VCFloat2: Floating-point error analysis in Coq

Andrew W. Appel[1] and Ariel E. Kellison[2]

[1] Princeton University
[2] Cornell University **DRAFT October 10, 2022**

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 present a novel extension to the core of VCFloat, a Coq package for verifying floating-point C programs. Our extension improves the modularity, accuracy, and performance of VCFloat. Our extension improves the modularity fo VCFloat by adding a new shallow-embedded functional modeling language in Coq that enables VCFloat proofs to more seamlessing compose with other Coq proofs; we improve the accuracy of VCFloat by proving the correctness of certain exact and close-to-exact cases in floating-point arithmetic. With regard to performance, some of VCFloat's analyses relied on a simplifier that could cause exponential blowup of nested formulas; we implemented a prove-correct high-performance special-purpose simplifier that also discards negligible terms with a proved bound on their sum. We evaluate the performance of our novel extension using common benchmarks, and find that the extension provides competitve error bounds and good speedup compared to the timings for proof checking floating-point error bounds from other state-of-the art tools.

1 Introduction

We describe VCFloat 2.0, an open-source tool for automated floating-point round-off analysis, foundationally verified sound in Coq, with a newly improved functional modelling language for better integration with other analyses of numerical programs, a new efficient optimizer for multinomials, and a new analysis for division by powers of two—available at tinyurl.com/mwuf8ru3.[3]

We want to build machine-checked proofs, assuming only the axioms of logic and the semantics of C (or some other language) and of IEEE-754 floating point, that programs correctly and accurately approximate the solution of specified mathematical problems. We decompose the problems into,

– Prove that the program (in a low-level language such as C) correctly implements a **float-valued functional model** (typically in the ML-like calculus of your logic), using a program logic for the low-level language. In this proof one can treat floating-point operations as uninterpreted functions. To help automate this proof, use a program-logic verification tool.

[3] Warning: following that link would compromise double-blind reviewing.

– Prove that the float-valued functional model approximates a corresponding **real-valued functional model**, with a given accuracy bound. Use a tool for *roundoff analysis,* the subject of this paper.
– 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 general-purpose logic equipped with a real-analysis theorem-library.

And finally compose all three theorems together—into a single correctness-and-accuracy theorem, in a single logic, checked by a single proof-checker—so that there are no gaps in which misunderstandings might compromise soundness.

There are several tools for floating-point roundoff analysis, but not all of them produce machine-checkable proofs that can be composed with other theorems about the program. See section 11 for related work.

VCFloat 1.0 was built in 2015 by Ramananandro *et al.* [17]. It was unmaintained since then—based on obsolete versions of Coq, of CompCert, and of Coq's floating-point theory (named Flocq). We brought it up to date (with Coq, Flocq, CompCert) and made several improvements:

1. VCFloat 1.0 took input expressions from C programs as parsed by CompCert's front end. (CompCert [14, 4] is a formally verified optimizing compiler for C.) We think there's a better way to think about functional models: we have added a new front end that can reify (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 §3).
2. One could connect these functional models to any programming language, but we do connect it formally and rigorously to C. With the Verified Software Toolchain [1] one can straightforwardly prove that C programs implement those functional models (§10).
3. To support our natural-style functional models, we added new floating-point literal notations within Coq that work at any floating-point precision—single, double, half, quad, or arbitary user-specified (§3).
4. VCFloat 1.0 recognized that floating-point multiplication by nonnegative powers of 2 is exact, with no roundoff error, provided it doesn't overflow. Floating-point division by powers of 2 (or multiplication by powers of 1/2) can also be exact or highly accurate, but the analysis is more complicated because of possible underflow. We have implemented this analysis (§7).
5. VCFloat 1.0 generated interval goals that could blow up. It generates verification conditions, that the user would normally prove by calling Coq's standard field_simplify tactic followed by a call to Coq's Interval package. But field_simplify can generate an exponentially large term that causes Coq to run out of memory, and without field_simplify the Interval package sometimes generates very loose bounds. We built an efficient and accurate interval-goal optimizer in Coq's term language (§8), proved correct (§9).

2 Review of round-off error analysis

We briefly review the fundamentals of round-off analysis [16]. A floating-point number may be 0, *normal finite*, *subnormal finite*, $+\infty$, $-\infty$, or *NaN* (not a number).[4] 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 $e_{\min} \leq e \leq e_{\max}$ 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 float x *exactly* represents some real number $\mathrm{R}(x)$. But the floating-point calculation $x+y$ or $x \cdot y$ is not (usually) exact: often $\mathrm{R}(x)+\mathrm{R}(y) \neq \mathrm{R}(x+y)$.

The result of a floating point operation may have more mantissa bits than fit into the representation, so it will be rounded off. If $x+y$ is a *normal* number we know that $\mathrm{R}(x+y) = (\hat{x}+\hat{y})(1+\delta)$, for some δ such that $|\delta| \leq 2^{-M}$. That is, there is a *relative error bound*. If $x+y$ is *subnormal* then $\mathrm{R}(x+y) = \mathrm{R}(x) + \mathrm{R}(y) + \epsilon$, where $|\epsilon| \leq 2^{-E}$, where $E = e_{\max} + M - 2$. For single precision $M = 24$ and $e_{\max} = 128$, so $|\delta| \leq 2^{-24}$ and $|\epsilon| \leq 2^{-150}$.

If we know $x+y$ will be finite but don't know whether it will be normal or subnormal, then $\mathrm{R}(x+y) = (\mathrm{R}(x) + \mathrm{R}(y))(1+\delta) + \epsilon$ for some δ and ϵ bounded as above. For other operations (subtraction, division) the analysis is similar: $\mathrm{R}(x \oplus y) = (\mathrm{R}(x) \oplus \mathrm{R}(y))(1+\delta) + \epsilon$, or without the ϵ or δ (respectively) if the result is known to be normal or subnormal.

There are some special cases: if x and y have the same binary order of magnitude ($\frac{1}{2} \leq x/y < 2$) then $\mathrm{R}(x-y) = \mathrm{R}(x) - \mathrm{R}(y)$ exactly; this is called *Sterbenz subtraction*. If $y = 2^k$ for $0 \leq k$, $\mathrm{R}(x \cdot y) = \mathrm{R}(x) \cdot \mathrm{R}(y)$ 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.

3 Overview of VCFloat2

We will use this formula as a running example:

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

which arises [11] in the simulation of a harmonic oscillator by leapfrog integration with time-step $h = 1/32$. Assume $2 \leq x \leq 4$ and $-2 \leq v \leq +2$, and we compute this in single-precision floating point.

VCFloat's job is to soundly insert 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.).

[4] The IEEE-754 standard did not fully standardize the treatment of not-a-number: if x or y is a NaN with a particular payload, then $x+y$ is a NaN whose payload is machine-architecture-dependent. VCFloat (1 or 2) treats this accurately with a type-class `NaNs` giving the architecture-specific details. All of VCFloat's analysis is parametric over `NaNs`—since VCFloat generally proves that NaNs do not arise, it does not matter *which* NaNs do not arise.

We will illustrate the process from the user's point of view, 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 Tsingle (h ∗ h)%F64)%F32 computes $h \cdot h$ in double precision and the rest in single precision.

VCFloat2 comes with these exciting new functions built in:

```
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. 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 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 has already been 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 need not write them explicitly.

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 leapfrog_bmap_list : list varinfo :=
  [ Build_varinfo Tsingle _x 2 4 ; Build_varinfo Tsingle _v (−2) 2 ].
Definition leapfrog_bmap : boundsmap := a line of boilerplate mentioning leapfrog_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.[5] 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 leapfrog_bmap vmap step' (1/4000000).
 (* Theorem statement: for all x and v in the bounds of leapfrog_bmap,
  the roundoff 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 programs; see section 5.

Point B. By this point, VCFloat's core algorithm has calculated several verification conditions. In this case there are 5 of them. 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:

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

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 [6, §4.2].

[5] If you don't know the bound in advance, VCFloat2 can calculate the bound and prove it for you at the same time. We don't illustrate that here.

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:

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

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.

4 Floating point types, operations, and notation

VCFloat2 (like VCFloat1) defines a floating point type as,

```
Record type: Type := {fprec: Z; femax: Z; prec_range: 1 < fprec < femax}.
Definition Tsingle : type := TYPE 24 128    ltac:(simpl;lia).
Definition Tdouble : type := TYPE 53 1024   ltac:(simpl;lia).
```

(some details elided). This record-type enforces (via dependent types) that the precision (number of mantissa bits) must be less than the maximum exponent. 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 [5, 6]. 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.

VCFloat (1 or 2) encapsulates the precision and emax into a type, and provides these wrappers for binary floats and their operations:

```
Definition ftype ty := binary_float (fprec ty) (femax ty).
Definition BINOP op ty := op (fprec ty) (femax ty) (fprec_gt_0 ty) (fprec_lt_emax ty) mode_NE.
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 Tsingle x y, so in VCFloat2 we provide Coq *notations*:

```
Notation "x + y" := (BPLUS Tsingle x y) (at level 50, left associativity) : float32_scope.
Notation "x + y" := (BPLUS Tdouble x y) (at level 50, left associativity) : 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.

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, so we implemented an entire scientific-notation parser and pretty-printer in Coq, and plugged it in using Coq's Number Notation.

5 Reification

To represent *in a logic* a function analyzing logical formulas of type τ, one cannot write a function with type $\tau \to$ 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 $\to \tau$. The opposite process, *reification,* cannot be done within the logic. One can program a reify function from τ to expr in the tactic language of a proof assistant such as Coq. One cannot prove reify correct, but one obtains a per-instance guarantee for each $f : \tau$ by checking that reflect(reify(f)) = f. This is *proof by reflection* [2, 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.* [17]. First the term is reified into syntax trees [17, Fig. 1]:

```
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: Type := Const (ty: type) (f: ftype ty)      | Var (ty: type) (i: V)
                      | Binop (b: binop) (e1 e2: expr)      | Unop (u: unop) (e1: expr).
```

The InvShift form of rounded_unop is new in VCFloat2; see §7.

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 [14] 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:

- The functional model is an important artifact in its own right. It will be the subject of significant analysis, not only for floating-point roundoff 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. We illustrate with a few clauses:

```
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 ident (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 ident (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 ident (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 ident SterbenzMinus a' b')
 | . . .
```

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, there is different "rounding knowledge" encoded into the syntax tree.

6 The core of VCFloat

VCFloat1's core algorithm is called rndval_with_cond: "compute rounded value with verification conditions."

```
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 expression e is $(x + \frac{1}{32}v + \frac{1}{2}\frac{1}{32}\frac{1}{32}(3 - x))$,

reified into the expr syntax. Then rndval_with_cond(e) is (r, m, vcs), where r 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). The result m 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. The result vcs is a list of verification conditions, such as the one that appears (reflected, above the line) at Point B.

VCFloat1's core soundness theorem, rndval_with_cond_correct, is a machine-checked proof in Coq. It is presented as Theorem 3 by Ramananandro *et al.* [17]. Basically, it says that

- for any environment ρ mapping reified variables (such as _x and _v in our example) to floating-point numbers,
- if each of the verifications vcs is true in 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) applies this core soundness 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.

8 An efficient simplifier for Interval goals

The Interval package [6, §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, but Interval uses only univariate Taylor series.

The Interval mode that can work on our problem is (repeated) bisection of the interval. But in practice, all those 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 $x - x$ is a disaster for interval arithmetic.

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,

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

The two terms x and $-x$ have been *symbolically* canceled. With this goal, Interval computes an excellent bound.

But 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 17 terms, which is not such a problem. But when we apply VCFloat to more substantial examples, Coq runs out of memory.

A fast ring simplifier. We have implemented a high-performance special-purpose simplifier, to clean up queries before asking Interval to solve them. It is an alternative to multivariate Taylor expansion, and 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. The associative law is unnecessary and would just thrash memory by rewriting large formulas.
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: $|x_i| \leq h_i$.

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,

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

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). The field_simplify tactic could cancel these terms, but we want to efficiently *in the already-reified trees,* without first reflecting back into Coq (as would be necessary if we used field_simplify).

Step 3 cancels terms, efficiently. 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.

Then 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 our their sum added to the residual terms. The result is shown at Point D.

Efficiency of the algorithm. Generally speaking, 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).[6]

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, the entire algorithm runs in Coq in 2 to 5 seconds.[7] 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.

A more efficient algorithm. It should be possible to cancel 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. But the analyses must be sound even in the presence of round-off error. So we compute in floating-point interval arithmetic, as provided by the Coq Interval package—which uses operators that carefully round down and round up, respectively, for the bottom and top of the interval.

[6] However, since variables (x, δ, ϵ) are represented by natural numbers with unary representation, all these numbers must be multiplied by the number of variables. It would be possible to reduce this to a factor logarithmic in the number of variables by using a binary representation.

[7] 1.55 to 5.7 seconds using vm_compute on an Intel Core i7 laptop in Coq 8.15.0. It would probably be 2–4x faster using native_compute. The variation in times may be explained by whether a major-generation garbage collection occurs. Maybe.

9 Soundness theorem for simplification

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 is chosen, 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 spec of simplify_and_prune:

```
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).
```

It says, suppose you wish to prove that, with bounds-hypotheses hyps and variables vars that satisfy those hypothesis, that $|e| \leq r$. And suppose that simplify_and_prune gives you a simplified expression e_1, and claims that the total of all deleted terms is bounded by s. Then it suffices to prove $|e_1| \leq r - s$.

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

10 Proving the C program implements the model

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 [13], Verifast [10], Frama-C [12]. Furthermore, because it is embedded in Coq, one can compose, *all in 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 [11].

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 low-level functional model of a floating-point computation. But VST provides no help in reasoning *about* the functional model. That is what VCFloat can do, with our improvements.

11 Related work and performance evaluation

Table 1. Performance of VCFloat2 compared to Gappa and FPTaylor.

Benchmark	Gappa	FPTaylor-v	VCFloat2	Error %	Time (s)
carbonGas	2.70e-08	9.20e-09	8.05e-08	8.75	3.31
doppler1	2.10e-13	1.60e-13	1.21e-12	7.56	72.03
doppler2	4.00e-13	2.90e-13	1.3e-12	4.10	63
doppler3	1.10e-13	8.30e-14	1.75e-13	2.07	38.41
himmilbeau	1.10e-12	1.40e-12	2.30e-12	2.09	3.62
jetEngine	8.30e06	1.40e-11	1.38e02	9.84e12	120
t_div_t1	1.00e03	5.80e-14	4.39e-16	0.01	0.59
kepler0	1.30e-13	9.50e-14	2.20e-13	2.32	4.24
kepler1	5.40e-13	3.60e-13	1.64e-12	4.57	11.97
kepler2	2.90e-12	2.00e-12	6.17e-12	3.08	28.71
predprey	2.10e-16	1.90e-16	1.99e-15	10.5	12.66
rigidBody1	3.00e-13	3.90e-13	3.05e-13	1.02	1.59
rigidBody2	3.70e-11	5.30e-11	3.90e-11	1.05	4.03
verhulst	4.20e-16	3.30e-16	2.32e-16	0.70	7.64
turbine1	8.40e-14	2.40e-14	1.15e-12	54.17	48.41
turbine2	1.30e-13	2.60e-14	5.20e-12	196.15	32.35
turbine3	4.00e01	1.30e-14	4.00e01	3.08e15	26.16

There are several tools that perform round-off error analysis and generate machine-checkable proofs: Gappa [3] is implemented in C++ and generates Coq proof scripts; PRECiSA [15] is implemented in Haskell and C and generates proofs in PVS; FPTaylor [18] is implemented in OCaml and is capable of generating

proofs in HOL Light; and Daisy [**?**] is implemented in Scala and has certificate generation and checking in Coq and HOL4 [**?**].

We measured the performance of VCFloat2 using the FPBench benchmarks[8]. We chose seventeen benchmarks for which there are previously reported results for FPTaylor and Gappa. The performance of VCFloat2 on the FPBench examples are given in Table 1. The column labeled "Error %" gives the factor by which the error bounds computed by VCFloat2 differ from the error bounds computed by FPTaylor that have been verified in HOL Light (FPTaylor-v) as reported by Solovyev and co-authors [18]. VCFloat2 finds bounds within an order of magnitude of the best performers on twelve of the seventeen examples. VCFloat2 fails to find a useful bound on two of the remaining five benchmarks (*jetEngine* and *turbine3*), which is similar to Gappa's preformance. Total verification time for all seventeen benchmarks took 7.83 minutes. 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 0.96 minutes each compared to an average of 37 minutes each for FPTaylor.

The failure of VCFloat on *turbine3* and *jetEngine* is not surprising. The Prune tactic provided by VCFloat2 currently 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. This is strong motivation for implementing proved-correct multivariable Taylor models Taylor approximations in Coq.

Finally, VCFloat 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 want to *import* problem specifications from the proof assistant into the tool, and *export* round-off proofs from the tool to the proof assistant.

There are other roundoff-analysis tools that do not generate proofs. One of these, Satire [9], is compositional in one way that the other tools (including VCFloat2) are not: it permits input variables to be treated as approximations to unknown real values. This would be a useful VCFloat2 feature in future work.

There are precision-tuning tools (e.g., [7]) that suggest which floating-point operations in a program should be done at double-precision, or single-precision, or half-precision, etc.; but they do not produce proofs. The tuned programs could, in principle, be proved by VCFloat2 (or by Gappa, PRECiSA, FPTaylor).

12 Future work

Improving the Interval package to handle multivariate Taylor analysis would give VCFloat power comparable to FPTaylor [18]; currently for multivariable goals (such as the example in this paper) we use Interval's bisection method.

Allowing input variables to come with their own error bounds (instead of being assumed exact) would give VCFloat modularity comparable to Satire [9].

VCFloat does not yet have a large library of axioms (or proofs) about library functions (e.g., trigonometric), which makes it difficult to test VCFloat on the FPBench benchmark suite [8].

13 Conclusion

Floating-point round-off analysis should 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. In such a context, it's important to have a language for writing floating-point functional models that clearly and simply relate to real-valued functional models. The individual analysis tools, of which round-off error is just one example, should be able to take their inputs and their outputs in the same logic as the rest of the framework.

VCFloat2 is quite useful in a toolchain for such analyses. Below it, the VST tool reasons about C program correctness and connects to functional models of the kind we have described here. Above VCFloat, tools for reasoning *with proofs in Coq* about the properties of real-valued functional models are an exciting area for future research.

References

1. Andrew W. Appel. Verified software toolchain. In Gilles Barthe, editor, *ESOP'11: European Symposium on Programming*, volume 6602 of *LNCS*, pages 1–17. Springer, 2011.
2. Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development*. Springer, 2004.
3. Sylvie Boldo, Jean-Christophe Filliâtre, and Guillaume Melquiond. Combining Coq and Gappa for certifying floating-point programs. In *International Conference on Intelligent Computer Mathematics*, pages 59–74. Springer, 2009.
4. Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. A formally-verified c compiler supporting floating-point arithmetic. In *2013 IEEE 21st Symposium on Computer Arithmetic*, pages 107–115. IEEE, 2013.
5. Sylvie Boldo and Guillaume Melquiond. Flocq: A unified library for proving floating-point algorithms in Coq. In *2011 IEEE 20th Symposium on Computer Arithmetic*, pages 243–252. IEEE, 2011.
6. Sylvie Boldo and Guillaume Melquiond. *Computer Arithmetic and Formal Proofs: Verifying Floating-point Algorithms with the Coq System*. Elsevier, 2017.
7. Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. Rigorous floating-point mixed-precision tuning. In *POPL'17: 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, page 300–315, New York, NY, USA, 2017. Association for Computing Machinery.
8. Nasrine Damouche, Matthieu Martel, Pavel Panchekha, Jason Qiu, Alex Sanchez-Stern, and Zachary Tatlock. Toward a standard benchmark format and suite for floating-point analysis. In *Numerical Software Verification (NSV'16)*, July 2016.

9. Arnab Das, Ian Briggs, Ganesh Gopalakrishnan, Sriram Krishnamoorthy, and Pavel Panchekha. Scalable yet rigorous floating-point error analysis. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2020.
10. Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for C and Java. In *NASA Formal Methods Symposium*, pages 41–55. Springer, 2011.
11. Ariel E. Kellison and Andrew W. Appel. Verified numerical methods for ordinary differential equations. In *15th International Workshop on Numerical Software Verification (NSV'22)*. Springer LNCS (forthcoming), August 2022.
12. Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, May 2015.
13. K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, LNCS 6355, pages 348–370. Springer, 2010.
14. Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.
15. Mariano M. Moscato, Laura Titolo, Aaron Dutle, and César A. Muñoz. Automatic estimation of verified floating-point round-off errors via static analysis. In *Computer Safety, Reliability, and Security - 36th International Conference, SAFECOMP'17*, pages 213–229, 2017.
16. Michael L. Overton. *Numerical Computing with IEEE Floating Point Arithmetic*. Society for Industrial and Applied Mathematics, 2001.
17. Tahina Ramananandro, Paul Mountcastle, Benoît Meister, and Richard Lethin. 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, page 15–26, New York, NY, USA, 2016. Association for Computing Machinery.
18. Alexey Solovyev, Marek S Baranowski, Ian Briggs, Charles Jacobsen, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic Taylor expansions. *ACM Transactions on Programming Languages and Systems*, 41(1):1–39, 2018.