A Verified Certificate Checker for Floating-Point Error Bounds

Heiko Becker\(^{1}\), Eva Darulova\(^{1}\), and Magnus O. Myreen\(^{2}\)

\(^{1}\) MPI–SWS \{hbecker,eva\}@mpi-sws.org
\(^{2}\) Chalmers University of Technology myreen@chalmers.se

Abstract. Being able to soundly estimate roundoff errors in floating-point computations is important for many applications in embedded systems and scientific computing. Due to the unintuitive nature of floating-point arithmetic, automated static analysis tools are highly valuable for this task. The results, however, are only as correct as the implementations of the static analysis tools.

This paper presents a new modular framework for the analysis of finite-precision computations which computes sound roundoff error bounds fully automatically. The main focus of this paper are the correctness certificates generated by our framework. These can be checked independently by our checker functions, thus providing more confidence in the analysis results.

We present implementations of certificate generation and checking for both Coq and HOL4 and evaluate it on a number of examples from the literature. The experiments use both in-logic evaluation of Coq and HOL4, and execution of extracted code outside of the logics: we benchmark Coq extracted unverified OCaml code and a CakeML-generated verified binary.

1 Introduction

Numerical programs, common in scientific computing or embedded systems, are often implemented in floating-point arithmetic. This approximation of real numbers inevitably introduces roundoff errors, potentially making the computed results unacceptably inaccurate. The unintuitive nature of floating-point arithmetic as well as the discrepancy between its finite nature and continuous real arithmetic make accurate and sound error estimation challenging. Automated tool support is thus highly valuable.

This fact was already recognized previously and resulted in a number of static analysis techniques and tools \([17,35,10,12]\) for computing sound worst-case absolute error bounds on numerical errors. The results of such static analysis tools are, however, only as correct as the tools’ implementation. Where formalizations or independently checkable proofs are provided \([12,35]\), we found each development to be either overly specialized, not easily extensible or reusable, or not fully automated. Tools implemented to be used interactively \([13]\) require expert knowledge on floating-point arithmetic, where our goal was to make our tool usable by non-experts.
This paper describes a new fully automated static analysis tool with a particularly strong soundness argument. Our tool, called Daisy, computes sound roundoff error bounds for floating-point computations and generates a proof certificate which can be independently checked by our certificate checking functions fully automatically. We have proved the correctness of these checking functions in both Coq and HOL4. The design of Daisy aims to be as modular and reusable as possible. The certificate checker functions, which are the focus of this paper, can be seen as a modular part of the new Daisy tool, since they can be used as part of Daisy or independently of it.

The reason for doing the verification in two theorem provers is partly for the extra assurance, partly to compare the two provers, but mostly to be able to connect to projects in both provers. In Coq, we hope to link to the CompCert compiler [24] and CertiCoq [1]; and in HOL4 to CakeML [36]. At the time of writing, we do not have connections to CompCert or CertiCoq. However, we already have a connection in HOL4 to CakeML: we use the CakeML toolchain [36,31] to produce a verified binary of our certificate checker.

Techniques for roundoff error verification can roughly be divided into forward static analyses [17,10,12] and global optimization [35]. We choose to verify the results of a forward static analysis, as it is also applicable to fixed-point arithmetic, which we aim to support in Daisy in the future.

Contributions

– This paper explains our modular, easy extensible and fully automated approach to certification of floating-point error bound analysis. The static analysis is implemented in the Daisy framework, where a certificate consists of a simple encoding of the analysis result and the analyzed function.

– Our checker is implemented and proven correct in both Coq and HOL4. (sources at: [https://gitlab.mpi-sws.org/AVA/daisy-certification-public]) This allows reuse of our results when combining Daisy with other developments in these theorem provers.

– We are the first to provide an efficient and verified way of checking our certificates by extracting a verified binary version of the certificate checker function from HOL4. The verified binary is produced by in-logic compilation using the CakeML compiler toolchain.

– We experimentally evaluate implementations of the checker on examples from the literature. The results are promising and justify the claim that our approach to certificate checking is feasible.

2 Overview of Error Estimation and Certificate Checking

We first give a high-level overview of the Daisy framework, its static roundoff error analysis, and the certificate generation and checking approach.

Daisy is meant to be a modular and easily extensible successor of the Rosa tool [10]. In this paper we focus on the first extension upon Rosa, the verified
def doppler(u: Real, v: Real, T: Real): Real = {
  require(-100.0 <= u && u <= 100 && 20 <= v && v <= 20000 && -30 <= T && T <= 50)
  val t1 = 331.4 + 0.6 * T
  (-(t1)*v) / ((t1 + u)*(t1 + u))
}

Fig. 1. Example input function computing the doppler shift

checking of the computed roundoff errors. Daisy’s input language is a (non-executable) real-valued subset of Scala. Each program consists of a number of functions which are analyzed separately. Figure 1 shows an example [10]. In the precondition of a function (the require clause) the user provides the ranges of all input variables. Each function’s body consists of an arithmetic expression with possibly local variable declarations. We do not consider loops or conditionals at this point; these have been shown to be challenging in the presence of numerical uncertainties [11] and we leave their formal treatment to future work.

Given a function and a uniform floating-point precision, Daisy computes a sound roundoff error bound on the function’s result and also generates the corresponding executable floating-point code in Scala. In the future, we plan to use our formalizations in Coq and HOL4 to support CakeML, CertiCoq and CompCert C as backends for code generation.

2.1 Floating-Point Arithmetic

We assume IEEE754 floating-point arithmetic in both the static analysis as well as the formalization, rounding-to-nearest rounding mode, and we further assume the following standard abstraction of arithmetic operations:

\[ x \circ_{fl} y = (x \circ y)(1 + \delta), \quad |\delta| \leq \varepsilon_m \]  

where \( \circ \in +, -, *, / \) and \( \circ_{fl} \) denotes the respective floating-point version, and \( \varepsilon_m \) is the so-called machine epsilon. The latter represents the maximum relative error for a single arithmetic operation. Daisy also supports unary negation, which does not incur a roundoff error. In the rest of the paper, we assume standard double floating-point precision (\( \varepsilon_m = 2^{-53} \)), but our approach carries over naturally to any other precision satisfying Equation 1 above. We further consider NaNs (not-a-number special values), infinities and ranges containing only denormal floating-point numbers to be errors and Daisy detects these automatically. We note that under these assumptions Equation 1 is indeed a sound abstraction. We also assume a uniform floating-point precision for the entire program, although an extension to mixed-precision arithmetic would be straightforward.

We will denote by \( f \) and \( x \) a real-valued function and variable, respectively, and by \( \tilde{f} \) and \( \tilde{x} \) their floating-point counterparts. The worst-case absolute error that Daisy computes is then:

\[ \max_{x \in [a,b]} |f(x) - \tilde{f}(\tilde{x})| \]  

3
where \([a, b]\) is the range for \(x\) given in the precondition. The input \(x\) may not be representable in floating-point arithmetic, and thus Daisy considers an initial error on the input: \(|x - \tilde{x}| = |x| \cdot \delta, \ \delta \leq \varepsilon_m\) which follows from Equation 1. This definition extends to multi-variate \(f\) component-wise.

We estimate and generate certificates for absolute errors. An automated and general estimation of relative errors \((|f(x) - \tilde{f}(\tilde{x})|/|f(x)|)\), though it may be more desirable, presents a significant challenge today. To the best of our knowledge, state-of-the-art static analyses only compute relative errors from the absolute error bound, and this only if the range of the expression in question (i.e. the range of \(f(x)\)) does not include zero. Thus, relative error bounds computed by today’s tools are not more informative or general than absolute error bounds.

2.2 Overview of Error Analysis

Daisy follows the same approach for computing absolute error bounds as Rosa, Fluctuat [17] and Gappa [13] and bounds absolute errors by a data-flow analysis. The magnitude of absolute floating-point roundoff errors depends on the magnitude of values of all intermediate subexpressions (this can easily be seen from Equation 1). Thus, if we want to accurately bound roundoff errors, we need to be able to bound the ranges of all (intermediate) expressions first.

One may think that just evaluating the program in interval arithmetic and interpreting the width of the resulting interval as the error bound would be sufficient. While this is certainly a sound approach, it computes too pessimistic error bounds in general. This is especially true if we consider relatively large ranges on inputs; we have no way of distinguishing which part of the interval width is due to the input interval or due to accumulated roundoff errors. Hence we need to compute ranges and errors separately.

At a conceptual level, we compute error bounds in two steps:

- **range analysis** Compute sound range bounds for all intermediate expressions.
- **error analysis** Using the previously computed ranges, propagate errors from subexpressions and compute the new worst-case roundoffs.

Both steps are performed recursively on the abstract syntax tree (AST) of the arithmetic expressions. In practice, we compute these two stages at the same time for efficiency reasons.

A side effect of this separation is that it provides us with a modular approach: we can choose different range arithmetics with different accuracy-efficiency trade-offs for ranges and errors and for different parts of a program, which is an opportunity we would like to further explore in the future. For now, we focus on the design of the overall framework and consider interval arithmetic [29] for both the ranges and the errors.

2.3 Certification of Error Analysis Results

Daisy encodes the result of the error analysis in a correctness certificate and also prints the inferred program information for the user. For each function \(f\), the certificate contains:
– an encoding of the abstract syntax tree of the function body,
– the precondition as a map from (input) variables to intervals,
– the analysis result as two maps $\Phi_R$ and $\Phi_E$ which map each subexpression of $f$ to an interval and a roundoff error, respectively.

The intervals in $\Phi_R$ are the real-valued ranges at each intermediate AST node which have been computed during the static analysis. Similarly, $\Phi_E$ stores the roundoff errors for each node, represented as a single (rational) value. The certificates generated for Coq, HOL4 and the extracted binaries are structurally the same and only differ in syntax.

Figure 2 gives an overview of our certificate verification structure. Note that our certificates only encode the analysis result, but do not contain a complete proof script for checking them. The check is done by our checker functions which we have implemented in Coq and HOL4. We have proven the functions correct once and for all, i.e. we have proven that if the functions return true, the analysis results are correct with respect to the execution semantics.

Analogously to the analysis, our certification is also modular and checks ranges and errors separately, allowing for different range arithmetics and error computations in the future. Hence, we have two checker functions which check $\Phi_R$ and $\Phi_E$ separately.

To verify a certificate generated by Daisy, it suffices to run the checker functions in Coq or HOL4 directly on it. However, while both provers natively support evaluation of functions, this is not very efficient. To speed up the certificate checkers, we have used the CakeML in-logic compilation toolchain [31], to extract a verified binary from our HOL4 checker definitions. Since the CakeML compiler is fully verified, the binary enjoys the very same correctness guarantees as the certificate checkers implemented in HOL4. Similarly, we have used the extraction mechanism [25] in Coq to extract an, albeit unverified, binary. The

Fig. 2. Overview on the structure of the Daisy framework
binary implementations of the checkers run natively and are thus significantly more efficient, as our experiments in section 5 demonstrate.

3 Certification of Error Analysis Results

Next, we focus on the technical details of our certificate checking. We first explain how our range and error checker functions work, and later discuss their soundness proof (subsection 3.4). Our checker functions have the same structure in both Coq and HOL4, which we first explain for arithmetic expressions only and then with let-bindings.

3.1 Checking Range Analysis Results

The range checker is implemented in the function validIntervalBounds(e, P, \Phi_R), which takes as input an expression e, the precondition P which captures the constraints on the input variables, and the real-valued ranges which are to be checked in \Phi_R. validIntervalBounds verifies by structural recursion on the AST that for each subexpression e’ of e, \Phi_R(e’) is a sound enclosure of the true range, which is computed inside the theorem prover with interval arithmetic. That is, we check the ranges in \Phi_R by effectively recomputing them inside the prover.

A constant n is trivially contained in the point interval [n, n]. Thus, the checker verifies that [n, n] \subseteq \Phi_R(n), where \subseteq interprets intervals as sets in the obvious way. For a variable x, the range is given by the precondition P and it needs to hold that P(x) \subseteq \Phi_R(x). For a binary expression e = e_1 \circ e_2, validIntervalBounds verifies that

\[ \Phi_R(e_1) \circ# \Phi_R(e_2) \subseteq \Phi_R(e_1 \circ e_2) \]

where \circ# is the arithmetic operation \circ interpreted on intervals. Unary negation is handled analogously.

3.2 Checking Error Analysis Results

The error checker is implemented in the function validErrorBounds(e, \Phi_R, \Phi_E), which takes as input the expression e, the range analysis result and the error result which is to be checked. That is, validErrorBounds assumes that the ranges have been verified independently. It verifies by structural recursion on the AST of e that for each subexpression e’ of e, \Phi_E(e’) is a sound bound on the absolute roundoff error.

For constants and variables, the error bounds are straight-forwardly derived using Equation 1 and the range analysis result. For arithmetic, the error check is more involved; we describe validErrorBounds for the case of addition: e = e_1 + e_2. In our notation, the absolute error is given by Equation 2 as

\[ |e - \tilde{e}| = |(e_1 + e_2) - (\tilde{e}_1 + \tilde{e}_2)| \]
using the abstraction of floating-point arithmetic from Equation 1 we obtain
\[ \tilde{e}_1 + f_1 \tilde{e}_2 = \tilde{e}_1 + \tilde{e}_2 + (\tilde{e}_1 + \tilde{e}_2) \cdot \delta \quad |\delta| \leq \varepsilon_m \]
combining the two and applying the triangle inequality, we obtain
\[ |(e_1 + e_2) - (\tilde{e}_1 + f_1 \tilde{e}_2)| \leq |e_1 - \tilde{e}_1| + |e_2 - \tilde{e}_2| + |(\tilde{e}_1 + \tilde{e}_2)| \cdot \varepsilon_m \quad (3) \]
|e_1 - \tilde{e}_1| and |e_2 - \tilde{e}_2| are the roundoff errors of the operands, which are propagated simply by addition. |(\tilde{e}_1 + \tilde{e}_2)| \cdot \varepsilon_m is the new roundoff error committed by the addition. Note that it depends on the magnitude and thus the range of \( \tilde{e}_1 \) and \( \tilde{e}_2 \).

To upper bound the right hand side of Equation 3 we use these two properties of intervals.

**Lemma 1.**
\[ a \in [a_{lo}, a_{hi}] \Rightarrow a \leq \max(|a_{lo}|, |a_{hi}|) \]
The first property (Lemma 1) states that for each interval, the maximum absolute value of the outer points is an upper bound on any element in the interval.

**Lemma 2.**
\[ |a - \tilde{a}| \leq e \land a \in [a_{lo}, a_{hi}] \Rightarrow \tilde{a} \in [a_{lo} - e, a_{hi} + e] \]
The second property (Lemma 2) states that if we know the error on \( a \) and \( a \)'s interval enclosure, we can compute an interval for the corresponding floating-point value by widening the bounds of the interval by the error. We have proven both lemmas in Coq and HOL4.

The remaining steps of the computation of the upper bound of Equation 3 are straight-forward and use the range analysis result from \( \Phi_R \) as well as the already verified error bounds on the subexpressions \( e_1 \) and \( e_2 \) in \( \Phi_E \). Similar bounds can be derived for the other arithmetic operations, however, for multiplication and division, the propagation of errors is more involved. E.g. for \( e = e_1 \cdot e_2 \) we obtain
\[ |(e_1 \cdot e_2) - (\tilde{e}_1 \cdot \tilde{e}_2)| \leq |e_1 \cdot e_2 - \tilde{e}_1 \cdot \tilde{e}_2| + |\tilde{e}_1 \cdot \tilde{e}_2| \cdot \varepsilon_m \]
and similarly for division. Note that the propagation error \( |e_1 \cdot e_2 - \tilde{e}_1 \cdot \tilde{e}_2| \) depends on both the real-valued and the float-valued ranges and requires 16 and 32 sub-cases for multiplication and division, respectively. The latter also involves checks for division-by-zero.

### 3.3 Checking Analysis Results for let-Bindings

Local variables or let-bindings, such as in our example in Figure 1, allow for the reuse of evaluation results, both at runtime as well as in the certificate checker. To distinguish between let-bound variables and function parameters, we extend the checker functions with a parameter \( D \) tracking let-bound variables: \( \text{validIntervalBounds}(f, P, \Phi_R, D) \) and \( \text{validErrorBounds}(f, \Phi_R, \Phi_E, D) \).

For an assignment \( x = e \), where \( e \) is an arithmetic expression, the wrapping checker functions need to verify, in addition to verifying \( e \) as before, that the ranges of \( x \) and \( e \) coincide. That is, for a variable \( x \) in \( D \) it must hold \( \Phi_R(x) = \Phi_R(e) \). Since storing a value does not introduce additional roundoff errors, the same property holds for \( \Phi_E \). If a variable \( x \) is not in \( D \), it is checked as before.
3.4 Soundness

We have proved in both Coq and HOL4 that it suffices to run both checker functions on the certificate to show the correctness of the static analysis result. We call this property the soundness of the checker functions. For this, the soundness proof relates a succeeding run of the functions validIntervalBounds and validErrorBounds to the semantics of the analyzed function.

We have formalized the semantics of functions according to Equation 1. For binary addition, the rule is the following:

\[
\frac{(e_1, E) \Downarrow^{\varepsilon_m} v_1 \quad (e_2, E) \Downarrow^{\varepsilon_m} v_2 \quad |f| \leq \varepsilon_m}{(e_1 + e_2, E) \Downarrow^{\varepsilon_m} (v_1 + v_2) \ast (1 + \delta)}
\]

The relation is parametric in the machine epsilon, \(\varepsilon_m\) and uses environment \(E\) to track values of bound variables. Real-valued executions instantiate the machine epsilon with 0. The rules for subtraction, multiplication and division are defined analogously. Unary negation does not introduce a new roundoff error.

The overall soundness theorem is the following:

**Theorem 1.** Let \(f\) be a real-valued program, \(P\) a precondition constraining the free variables of \(f\), \(\Phi_R\) a range analysis result, and \(\Phi_E\) an error analysis result. Assume \(f\) evaluates to a value \(v\) and \(\tilde{f}\) to \(\tilde{v}\). Then

\[
\text{validSSA}(f) \land \text{validIntervalBounds}(f, P, \Phi_R, D) \land \\
\text{validErrorBounds}(f, \Phi_R, \Phi_E, D) \implies \\
|v - \tilde{v}| \leq \Phi_E(f)
\]

We begin by giving an idea of the proof for validIntervalBounds and validErrorBounds first, and then explain function validSSA and how the proof needs to be adapted to support let-bindings.

**Range Checker** For validIntervalBounds, soundness means that from a succeeding run of the checker on a given \(e\) and \(\Phi_R\), if \(e\) evaluates to a value \(v\), then \(v\) is contained in \(\Phi_R(e)\). The proof is by structural induction on \(f\). To prove soundness of validIntervalBounds we had to prove monotonicity of interval arithmetic for each supported binary operator and negation.

**Lemma 3.** Let \(I\) and \(J\) be intervals and \(a, b\) real values. Then

\[
a \in I \land b \in J \Rightarrow (a \circ b) \in I \circ^\# J
\]

**Error Checker** If validErrorBounds\((e, \Phi_R, \Phi_E)\) succeeds, \(e\) evaluates to \(v\) and \(\tilde{e}\) evaluates to \(\tilde{v}\), we want to show that \(|v - \tilde{v}| \leq \Phi_E(e)\). The challenge in this proof lies in the fact that we reason about two different executions of similar expressions, \(e\) and \(\tilde{e}\). The real-valued values in the environment \(E\) may not be representable in the floating-point environment \(\tilde{E}\) and thus the values for the related variables
EmptySim \quad \frac{(_\mapsto \perp) \sim_{\emptyset} (_\mapsto \perp)}{}

AddFreeVar \quad \frac{E \sim_{V} \tilde{E} \quad x \notin V \quad |v - \tilde{v}| \leq v \ast \varepsilon_m}{(E[x \mapsto v]) \sim_{(\{x\} \cup V)} (\tilde{E}[\tilde{x} \mapsto \tilde{v}])}

Fig. 3. Environmental Approximation Relation \sim without handling of defined variables.

Let-Bindings To extend the soundness proofs to validIntervalBounds and validErrorBounds, we have to check that the analyzed function \( f \) is in SSA form. For this we use the formalization of SSA based on renamed apart programs defined in the LVC framework [34] since it nicely states SSA form using sets of variables. We have implemented it in the function validSSA, which is proven sound in both Coq and HOL4. Soundness of validSSA states that if it runs successfully on function \( f \), then \( f \) is in SSA form.

Furthermore, we need to adapt the approximation relation \( \sim \) to include let-bound variables. Take \( f := \text{Let} x = e \text{ in } g \). If \( \Phi_E \) has been validated for \( e \), and the check \( \Phi_E(x) = \Phi_E(e) \) succeeds, the roundoff error \( |x - \tilde{x}| \) is upper bound by \( \Phi_E(x) \). We extend \( \sim \) by the rule in Figure 4 that allows us to record this information inside the relation. For this, we add two parameters to the relation: a set \( D \), tracking variables added to both environments using let-bindings and the error analysis result \( \Phi_E \). The sets \( D \) and \( V \) are used to distinguish whether a variable \( x \) is free or let-bound.

AddDefinedVar \quad \frac{E \sim_{(\Phi_E), V, D} \tilde{E} \quad x \notin V \cup D \quad |v - \tilde{v}| \leq \Phi_E(x)}{(E[x \mapsto v]) \sim_{(\Phi_E), V \cup \{x\} \cup D} (\tilde{E}[\tilde{x} \mapsto \tilde{v}])}

Fig. 4. Rule for let-bound variables for \( \sim \)
Tying the pieces together, we obtain the soundness of the certificate checker for Daisy as the conjunction of the results of the functions validSSA, validIntervalBounds and validErrorBounds.

### 3.5 Division Bug Found

While proving soundness of division, we found a subtle bug in Daisy’s static analysis. The error bounds are only sound in the absence of division-by-zero errors, but only the real-valued range of the denominator was checked for whether it contains zero. It is possible, however, that the real-valued range does not contain zero, while the corresponding floating-point range does, essentially due to large enough roundoff errors.

### 3.6 Formalization Details

The input of Daisy is encoded in the certificate as the AST of the analyzed function using the following grammar:

\[
x \in \mathbb{N} \quad c \in \mathbb{V} \\
e_1, e_2 ::= x \mid c \mid -e_1 \mid e_1 \circ e_2 \quad \circ \in \{+, -, \ast, /\}
\]

\[
f ::= \text{Let } x = e \text{ in } f \mid e
\]

where the let-bindings encode local variable declarations. Since the input to Daisy is a valid Scala program, let-bindings appear only at the top level. Variables \( x \) are encoded as natural numbers. The definition is parametric in the type of constants \( \mathbb{V} \).

Roundoff errors and our theorems relate real-valued executions to floating-point ones and we thus need a way to represent the numbers and also compute on them. However, the latter is problematic for infinite-precision reals. Daisy uses internally rationals (\( \mathbb{Q} \)) implemented with big integers for the numerator and denominator and we use rationals to represent the values in the certificates. To relate these values to the real-valued (\( \mathbb{R} \)) executions in the theorem statement, we use the fact that rationals are a subset of the real type in HOL4, and in Coq we use the translation \( \mathbb{Q} \rightarrow \mathbb{R} \) and exploit that our AST is parametric in the constant type by instantiating it with \( \mathbb{Q} \) for computations and \( \mathbb{R} \) for theorems.

### 4 Extracting a Verified Binary with CakeML

Running the range and error checker functions in Coq and HOL4 directly is quite inefficient (see our experiments in [section 5]). We have thus extracted a verified binary from our HOL4 checker function definitions, and an unverified binary for Coq. We are aware of the work on certified extraction from Coq in the CertiCoq [H] project, but at the time of writing, the project has not yet been released publicly.

We have implemented in HOL4 and Coq an unverified lexer and parser for the encoding of the certificates. The lexer and parser are included in the extracted binaries in both Coq and HOL4.
Extracting from HOL4 For extracting a binary from HOL4, we use the CakeML proof-producing synthesis tool [31] which translates ML-like HOL4 functions into deeply embedded CakeML programs that exhibit the same behaviour. For each of the arithmetic operations over the real type that we used in the HOL4 development, we define a translation into a representation of the rationals in CakeML. Each HOL real is represented as a pair of integers that implement $\frac{n}{d}$. Here $n$ is an integer and $d$ a positive integer. CakeML’s integers are arbitrary precision integers.

CakeML and HOL4 have different notions of equality. Since we perform equality tests in the certificate checkers, we had to prove that our newly defined representation of real numbers respects CakeML’s semantics for structural equality. For this purpose, we had to require and prove that our representation of rationals maintains $gcd(n, d) = 1$.

When translating a HOL4 function into CakeML code, the CakeML toolchain generates preconditions that exclude runtime exceptions, e.g. divisions by zero. We have shown that all generated preconditions are always satisfied, hence the specification theorem for the generated ML code does not have any preconditions.

Having compiled the CakeML libraries beforehand, we can compile the checking functions into a verified binary in around 30 minutes on the same machine as we used for the experiments in section 5. Checking the certificate with the binary is then extremely fast, since no theorem prover logic is loaded.

Extracting from Coq Coq natively supports unverified extraction into OCaml code [25]. We used the existing libraries for translating Coq numbers into OCaml’s Big_int type from the base library. OCaml supports both compilation to native code and byte code. Native-code compilation (“ocamlopt”) produces a standalone executable which does not have any additional dependencies, while byte-code compilation (“ocamlc”) produces code that is run by the “ocamlrnu” interpreter. We have used both options in our experiments.

5 Evaluation

We have evaluated Daisy and its certificate checkers on examples taken from the Rosa [9] and real2float [27] projects. Each benchmark consists of one or more separate functions. Daisy analyzes all functions of one benchmark together and produces one certificate containing a call to the certificate checker for each separate function. The ‘# ops.’ column gives the number of arithmetic operations in the whole benchmark.

Table 1 shows the running times of Daisy, Coq, HOL4 and the extracted binaries on our examples. The running times are the elapsed wall clock times measured by the UNIX time command. All times are in seconds.

The execution time of Daisy is the end-to-end time, including parsing the input file, running the analysis and generating the certificate. The running times

---

3 For our experiments we used a machine with a four core Intel i3 processor with 3.3GHz, 8 GB of RAM, running Debian 8.
reported in the Coq and HOL4 columns are for checking the certificates inside the logic of the theorem prover. This includes type-checking the definitions inside the certificate, but not verifying the soundness of the checker functions, which should be done offline once. For the evaluation in logic, the Coq scripts use the `vm_compute` tactic to evaluate. In HOL4, we use the tactic `EVAL_TAC`.

Both the extracted OCaml and CakeML binaries first parse the input and then call the certificate checking function if parsing succeeded. The “HOL4 Binary” column refers to the verified extracted binary. Since OCaml can both compile to native code and byte code, we measured both running times and compare them separately as “Coq Native” and “Coq Byte” respectively.

Since the execution times of Daisy, Coq and HOL4 are substantial, we report them for a single run. For the binaries however, the execution times are much smaller, so that we report the times for checking the certificates hundred times.

We observe that the evaluation of the Coq checker in logic is faster than the evaluation of the HOL4 checker. This is probably due to the fact that evaluation in HOL4 still passes through the kernel in the form of inference rule applications.
6 Related Work

The tools FPTaylor [35], Gappa [13], Gappa++ [26], Precisa [28] and VCFloat [32] are most closely related to our work in that they are able to formally verify floating-point roundoff errors. Each tool supports a different set of features and techniques, which we summarize in Table 2. We use the abbreviations ‘IA’ for interval arithmetic and ‘AA’ affine arithmetic [14], which is a more sophisticated range analysis than interval arithmetic, able to track linear correlations. ‘Taylor Approx.’ is the abbreviation for the global optimization technique based on Taylor polynomials used in FPTaylor. In the control flow row, ‘let’s’ stands for support of let-bindings, ‘let’s +’ stands for more sophisticated support, including conditionals and/or loops. Note that Daisy is the only tool to support both Coq and HOL4, which are needed for linking with the developments of CompCert, CertiCoq as well as CakeML. Furthermore, Daisy provides full automation in combination with a verified binary. In the rest of this section, we put our work into a broader context.

Sound Verification of Floating-point Computations Floating-point computations play a role in many applications and have thus been considered before in the

| Tool        | Daisy | FPTaylor | Gappa | Gappa++ | Precisa | VCFloat |
|-------------|-------|----------|-------|---------|---------|---------|
| Supported   | Coq   | HOL4     | Coq   | Coq     | PVS     | Coq     |
| Prover      | HOL-Light | Full    | Partial | Partial | Full    | Partial |
| Automation  | Full  | Full     | Partial | Partial | Full    | Partial |
| Control Flow| let’s | No       | let’s  | let’s    | let’s+  | let’s+  |
| Verification| Taylor| IA       | Appr.  | IA       | AA      | IA      |
| Method      | IA    | Yes      | Yes    | Yes      | No      | No      |
| Transcendental Functions | No | Yes | Yes | Yes | No | No |
| Mixed Precision | No | No | No | No | No | Yes |
| Verified Binary | Yes | No | No | No | No | No |
| Input Language | Scala | Custom | Custom | Custom | PVS files | C-light |

Table 2. Comparison of supported features of related tools

In Coq the functions that are evaluated need not be checked by the kernel when being run, apart from type-checking the arguments. All of our extracted binaries are able to validate certificates significantly faster than in-logic evaluation.

The benchmarks involving divisions (marked in the table by ‘*) cause peaks in the running times. We suspect that this is due to the increased complexity of the additional checks which need to be performed (e.g. checking for division-by-zero) as well as the more involved computation of the propagation error, as explained in subsection 3.2.

Sound Verification of Floating-point Computations Floating-point computations play a role in many applications and have thus been considered before in the
context of program verification. Notable are efforts in abstract interpretation, where Blanchet et al. [2], Chen et al. [8] and Jeannet and Miné [21] have developed abstract domains for proving the absence of runtime errors, which are sound wrt. floating-point arithmetic. Jourdan et al. [22] have also formalized some of these abstract domains in Coq. Note, however, that these domains do not quantify the difference between a real-valued and the finite-precision semantics and can only show the absence of runtime errors.

Moscato et al. [30] have built a formalization and implementation of affine arithmetic for computation of real-valued ranges in the PVS system. Our focus was not to compute the tightest possible bounds but on modularity and reusability of the framework. Nevertheless, we may be able to use some of their ideas when extending our framework to affine arithmetic.

Coq has also been used in proving entire programs correct, also wrt. numerical uncertainties such as roundoff errors. For example, Boldo et al. [3] prove the solution of a wave equation functionally correct in Coq using the Frama-C platform [16] for generating verification conditions. However, in these efforts much of the work is still manual. Our current development can be seen as complementary as it could potentially provide automation for the verification of roundoff error bounds.

The CompCert compiler also supports floating-point computations [4]. But the correctness proof only reasons about semantic preservation for floating-point computations and does not reason about the roundoff error with respect to the real-valued execution.

In the context of automated verification within SMT-solvers, Rümmer and Wahl [33] have proposed an SMT-lib theory for floating-point arithmetic. Decision procedures for this theory mostly rely on a bit-precise encoding and in general need approximation techniques to deal with the inherent prohibitive complexity, see for instance [18]. Again, these techniques do not estimate roundoff errors wrt. a real-valued semantics, in part because this require a combination of theories which is an open problem at present.

Real Arithmetic and Floating-point Formalizations Formalizations of floating-point arithmetic exist in HOL-Light [20], in Coq in the Flocq library [7], as well as in Isabelle [27] and HOL4 [15]. All of these are able to closely model the hardware specification of floating points, for instance, they provide a representation which can handle different rounding modes and expresses floating point values using a mantissa and an exponent. We found using these formalizations in Coq and HOL4 more complex than was necessary for our purpose so that we relied on the simpler abstraction from Equation 1.

To the best of our knowledge, there exist two frameworks which extend Coq with formalisations of real arithmetic: the Coquelicot library from Boldo et al. [6] and the C-CoRN library [23]. Since we made the sound simplification of using rationals for our computations, it was not necessary to use either Coq formalisation of the real numbers in our development. Boldo et al. [5] provide a broader comparison of theorem provers in the context of real numbers, e.g. their implementation, inclusion of certain operations and support for proof automation.
Harrison [14] has formally verified a floating point implementation of the exponential function inside HOL-Light. The analysis is detailed and specific to this particular function. In contrast, our work aims to provide a fully automated verified analysis for arbitrary real-valued expressions, but at a higher level of abstraction.

7 Conclusion

We have shown our modular, reusable and easily extendable approach to certificate checking for error bound analysis. Our checker is fully-automated and does require neither user interaction, nor expert knowledge. We are the first to use the CakeML extractor to extract a verified binary from HOL4 code and have shown that we achieve significant performance improvements when using the binary.

References

1. Anand, A., Appel, A., Morrisett, G., Paraskevopoulou, Z., Pollack, R., Belanger, O.S., Sozeau, M., Weaver, M.: CertiCoq: A verified compiler for Coq. In: Coq for Programming Languages (CoqPL) (2017)
2. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A Static Analyzer for Large Safety-Critical Software. In: PLDI (2003)
3. Boldo, S., Clément, F., Filliâtre, J.C., Mayero, M., Melquiond, G., Weis, P.: Wave Equation Numerical Resolution: A Comprehensive Mechanized Proof of a C Program. Journal of Automated Reasoning 50(4), 423–456 (2013)
4. Boldo, S., Jourdan, J.H., Leroy, X., Melquiond, G.: Verified Compilation of Floating-Point Computations. Journal of Automated Reasoning 54(2), 135–163 (2015)
5. Boldo, S., Lelay, C., Melquiond, G.: Formalization of Real Analysis: A Survey of Proof Assistants and Libraries. Mathematical Structures in Computer Science pp. 1–38 (2014)
6. Boldo, S., Lelay, C., Melquiond, G.: Coquelicot: A User-Friendly Library of Real Analysis for Coq. Mathematics in Computer Science 9(1), 41–62 (2015)
7. Boldo, S., Melquiond, G.: Flocoq: A Unified Library for Proving Floating-Point Algorithms in Coq. In: ARITH (2011)
8. Chen, L., Miné, A., Cousot, P.: A Sound Floating-Point Polyhedra Abstract Domain. In: APLAS (2008)
9. Darulova, E.: Rosa - The real compiler. https://github.com/malyzajko/rosa (2015)
10. Darulova, E., Kuncak, V.: Sound Compilation of Reals. In: POPL (2014)
11. Darulova, E., Kuncak, V.: Towards a Compiler for Reals. TOPLAS 39(2) (2017)
12. Daumas, M., Melquiond, G.: Certification of Bounds on Expressions Involving Rounded Operators. ACM Trans. Math. Softw. 37(1), 2:1–2:20 (2010)
13. De Dinechin, F., Lauter, C.Q., Melquiond, G.: Assisted Verification of Elementary Functions using Gappa. In: SAC (2006)
14. de Figueiredo, L.H., Stolfi, J.: Affine Arithmetic: Concepts and Applications. Numerical Algorithms 37(1-4) (2004)
15. Fox, A., Harrison, J., Akbarpour, B.: A Formal Model of IEEE Floating Point Arithmetic. HOL4 Theorem Prover Library (Apr 2017), https://github.com/HOL-Theorem-PROVER/HOL/tree/master/src/floating-point
16. The Frama-C platform for static analysis of C programs. [https://frama-c.com/](https://frama-c.com/) (2008)
17. Goubault, E., Putot, S.: Static Analysis of Finite Precision Computations. In: VMCAI (2011)
18. Haller, L., Griggio, A., Brain, M., Kroening, D.: Deciding Floating-Point Logic with Systematic Abstraction. In: FMCAD (2012)
19. Harrison, J.: Floating Point Verification in HOL Light: The Exponential Function. Form. Methods Syst. Des. 16(3) (2000)
20. Jacobsen, C., Solovyev, A., Gopalakrishnan, G.: A Parameterized Floating-Point Formalization in HOL Light. Electronic Notes in Theoretical Computer Science 317, 101–107 (2015)
21. Jeannet, B., Miné, A.: Apron: A Library of Numerical Abstract Domains for Static Analysis. In: CAV (2009)
22. Jourdan, J.H., Laporte, V., Blazy, S., Leroy, X., Pichardie, D.: A Formally-Verified C Static Analyzer. In: POPL (2015)
23. Krebbers, R., Spitters, B.: Computer Certified Efficient Exact Reals in Coq. In: Intelligent Computer Mathematics, vol. 6824, pp. 90–106. Springer (2011)
24. Leroy, X.: Formal Verification of a Realistic Compiler. Communications of the ACM 52(7) (2009)
25. Letouzey, P.: A New Extraction for Coq. In: TYPES (2002)
26. Linderman, M.D., Ho, M., Dill, D.L., Meng, T.H., Nolan, G.P.: Towards program optimization through automated analysis of numerical precision. In: CGO (2010)
27. Magron, V., Constantinides, G.A., Donaldson, A.F.: Certified Roundoff Error Bounds Using Semidefinite Programming. CoRR abs/1507.03331 (2015)
28. Mariano M. Moscato and Titolo, Laura and Dutle, Aaron and Munoz, César A: A static analysis framework for the estimation of verified floating-point round-off errors. Tech. rep. (sep 2016)
29. Moore, R.: Interval Analysis. Prentice-Hall (1966)
30. Moscato, M.M., Muñoz, C.A., Smith, A.P.: Affine Arithmetic and Applications to Real-Number Proving. In: ITP (2015)
31. Myreen, M.O., Owens, S.: Proof-Producing Synthesis of ML from Higher-Order Logic. In: ICFP (2012)
32. Ramananandro, T., Mountcastle, P., Meister, B., Lethin, R.: A Unified Coq Framework for Verifying C Programs with Floating-Point Computations. In: CPP (2016)
33. Rümmer, P., Wahl, T.: An SMT-LIB Theory of Binary Floating-Point Arithmetic. In: SMT at FLoC (2010)
34. Schneider, S., Smolka, G., Hack, S.: A Linear First-Order Functional Intermediate Language for Verified Compilers. In: ITP (2015)
35. Solovyev, A., Jacobsen, C., Rakamaric, Z., Gopalakrishnan, G.: Rigorous Estimation of Floating-Point Round-off Errors with Symbolic Taylor Expansions. In: FM (2015)
36. Tan, Y.K., Myreen, M.O., Kumar, R., Fox, A., Owens, S., Norrish, M.: A New Verified Compiler Backend for CakeML. In: ICFP (2016)
37. Yu, L.: A Formal Model of IEEE Floating Point Arithmetic. Archive of Formal Proofs (Jul 2013). [http://isa-afp.org/entries/IEEE_Floating_Point.shtml](http://isa-afp.org/entries/IEEE_Floating_Point.shtml)