perspective. *Formal Aspects of Computing* 27, 3 (May 2015), 573–609. https://doi.org/10.1007/s00165-014-0326-7

[29] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers (LNCS 6355)*, Edmund M. Clarke and Andrei Voronkov (Eds.). Springer, 348–370. https://doi.org/10.1007/978-3-642-17511-4_20

[30] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. https://doi.org/10.1145/1538788.1538814

[31] Victor Magron. 2014. NLCertify: A Tool for Formal Nonlinear Optimization. In *Mathematical Software – ICMS 2014*, Hoon Hong and Chee Yap (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 315–320. https://doi.org/10.1007/978-3-662-44199-2_49

[32] Victor Magron, George Constantinides, and Alastair Donaldson. 2017. Certified Roundoff Error Bounds Using Semidefinite Programming. *ACM Trans. Math. Softw.* 43, 4, Article 34 (Jan. 2017), 31 pages. https://doi.org/10.1145/3015465

[33] Assia Mahboubi and Enrico Tassi. 2022. *Mathematical Components*. Zenodo. https://doi.org/10.5281/zenodo.7118596

[34] Mariano M. Moscato, Laura Titolo, Aaron Dutle, and César A. Muñoz. 2017. Automatic Estimation of Verified Floating-Point Round-Off Errors via Static Analysis. In *Computer Safety, Reliability, and Security – 36th International Conference, SAFECOMP'17*. Springer, 213–229. https://doi.org/10.1007/978-3-319-66266-4_14

[35] Jean-Michel Muller, Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, Mioara Joldes, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres. 2018. *Handbook of Floating-Point Arithmetic, 2nd edition*. Birkhäuser Boston. 632 pages. https://doi.org/10.1007/978-3-319-76526-6

[36] Jean-Michel Muller and Laurence Rideau. 2022. Formalization of Double-Word Arithmetic, and Comments on "Tight and Rigorous Error Bounds for Basic Building Blocks of Double-Word Arithmetic". *ACM Trans. Math. Softw.* 48, 1, Article 9 (Feb. 2022), 24 pages. https://doi.org/10.1145/3484514

[37] Hong Diep Nguyen. 2011. *Efficient algorithms for verified scientific computing: Numerical linear algebra using interval arithmetic.* Thesis. Ecole Normale Supérieure de Lyon. https://theses.hal.science/tel-00680352

[38] Michael L. Overton. 2001. *Numerical Computing with IEEE Floating Point Arithmetic*. Society for Industrial and Applied Mathematics. https://doi.org/10.1137/1.9780898718072

[39] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. 2015. Automatically Improving Accuracy for Floating Point Expressions. In *PLDI'15: 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 1–11. https://doi.org/10.1145/2813885.2737959

[40] Tahina Ramananandro, Paul Mountcastle, Benoît Meister, and Richard Lethin. 2016. A Unified Coq Framework for Verifying C Programs with Floating-Point Computations. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs (CPP 2016)*. Association for Computing Machinery, New York, NY, USA, 15–26. https://doi.org/10.1145/2854065.2854066

[41] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, and David Hough. 2013. Precimonious: Tuning assistant for floating-point precision. In *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 1–12. https://doi.org/10.1145/2503210.2503296

[42] Alexey Solovyev, Marek S Baranowski, Ian Briggs, Charles Jacobsen, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. 2018. Rigorous Estimation of Floating-point Round-off Errors with Symbolic Taylor Expansions. *ACM Transactions on Programming Languages and Systems* 41, 1 (2018), 1–39. https://doi.org/10.1145/3230733

[43] Mohit Tekriwal, Andrew W. Appel, Ariel E. Kellison, David Bindel, and Jean-Baptiste Jeannin. 2023. Verified Correctness, Accuracy, and Convergence of a Stationary Iterative Linear Solver: Jacobi Method. In *16th Conference on Intelligent Computer Mathematics*. Springer, 206–221. https://doi.org/10.1007/978-3-031-42753-4_14

Received 2023-09-19; accepted 2023-11-25