On Automatically Proving the Correctness of `math.h` Implementations

WONYEOL LEE*, Stanford University, USA
RAHUL SHARMA, Microsoft Research, India
ALEX AIKEN, Stanford University, USA

Industry standard implementations of `math.h` claim (often without formal proof) tight bounds on floating-point errors. We demonstrate a novel static analysis that proves these bounds and verifies the correctness of these implementations. Our key insight is a reduction of this verification task to a set of mathematical optimization problems that can be solved by off-the-shelf computer algebra systems. We use this analysis to prove the correctness of implementations in Intel's math library automatically. Prior to this work, these implementations could only be verified with significant manual effort.

CCS Concepts: • **Software and its engineering** → **Formal software verification**; **Correctness**; • **Mathematics of computing** → *Mathematical software*;

Additional Key Words and Phrases: Floating-point, verification, transcendental functions, rounding error, correctness

ACM Reference Format:
Wonyeol Lee, Rahul Sharma, and Alex Aiken. 2018. On Automatically Proving the Correctness of `math.h` Implementations. *Proc. ACM Program. Lang.* 2, POPL, Article 47 (January 2018), 32 pages. https://doi.org/10.1145/3158135

1 INTRODUCTION

Industry standard math libraries, such as Intel's implementation of `math.h`, have very strict correctness requirements. In particular, Intel guarantees that the maximum *precision loss*, *i.e.*, the difference between the computed floating-point value and the actual mathematical result, is very small. However, to the best of our knowledge, this claim is not backed by formal proofs. Establishing the correctness of these implementations is non-trivial: the error bounds are tight (see below), floating-point operations have rounding errors that are non-trivial to reason about, and these high performance libraries are full of undocumented code optimization tricks. We describe a novel automatic verification technique capable of establishing the correctness of these implementations.

For example, consider the `sin` function of `math.h`. Since it is a transcendental, for most floating-point inputs, $sin\, x$ is an irrational number inexpressible as a 64-bit double precision floating-point number. Most standard libraries guarantee that the maximum precision loss is strictly below one *ulp*, *i.e.*, if the exact mathematical result $sin\, x$ is in the interval (d_1, d_2) where d_1 and d_2 are two consecutive floating-point numbers, then the computed result is generally either d_1 or d_2. This

[*] Author did part of the work as a research intern at Microsoft Research Bangalore, India.

Authors' addresses: Wonyeol Lee, Computer Science, Stanford University, USA, wonyeol@cs.stanford.edu; Rahul Sharma, Microsoft Research, India, rahsha@microsoft.com; Alex Aiken, Computer Science, Stanford University, USA, aiken@cs.stanford.edu.

2475-1421/2018/1-ART47
https://doi.org/10.1145/3158135

requirement is difficult to meet because of floating-point rounding errors. Consider, for example, what happens if we implement x^4 using `x * x * x * x`. For some inputs the precision loss of this implementation is more than one ulp.

An algorithm for computing $sin\,x$ was verified to be correct, *i.e.*, meeting the one ulp bound, for any $x \in [-2^{63}, 2^{63}]$ by Harrison using the proof assistant HOL Light [Harrison 2000b]. Constructing such machine-checkable proofs requires a Herculean effort and Harrison remarks that each proof can take weeks to months of manual effort [Harrison 2000a]. In our own recent work [Lee et al. 2016], we proved an error bound of 9 ulps for Intel's `sin` implementation over the input interval $[-\frac{\pi}{2}, \frac{\pi}{2}]$ automatically, *i.e.*, in most cases there can be at most 9 floating-point numbers between the computed and the mathematically exact result for any input in $[-\frac{\pi}{2}, \frac{\pi}{2}]$. In this paper, we focus on automatically proving much tighter error bounds. We describe an analysis that is fully automatic and, for example, proves that the maximum precision loss of `sin` is below one ulp over the input interval $[-\pi, \pi]$.

The main source of imprecision in previous work stems from modeling every floating-point operation as having a rounding error about which worst-case assumptions must be made. However, floating-point operations do not always introduce rounding errors. In particular, there are several *exactness* results that describe conditions under which floating-point operations are exact, *i.e.*, the floating-point operation gives the mathematical result. For example, although $2^{100} - 1 = 2^{100}$ (due to rounding errors), $0.5 - 0.25$ is exactly equal to 0.25 in floating-point arithmetic. An important example of such an exactness result is Sterbenz's theorem [Sterbenz 1973], which says that when two floating-point numbers are close to each other then their subtraction is exact. Our approach to improving the provable error bounds is to identify floating-point computations that are exact and thus avoid introducing unneeded potential rounding errors into the modeling of those computations. Our main technical contribution is in reducing the problem of checking whether an exactness result applies to a set of mathematical optimization problems that can be solved soundly and automatically by off-the-shelf computer algebra systems. For example, our analysis checks the closeness conditions in Sterbenz's theorem by solving four optimization problems.

We apply this analysis to the benchmarks of [Lee et al. 2016; Schkufza et al. 2014], *i.e.*, Intel's `sin`, `tan`, and `log`. For `log`, we prove that for all valid inputs the precision loss is below one ulp. We are not aware of any other formal correctness proof of this implementation. Previous to this work, the best known provable error bound for `log` was 10^{14} ulps by [Lee et al. 2016], which says the implementation is provably correct only up to two decimal digits. We note that our proof is computationally intensive and took more than two weeks of computation time on 16 cores (but is also highly parallelizable). Next, we prove the correctness of `sin` for inputs between $-\pi$ and π. Recall that $sin\,x$ is periodic and our results can be extended to all valid inputs at the expense of more computational resources. For `tan`, we proved correctness only for a part of the input interval. In particular, our abstractions lose precision and the inferred bounds, though sound, are imprecise for inputs near $\frac{\pi}{2}$ (§7). The previously known bounds for `tan` were loose (up to several orders of magnitude greater than the bounds we prove) and are sometimes not guaranteed to be sound [Lee et al. 2016].

Our main contributions are as follows:

- We show a novel automatic analysis that systematically uses exactness results about floating-point arithmetic. In particular, the analysis verifies that the result of a floating-point operation is exact by solving several mathematical optimization problems soundly.
- We describe the first technique that automatically proves the correctness of transcendental functions in industry standard math libraries. Prior to this work, these implementations could only be verified with significant manual effort.

- We present the properties of floating-point used in these proofs. Some of these properties are only well-known to floating-point experts, and others are new in the sense that they have not been stated explicitly in the literature.

The rest of the paper is organized as follows. §2 motivates our analysis using an example. §3 discusses the basics of floating-point and rounding errors. §4 and §5 present the two major components of our method: An abstraction of floating-point is described in §4 and proven to be sound; the analysis is described in §5. The analysis uses some well-known results (§5.1, 5.2, 5.3, 5.6) and other results about floating-point that we have proven (§5.4, 5.5) and found useful. §6 mentions some interesting implementation details and §7 evaluates the analysis on a number of functions from `math.h`. Finally, §8 discusses related work and §9 concludes.

2 MOTIVATION

We discuss an example on which standard analyses produce very imprecise bounds and show how the precision can be recovered by applying exactness results. Consider Intel's `log` implementation of the natural logarithm function over the input interval $X = [4095/4096, 1)$. The complete `log` implementation is quite complicated but if we restrict the inputs to the small interval X, it can be significantly simplified. For an input $x \in X$, `log` first computes the following quantity:

$$r(x) = \left(\left((2 \otimes x) \ominus \frac{255}{128}\right) \otimes \frac{1}{2}\right) \oplus \left(\left(\frac{255}{128} \otimes \frac{1}{2}\right) \ominus 1\right) \tag{1}$$

where $\circledast$ denotes the floating-point operation corresponding to the real-valued operation $* \in \{+, -, \times, /\}$. Then `log` returns $v(x) = v_3 \oplus v_2 \oplus v_5 \oplus v_4 \oplus v_1$, where $v_1, \cdots, v_5$ are computed as:

$$\begin{aligned}
v_1 &= (d_1 \otimes n) \oplus t_1 \oplus r \\
v_2 &= (d_1 \otimes n) \oplus t_1 \ominus v_1 \oplus r \\
v_3 &= (d_2 \otimes n) \oplus t_2 \\
v_4 &= [c_2 \oplus (c_3 \otimes r) \oplus (c_4 \otimes (r \otimes r))] \otimes (r \otimes r) \\
v_5 &= [((c_5 \oplus (c_6 \otimes r)) \otimes r) \oplus (c_7 \otimes r \otimes (r \otimes r))] \otimes ((r \otimes r) \otimes (r \otimes r))
\end{aligned}$$

Here every floating-point operation is assumed to be left-associative, r denotes $r(x)$, and the floating-point constants c_i, d_i, t_i, and n are $c_i \approx (-1)^{i+1}/i$ $(i = 2, \cdots, 7)$, $d_1 \approx (log\,2)/16$, $d_2 \approx (log\,2)/16 - d_1$, $t_1 \approx log\,2$, $t_2 \approx log\,2 - t_1$, and $n = -16$, where $log\,x$ is the natural logarithm function.

A standard technique to automatically bound the maximum precision loss of such a floating-point implementation is the well-known $(1 + \epsilon)$-property (§3). The property states that for any mathematical operator $* \in \{+, -, \times, /\}$, the result of a floating-point operation $a \circledast b$ is $(a * b)(1 + \delta)$ for some $|\delta| < 2^{-53}$. By applying the $(1 + \epsilon)$-property to each floating-point operation of $r(x)$, we obtain the following abstraction $\mathcal{A}(x)$ of $r(x)$:

$$\begin{aligned}
\mathcal{A}(x) \triangleq & \left[\left(\left((2x)(1+\delta_0) - \frac{255}{128}\right)(1+\delta_1) \times \frac{1}{2}\right)(1+\delta_2) + \left(\left(\frac{255}{128} \times \frac{1}{2}\right)(1+\delta_3) - 1\right)(1+\delta_4)\right](1+\delta_5) \\
&= (x-1) + \left(x - \frac{255}{256}\right)\delta_1 + \cdots
\end{aligned} \tag{2}$$

where each δ_i ranges over $(-2^{-53}, 2^{-53})$. We call $\mathcal{A}(x)$ an abstraction as it over-approximates $r(x)$, *i.e.*, $\forall x \in X. \exists \delta_0, \cdots, \delta_5.\ r(x) = \mathcal{A}(x)$. Observe that the rounding errors accumulate with each floating-point operation, and the maximum precision loss of the final result $v(x)$ (a polynomial in $r(x)$) is at least as large as the maximum precision loss of $r(x)$. Using the abstraction $\mathcal{A}(x)$ of $r(x)$,

```
1: Let $x = 2^p \times 1.g_2 \cdots g_{53\,(2)}$
2: Compute $s = 1.g_2 \cdots g_{53\,(2)}$ and $s' = 1.g_2 \cdots g_8 0 \cdots 0_{\,(2)}$
3: Compute $s_{inv} = 2^q \times 1.h_2 \cdots h_8 0 \cdots 0_{\,(2)}$ such that $s_{inv} \approx 1/s$
4: Compute $r(x) = (s \ominus s') \otimes s_{inv} \oplus (s' \otimes s_{inv} \ominus 1)$
```

Fig. 1. The computation of $r(x)$ in Intel's implementation `log` of the natural logarithm function.

the maximum relative error of $r(x)$ is bounded by:

$$\max_{x \in X, |\delta_i| < 2^{-53}} \left| \frac{\mathcal{A}(x) - (x-1)}{x-1} \right|$$

Because of the term $(x - \frac{255}{256})\delta_1$ in the abstraction $\mathcal{A}(x)$ of $r(x)$, this error is at least

$$\max_{x \in X} \left| \frac{x - \frac{255}{256}}{x-1} \right| \epsilon \tag{3}$$

Unfortunately the objective function in Eq. 3 is unbounded for $x \in X$, and thus, using this analysis, we are unable to bound the maximum relative error of the result.

A more precise analysis can bound the maximum relative error of $r(x)$. The key insight is that some floating-point operations in Eq. 1 are exact and do not introduce any rounding errors. In particular, the subtraction operations in Eq. 1 are exact according to Sterbenz's theorem: $a \ominus b$ is exact whenever a is within a factor of 2 of b (§5.2). Here, $x \in [4095/4096, 1)$ and hence $\frac{1}{2} \cdot \frac{255}{128} \leqslant 2x \leqslant 2 \cdot \frac{255}{128}$ holds. Moreover, multiplication and division by 2 are also exact (§5.1). Using this information, we can construct a more precise abstraction of $r(x)$. In particular, for an exact operation $a \circledast b$, we have $a \circledast b = a * b$ and we do not need to introduce δ variables. Since all the operations except $\oplus$ in Eq. 1 are exact, we have $r(x) = \big((2x - \frac{255}{128}) \times \frac{1}{2}\big) \oplus \big((\frac{255}{128} \times \frac{1}{2}) - 1\big)$. Therefore, by applying the $(1+\epsilon)$-property only once to the operation $\oplus$, we obtain the following abstraction $\mathcal{A}'(x)$ of $r(x)$ which is more precise than Eq. 2:

$$\mathcal{A}'(x) \triangleq \left[\left(\left(2x - \frac{255}{128} \right) \times \frac{1}{2} \right) + \left(\left(\frac{255}{128} \times \frac{1}{2} \right) - 1 \right) \right] (1 + \delta') = (x-1) + (x-1)\delta'$$

where δ' ranges over $(-2^{-53}, 2^{-53})$. We use this more precise abstraction to find a better bound on the maximum relative error of $r(x)$:

$$\max_{x \in X, |\delta'| < 2^{-53}} \left| \frac{\mathcal{A}'(x) - (x-1)}{x-1} \right| \leqslant 2^{-53} \tag{4}$$

Note that $\mathcal{A}'(x)$ does not contain the term $(x - \frac{255}{256})\delta_1$ unlike $\mathcal{A}(x)$ (Eq. 2), and we do not need to solve the optimization problem of Eq. 3 that has an unbounded objective. In our analysis of `log`, this step is the key to improving the error bound from the previously published bound of 10^{14} ulps to 0.583 ulps (§7).

In general, for any 64-bit double precision floating-point number (or any *double*) $x \geqslant 2^{-1022}$, `log` computes the quantity $r(x)$ as described in Figure 1. The `log` implementation first extracts the exponent p and the 53-bit significand $1.g_2 \cdots g_{53\,(2)}$ of the double x (line 1), and constructs two doubles s and s' that represent the significand of x and the result of masking out the 45 least significant bits of the significand, respectively (line 2). It then computes a double s_{inv} that is close to $1/s$ while having only an 8-bit significand (line 3). Using doubles s, s', and s_{inv}, `log` computes $r(x)$ that approximates $s \times s_{inv} - 1$ (line 4). Note that, if we restrict inputs to $[4095/4096, 1)$, $s = 2 \otimes x$, $s' = 255/128$, $s_{inv} = 1/2$, and line 4 becomes Eq. 1.

By using additional properties of floating-point arithmetic, we can show that all the operations except the addition $\oplus$ in line 4 of Figure 1 are exact for any input $x \geqslant 2^{-1022}$. The operation $\ominus$ in $s \ominus s'$ is exact according to the Sterbenz's theorem, because $s'/2 \leqslant s \leqslant 2s'$ for any $x \geqslant 2^{-1022}$. The multiplication $\otimes$ in $(s \ominus s') \otimes s_{inv}$ is also exact according to the following property: $a \otimes b$ is exact whenever $\sigma(a) + \sigma(b) \leqslant 53$, where $\sigma(d)$ for a double d denotes the number of significand bits of d that are not trailing zeros (§5.4). Note that $\sigma(s \ominus s') \leqslant 45$ because the 8 most significant bits $1, g_2, \cdots, g_8$ of s and s' are cancelled out during the subtraction $s \ominus s'$, and that $\sigma(s_{inv}) \leqslant 8$ by the definition of s_{inv}; thus, we have $\sigma(s \ominus s') + \sigma(s_{inv}) \leqslant 53$ for any $x \geqslant 2^{-1022}$. Similarly, we can show that the two operations in $s' \otimes s_{inv} \ominus 1$ are also exact, using Sterbenz's theorem and the property of $\sigma(\cdot)$.

Based on the above exactness results, we can tightly bound the maximum relative error of $r(x)$ for any $x \geqslant 2^{-1022}$. Since all the operations except $\oplus$ in line 4 of Figure 1 are exact, the following is an abstraction of $r(x)$ for any $x \geqslant 2^{-1022}$ by the $(1+\epsilon)$-property:

$$\mathcal{A}''(x) \triangleq [(s - s') \times s_{inv} + (s' \times s_{inv} - 1)](1 + \delta'') = (s \times s_{inv} - 1) + (s \times s_{inv} - 1)\delta''$$

where δ'' ranges over $(-2^{-53}, 2^{-53})$. Using the abstraction $\mathcal{A}''(x)$ of $r(x)$, the maximum relative error of $r(x)$ for any $x \geqslant 2^{-1022}$ is bounded by

$$\max_{x \geqslant 2^{-1022}, |\delta''| < 2^{-53}} \left| \frac{\mathcal{A}''(x) - (s \times s_{inv} - 1)}{s \times s_{inv} - 1} \right| \leqslant 2^{-53}$$

This analysis generalizes the previous result (Eq. 4) that the maximum relative error of $r(x)$ is bounded by 2^{-53} for $x \in [4095/4096, 1)$ to the larger input interval $x \geqslant 2^{-1022}$. Note that we cannot obtain such tight bounds on the maximum relative error without proving the exactness of floating-point operations.

In the next three sections, we describe an analysis that automatically exploits such non-trivial properties of floating-point arithmetic to tightly bound the maximum precision loss of floating-point implementations. After defining metrics for precision loss, we present all the properties of floating-point used in our analysis. Some of these are well-known to floating-point experts but not to others, and some properties are new in the sense that they have not been stated explicitly in the literature. Our main contribution is a reduction from the problem of automatically applying these properties to mathematical optimization problems. For example, optimization problems are used to check preconditions of the properties we use above (*e.g.*, whether two values are within a factor of 2) and to compute relevant quantities (*e.g.*, $\sigma(\cdot)$).

3 BACKGROUND

A 64-bit double precision floating-point number (or a double) is defined as follows:

Definition 3.1. $x \in \mathbb{R}$ is a *double* if for some $s \in \{0, 1\}$, $g_i \in \{0, 1\}$ $(1 \leqslant i \leqslant 53)$, and $p \in \mathbb{Z}$,

$$x = (-1)^s \times 2^p \times g_1.g_2 \cdots g_{53\ (2)}$$

where either (a) $-1022 \leqslant p \leqslant 1023$ and $g_1 \neq 0$, or (b) $p = -1022$ and $g_1 = 0$ holds. We call s the *sign*, p the *exponent*, and $g_1.g_2 \cdots g_{53\ (2)}$ the *significand* of the double x. We call x a *subnormal* number if (b) holds.

Let $\mathbb{F} \subset \mathbb{R}$ denote the set of all doubles including subnormals. We say $x \in \mathbb{R}$ is in the *subnormal range* if $|x| < 2^{-1022}$. In this paper $\mathbb{F}$ does not include $\pm\infty$ and NaN (not-a-number) which are part of the IEEE 754 standard. These can be introduced by overflows and divide-by-0 errors and we prove their absence for our benchmarks (§6). Furthermore, we do not discuss single-precision floating-point numbers as 32-bit math.h implementations can be verified by exhaustive testing.

To define floating-point operations, we first define a *rounding* $\mathrm{fl} : \mathbb{R} \to \mathbb{F}$ that converts a real number to a double, as follows:

$$\mathrm{fl}(r) \triangleq \arg\min_{d \in \mathbb{F}} |r - d|$$

The ties are broken by choosing the d with 0 in the least significant position. We use the "rounding to nearest even" instead of other rounding modes (*e.g.*, "rounding toward 0") to define $\mathrm{fl}(\cdot)$, since it is the default rounding mode for doubles in the IEEE 754 standard. Our techniques in this paper are easily extended to support other rounding modes by modifying Theorem 3.2. Using the rounding function $\mathrm{fl}(\cdot)$, we define a floating-point operation $\circledast$ as

$$x \circledast y \triangleq \mathrm{fl}(x * y)$$

where $* \in \{+, -, \times, /\}$ and $x, y \in \mathbb{F}$ with $|x * y| \leqslant \max \mathbb{F}$. Here the operation without a circle denotes a real-valued operation, while one with a circle denotes a floating-point operation. Throughout the paper, we assume that each floating-point operation is left-associative.

The rest of this paper relies heavily on the following $(1 + \epsilon)$-property that enables us to model the rounding error of a floating-point operation:

Theorem 3.2. *Let $x, y \in \mathbb{F}$ and $* \in \{+, -, \times, /\}$. Assume $|x * y| \leqslant \max \mathbb{F}$. Then,*

$$x \circledast y = (x * y)(1 + \delta) + \delta'$$

for some $|\delta| < \epsilon$ and $|\delta'| \leqslant \epsilon'$, where $\epsilon \triangleq 2^{-53}$ and $\epsilon' \triangleq 2^{-1075}$ are constants.

The $(1 + \epsilon)$-property states that each floating-point operation can introduce two kinds of errors, a multiplicative error modeled by δ and an additive error modeled by δ', but that each type of error is always bounded by very small constants ϵ and ϵ' respectively, regardless of the operands x and y. Here the constant ϵ is often called the machine epsilon.

To measure the rounding errors, we define three metrics: the absolute error, the relative error, and the ulp (units in last place) error. Let $r \in \mathbb{R}$ be an exact value and $r' \in \mathbb{R}$ be an approximation. The two standard metrics, the absolute/relative error of r' with respect to r, are defined as

$$\mathrm{ErrAbs}(r, r') \triangleq |r - r'|, \quad \mathrm{ErrRel}(r, r') \triangleq \left| \frac{r - r'}{r} \right|$$

To define the ulp error, we introduce the ulp function $\mathrm{ulp}(\cdot)$:

$$\mathrm{ulp}(r) = \begin{cases} 2^{k-52} & \text{for } |r| \in [2^k, 2^{k+1}), \text{ where } k \in [-1022, 1023] \cap \mathbb{Z} \\ 2^{-1074} & \text{for } |r| \in [0, 2^{-1022}) \end{cases}$$

$\mathrm{ulp}(r)$ represents the gap between two adjacent doubles $x, y \in \mathbb{F}$ that surrounds r, *i.e.*, $x \leqslant r < y$ if $r \geqslant 0$, and $x < r \leqslant y$ if $r < 0$. Using the ulp function $\mathrm{ulp}(\cdot)$, the ulp error of r' with respect to r is defined as

$$\mathrm{ErrUlp}(r, r') \triangleq \frac{|r - r'|}{\mathrm{ulp}(r)}$$

For example, the ulp error of a floating-point operation with respect to the corresponding exact operation is always bounded by 0.5 ulps: $\mathrm{ErrUlp}(x * y, x \circledast y) \leqslant 1/2$ for any $x, y \in \mathbb{F}$ with $|x * y| \leqslant \max \mathbb{F}$. Although the absolute/relative errors are widely known, the ulp error is more commonly used than the other two when measuring the rounding error of math libraries. The absolute error can be large even when the result is incorrect only in the least significant bit. The relative error does not suffer from this problem but it is undefined at $r = 0$. On the other hand, the ulp error is proportional to the relative error (Theorem 3.3), is always defined, and hence is preferable. Therefore, this paper focuses on the ulp error of floating-point implementations.

$$\begin{array}{rccl}
\text{expression} & e & ::= & c \mid x \mid e \circledast e \mid \mathit{bit\text{-}mask}(e, B) \\
\text{floating-point constant} & c & \in & \mathbb{F} \\
\text{floating-point operation} & \circledast & \in & \{\oplus, \ominus, \otimes, \oslash\} \\
\text{bit-mask constant} & B & \in & \{1, 2, \cdots, 52\}
\end{array}$$

Fig. 2. The abstract syntax of our core language

The ulp error is closely related to the relative error in the following way:

Theorem 3.3. *Let $r \in [-\max \mathbb{F}, \max \mathbb{F}]$ and $r' \in \mathbb{R}$. Then,*

$$\begin{array}{rl}
\mathrm{ErrUlp}(r, r') \leqslant \mathrm{ErrRel}(r, r') \cdot \dfrac{1}{\epsilon} & \quad \text{if } r \neq 0 \\
\mathrm{ErrRel}(r, r') \leqslant \mathrm{ErrUlp}(r, r') \cdot 2\epsilon & \quad \text{if } |r| \geqslant 2^{-1022}
\end{array}$$

As a corollary of the theorem, we have $\mathrm{ErrUlp}(r, r') \in [\mathrm{ErrRel}(r, r')/(2\epsilon), \mathrm{ErrRel}(r, r')/\epsilon]$ for any $|r| \in [2^{-1022}, \max \mathbb{F}]$ and $r' \in \mathbb{R}$.

We remark that slightly different definitions of the ulp function have been proposed [Muller 2005]: for example, [Goldberg 1991], [Harrison 1999], and [Kahan 2004]. However, these definitions coincide on almost all doubles: Harrison's and Kahan's definition are identical on $\mathbb{F}$, and Goldberg's definition is identical to the other two on $\mathbb{F} \backslash \{2^k : k \in \mathbb{Z}\}$. In this paper, we use Goldberg's definition for $\mathrm{ulp}(\cdot)$, as it is the most widely known.

4 ABSTRACTION

In this section, we describe the syntax of the floating-point expressions we consider, the abstraction that over-approximates behaviors of an expression, and the abstraction process.

Figure 2 defines the abstract syntax of the core language for our formal development. An elementary expression e can be a 64-bit floating-point constant c, a 64-bit floating-point input x, an application of a floating-point operation $\circledast$ to subexpressions, or an application of the bit-mask operation $\mathit{bit\text{-}mask}(\cdot, \cdot)$ to a subexpression. The bit-mask operation $\mathit{bit\text{-}mask}(e, B)$ masks out B least significant bits of e's significand ($1 \leqslant B \leqslant 52$). For brevity, we describe our techniques for elementary or uni-variate expressions, but they can easily be extended to expressions with multiple inputs. Let $X \subset \mathbb{R}$ denote the input interval of an expression, *i.e.*, the floating-point input $x \in X$. And let $\mathcal{E}(e) : X \cap \mathbb{F} \to \mathbb{F}$ denote the concrete semantics of the language, *i.e.*, the result of evaluating e over an input $x \in X \cap \mathbb{F}$ is given by $\mathcal{E}(e)(x)$.

In the remaining parts of this paper, we use the following abstraction to over-approximate the behaviors of an expression e:

$$\mathcal{A}_{\vec{\delta}}(x) = a(x) + \sum_i b_i(x)\delta_i \qquad \text{where } |\delta_i| \leqslant \Delta_i$$

The abstraction $\mathcal{A}_{\vec{\delta}} : X \to \mathbb{R}$ is a function of $x \in X$ and $\vec{\delta} = (\delta_1, \cdots, \delta_n) \in \mathbb{R}^n$, where each δ_i represents a rounding error. $\mathcal{A}_{\vec{\delta}}(x)$ consists of two parts: $a(x)$ and the sum over $b_i(x)\delta_i$. The first part $a(x)$ represents the exact result of e on an input x, which is obtained by replacing every floating-point operation in e with its corresponding real-valued operation and by ignoring every bit-mask operation. In particular, for our benchmarks $a(x)$ is non-linear, *i.e.*, composed of polynomials and rational functions in x. In the second part $b_i(x)\delta_i$ represents an error term that arises from the rounding error of one or more floating-point/bit-mask operation(s). Here the variable $\delta_i \in [-\Delta_i, \Delta_i]$, where $\Delta_i \in \mathbb{R}_{\geqslant 0}$ is a constant. This abstraction is similar to the one described in [Solovyev et al. 2015].

We next define *sound* abstractions of expressions as follows:

Definition 4.1. $\mathcal{A}_{\vec{\delta}}(x)$ is a *sound* abstraction of e if $\forall x \in X \cap \mathbb{F}.\ \mathcal{E}(e)(x) \in \{\mathcal{A}_{\vec{\delta}}(x) : |\delta_i| \leqslant \Delta_i\}$.

The abstractions form a partial order: $\mathcal{A}_{\vec{\delta}}(x) \sqsubseteq \mathcal{A}'_{\vec{\delta}'}(x)$ if $\forall x \in X \cap \mathbb{F}.\ \{\mathcal{A}_{\vec{\delta}}(x) : |\delta_i| \leqslant \Delta_i\} \subseteq \{\mathcal{A}'_{\vec{\delta}'}(x) : |\delta'_i| \leqslant \Delta'_i\}$. The abstractions higher up in the order are more over-approximate. The goal of this section is to construct a sound abstraction of a given expression.

Before describing how to construct such an abstraction, we define the four elementary operations on abstractions that over-approximate their real-valued counterparts. They are defined as:

$$\begin{aligned}
\mathcal{A}_{\vec{\delta}}(x) \boxplus \mathcal{A}'_{\vec{\delta}}(x) &\triangleq \mathcal{A}_{\vec{\delta}}(x) + \mathcal{A}'_{\vec{\delta}}(x) \\
\mathcal{A}_{\vec{\delta}}(x) \boxminus \mathcal{A}'_{\vec{\delta}}(x) &\triangleq \mathcal{A}_{\vec{\delta}}(x) - \mathcal{A}'_{\vec{\delta}}(x) \\
\mathcal{A}_{\vec{\delta}}(x) \boxtimes \mathcal{A}'_{\vec{\delta}}(x) &\triangleq \mathit{linearize}(\mathcal{A}_{\vec{\delta}}(x) \times \mathcal{A}'_{\vec{\delta}}(x)) \\
\mathcal{A}_{\vec{\delta}}(x) \boxslash \mathcal{A}'_{\vec{\delta}}(x) &\triangleq \mathcal{A}_{\vec{\delta}}(x) \boxtimes \mathit{inv}(\mathcal{A}'_{\vec{\delta}}(x))
\end{aligned}$$

Observe that $\boxplus$ and $\boxminus$ are defined simply as $+$ and $-$. On the other hand, the real-valued multiplication of two abstractions may not be an abstraction because of $\delta_i\delta_j$ terms. To soundly remove such quadratic δ terms, we introduce a new operation $\mathit{linearize}(\cdot)$:

$$\mathit{linearize}\left(a(x) + \sum_i b_i(x)\delta_i + \sum_{i,j} b_{i,j}(x)\delta_i\delta_j\right) \triangleq a(x) + \sum_i b_i(x)\delta_i + \sum_{i,j} b_{i,j}(x)\delta'_{i,j} \qquad \text{where } |\delta'_{i,j}| \leqslant \Delta_i\Delta_j$$

Here $\delta'_{i,j}$ is a fresh variable ranging over $[-\Delta_i\Delta_j, \Delta_i\Delta_j]$. Using the new operation, $\boxtimes$ is defined as the application of $\mathit{linearize}(\cdot)$ to the real-valued multiplication of two abstractions. Note that this abstraction is more precise than the ones considered in prior work that either bound all the quadratic error terms by one ulp [Lee et al. 2016] or bound the coefficients of the quadratic terms by constants obtained via interval analysis [Solovyev et al. 2015]. These constants can lead to imprecise ulp error bounds when $a(x) \approx 0$ and we give an example at the end of this section.

To define $\boxslash$, it is enough to define the operation $\mathit{inv}(\mathcal{A}_{\vec{\delta}}(x))$ that over-approximates the inverse of $\mathcal{A}_{\vec{\delta}}(x)$. We first over-approximate $\mathcal{A}_{\vec{\delta}}(x) = a(x) + \sum_i b_i(x)\delta_i$ to obtain a simpler abstraction $a(x) + a(x)\delta'$ that has only one δ term, and then over-approximate the inverse of the simplified abstraction, $\frac{1}{a(x)+a(x)\delta'} = \frac{1}{a(x)} \cdot \frac{1}{1+\delta'}$, to obtain the final abstraction. This is formalized as:

$$\mathit{inv}\left(a(x) + \sum_i b_i(x)\delta_i\right) \triangleq \frac{1}{a(x)} + \frac{1}{a(x)}\delta'' \qquad \text{where } |\delta''| \leqslant \frac{\Delta'}{1-\Delta'} \quad (\text{assumes } \Delta' < 1)$$

Here δ'' is a fresh variable and Δ' is obtained by solving the following optimization problem:

$$\Delta' = \sum_i \max_{x \in X} \left| \frac{b_i(x)}{a(x)} \right| \cdot \Delta_i$$

Throughout the paper, for any function $f(x)$ and $g(x)$, the value of $|f(x_0)/g(x_0)|$ at x_0 with $g(x_0) = 0$ is defined as 0 if $f(x_0) = 0$, and ∞ if $f(x_0) \neq 0$. Note that Δ' bounds the relative error of $\mathcal{A}_{\vec{\delta}}(x)$ with respect to its exact term $a(x)$, *i.e.*, $\mathrm{ErrRel}(a(x), \mathcal{A}_{\vec{\delta}}(x)) \leqslant \Delta'$ for all $x \in X$ and $|\delta_i| \leqslant \Delta_i$. Technically, the above definition of $\mathit{inv}(\cdot)$ assumes $\Delta' < 1$, but it can be extended to work even when $\Delta' \geqslant 1$. However, the condition $\Delta' < 1$ holds for all applications of $\mathit{inv}(\cdot)$ in our benchmarks.

Next, we show that the four operations $\boxast$ defined above over-approximate their real-valued counterparts:

$$\frac{e \in dom(\mathcal{K}) \qquad \mathcal{K}(e) = (\mathcal{A}_{\vec{\delta}}, \_)}{(\mathcal{K}, e) \rhd (\mathcal{K}, \mathcal{A}_{\vec{\delta}})}\ \textsc{Load}$$

$$\frac{}{(\mathcal{K}, c) \rhd (\mathcal{K}[c \mapsto (c, \texttt{false})], c)}\ \text{R1} \qquad \frac{}{(\mathcal{K}, x) \rhd (\mathcal{K}[x \mapsto (x, \texttt{false})], x)}\ \text{R2}$$

$$\frac{\begin{array}{c}(\mathcal{K}, e_1) \rhd (\mathcal{K}_1, \mathcal{A}_{1,\vec{\delta}}) \\ (\mathcal{K}_1, e_2) \rhd (\mathcal{K}_2, \mathcal{A}_{2,\vec{\delta}})\end{array} \qquad * \in \{+, -, \times, /\}}{(\mathcal{K}, e_1 \circledast e_2) \rhd (\mathcal{K}', \mathcal{A}'_{\vec{\delta}}),\ \text{where} \begin{cases} \delta' = fresh(\epsilon),\ \delta'' = fresh(\epsilon') \\ \mathcal{A}'_{\vec{\delta}} = compress((\mathcal{A}_{1,\vec{\delta}} \boxast \mathcal{A}_{2,\vec{\delta}}) \boxtimes (1 + \delta') \boxplus \delta'') \\ \mathcal{K}' = \mathcal{K}_2[e_1 \circledast e_2 \mapsto (\mathcal{A}'_{\vec{\delta}}, \texttt{false})] \end{cases}}\ \text{R3}$$

$$\frac{(\mathcal{K}, bit\text{-}mask(e_1, B)) \rhd (\mathcal{K}_1, \mathcal{A}_{1,\vec{\delta}})}{(\mathcal{K}, bit\text{-}mask(e_1, B)) \rhd (\mathcal{K}', \mathcal{A}'_{\vec{\delta}}),\ \text{where} \begin{cases} \delta' = fresh(2^{-52+B}),\ \delta'' = fresh(2^{-1074+B}) \\ \mathcal{A}'_{\vec{\delta}} = compress(\mathcal{A}_{1,\vec{\delta}} \boxtimes (1 + \delta') \boxplus \delta'') \\ \mathcal{K}' = \mathcal{K}_1[bit\text{-}mask(e_1, B) \mapsto (\mathcal{A}'_{\vec{\delta}}, \texttt{false})] \end{cases}}\ \text{R4}$$

Fig. 3. Rules for constructing an abstraction of an expression

Lemma 4.2. *$\mathcal{A}_{\vec{\delta}}(x) * \mathcal{A}'_{\vec{\delta}}(x) \sqsubseteq \mathcal{A}_{\vec{\delta}}(x) \boxast \mathcal{A}'_{\vec{\delta}}(x)$ for any $* \in \{+, -, \times, /\}$.*

Proof. We sketch the argument that *linearize*$(\cdot)$ and *inv*$(\cdot)$ over-approximate their arguments. The main observation is that $\{\delta_i\delta_j : |\delta_i| \leqslant \Delta_i, |\delta_j| \leqslant \Delta_j\} \subseteq [-\Delta_i\Delta_j, \Delta_i\Delta_j]$, and $\{1/(1+\delta') - 1 : |\delta'| \leqslant \Delta'\} \subseteq [-\Delta'/(1+\Delta'), \Delta'/(1-\Delta')] \subseteq [-\Delta'/(1-\Delta'), \Delta'/(1-\Delta')]$ if $\Delta' < 1$. The proof of Lemma 4.5 shows $a(x) + a(x)\delta'$ with $|\delta'| \leqslant \Delta'$ over-approximates $\mathcal{A}_{\vec{\delta}}(x)$. □

The rules for constructing a sound abstraction of e are given in Figure 3. In the rules, $\mathcal{K}$ (and $\mathcal{K}'$) represents a *cache*, a mapping from expressions to tuples of size 2, which stores already computed analysis results. A cache is defined to be *sound* if for any $e \in dom(\mathcal{K})$ with $\mathcal{K}(e) = (\mathcal{A}_{\vec{\delta}}, b)$, $\mathcal{A}_{\vec{\delta}}$ is a sound abstraction of e and $b = \texttt{true}$ implies that e is not atomic and the last operation of e is exact. The judgment $(\mathcal{K}, e) \rhd (\mathcal{K}', \mathcal{A}_{\vec{\delta}})$ denotes that given a sound cache $\mathcal{K}$, our analysis of e constructs a provably sound abstraction $\mathcal{A}_{\vec{\delta}}$ of e and a sound cache $\mathcal{K}'$ that stores both previous and new analysis results. The function *fresh*(Δ) returns a fresh variable δ with the constraint $|\delta| \leqslant \Delta$. The operations $\mathcal{A}_{\vec{\delta}}(x) \boxtimes (1 + \delta')$ and $\mathcal{A}_{\vec{\delta}}(x) \boxplus \delta'$ are defined as a special case of $\mathcal{A}_{\vec{\delta}}(x) \boxtimes \mathcal{A}'_{\vec{\delta}}(x)$ and $\mathcal{A}_{\vec{\delta}}(x) \boxplus \mathcal{A}'_{\vec{\delta}}(x)$:

$$\begin{aligned} \mathcal{A}_{\vec{\delta}}(x) \boxtimes (1 + \delta') &\triangleq \mathcal{A}_{\vec{\delta}}(x) \boxtimes \mathcal{A}'_{\vec{\delta}}(x) \qquad && \text{where } \mathcal{A}'_{\vec{\delta}}(x) = 1 + 1 \cdot \delta' \\ \mathcal{A}_{\vec{\delta}}(x) \boxplus \delta' &\triangleq \mathcal{A}_{\vec{\delta}}(x) \boxplus \mathcal{A}'_{\vec{\delta}}(x) \qquad && \text{where } \mathcal{A}'_{\vec{\delta}}(x) = 0 + 1 \cdot \delta' \end{aligned}$$

For now, let us ignore the operation *compress*$(\cdot)$. The rule Load is applied first whenever applicable; other rules are applied only when the rule Load is not applicable (*i.e.*, an analysis result of an expression is not found in the current cache). The rule R3 is based on the $(1 + \epsilon)$-property[1], and the rule R4 is based on the following lemma about abstracting the bit-mask operation:

[1] For $* \in \{+, -\}$, we can soundly remove the term $\boxplus\delta''$ from the rule R3 by Theorem 5.8 in §5.5 (see R14 in Figure 8).

Lemma 4.3. *Given $x \in \mathbb{F}$ and $B \in \{1, 2, \cdots, 52\}$, let $y \in \mathbb{F}$ be the result of masking out B least significant bits of x's significand. Then for some $|\delta| < 2^{-52+B}$ and $|\delta'| \leqslant 2^{-1074+B}$,*

$$y = x(1+\delta) + \delta'$$

The rules in Figure 3 (with $compress(\cdot)$ erased) can be used to construct a sound abstraction, but the final abstraction can potentially have a huge number of δ variables. Specifically, for $\mathcal{A}_{i,\vec{\delta}}(x)$ with k_i δ variables ($i = 1, 2$), $\mathcal{A}_{1,\vec{\delta}}(x) \boxtimes \mathcal{A}_{2,\vec{\delta}}(x)$ has $(k_1+1)(k_2+1)-1$ δ variables. Using this fact, we can prove that the abstraction of e can potentially have more than 2^k δ variables, where k is the number of floating-point/bit-mask operations in e. This property holds because for each floating-point/bit-mask operation, we need to apply either the rule R3 or R4 both of which introduce new $(1+\delta')$ terms and perform the operation $(\cdots) \boxtimes (1+\delta')$. Thus, constructing an abstraction based on the rules in Figure 3 without $compress(\cdot)$ is intractable.

To address this issue, we re-define the operation $\mathcal{A}_{\vec{\delta}}(x) \boxtimes (1+\delta')$ as:

$$\mathcal{A}_{\vec{\delta}}(x) \boxtimes (1+\delta') \triangleq a(x) + a(x)\delta' + \sum_i b_i(x)\delta'_i \qquad \text{where } |\delta'_i| \leqslant \Delta_i(1+\Delta')$$

Here a given variable δ' ranges over $[-\Delta', \Delta']$ and δ'_i is a fresh variable. Note that under the new definition, $\mathcal{A}_{\vec{\delta}}(x) \boxtimes (1+\delta')$ now has $k+1$ (instead of $2k+1$) δ variables for any $\mathcal{A}_{\vec{\delta}}(x)$ with k δ variables, and it still over-approximates $\mathcal{A}_{\vec{\delta}}(x) \times (1+\delta')$:

Lemma 4.4. $\mathcal{A}_{\vec{\delta}}(x) \times (1+\delta') \sqsubseteq \mathcal{A}_{\vec{\delta}}(x) \boxtimes (1+\delta')$

Proof. For $\{\delta_i(1+\delta') : |\delta_i| \leqslant \Delta_i, |\delta'| \leqslant \Delta'\} \subseteq [-\Delta_i(1+\Delta'), \Delta_i(1+\Delta')]$. □

This revised definition of $\mathcal{A}_{\vec{\delta}}(x) \boxtimes (1+\delta')$ resolves the issue to some extent, but not completely because the number of δ variables is still exponential in the number of multiplications in e. To this end, we define a new operation $compress(\mathcal{A}_{\vec{\delta}}(x))$ that significantly reduces the number of δ variables in $\mathcal{A}_{\vec{\delta}}(x)$, as follows:

$$compress(\mathcal{A}_{\vec{\delta}}(x)) \triangleq a(x) + a(x)\delta' + \sum_{i \notin S} b_i(x)\delta_i \qquad \text{where } |\delta'| \leqslant \sum_{i \in S} \gamma_i$$

Here δ' is a fresh variable, and $\gamma_i \in \mathbb{R}_{\geqslant 0} \cup \{\infty\}$ and the set S are computed as

$$\gamma_i = \max_{x \in X} \left| \frac{b_i(x)}{a(x)} \right| \cdot \Delta_i \qquad \text{and} \qquad S = \left\{ i : \frac{\gamma_i}{\epsilon} \leqslant \tau \right\} \tag{5}$$

The operation $compress(\mathcal{A}_{\vec{\delta}}(x))$ can remove some δ variables in $\mathcal{A}_{\vec{\delta}}(x)$, and how aggressively it removes the δ variables is determined by a user-given constant $\tau \in \mathbb{R}_{\geqslant 0}$ ($\tau = 10$ in our experiments): if τ is big, $compress(\mathcal{A}_{\vec{\delta}}(x))$ would have a small number of δ variables but can be too over-approximate, and if τ is small, $compress(\mathcal{A}_{\vec{\delta}}(x))$ would capture most behaviors of $\mathcal{A}_{\vec{\delta}}(x)$ precisely but can have many δ variables. Note that γ_i is computed by solving the optimization problem (of a single variable) that also appears in the computation of $inv(\cdot)$. The quantity γ_i represents the contribution of the i-th error term $b_i(x)\delta_i$ to the overall relative error of $\mathcal{A}_{\vec{\delta}}(x)$ with respect to $a(x)$; thus, from Theorem 3.3, γ_i/ϵ represents how much the error term $b_i(x)\delta_i$ contributes to the overall ulp error of $\mathcal{A}_{\vec{\delta}}(x)$. Thus, the set S represents the indices of the error terms whose contribution to the overall ulp error is small enough, *i.e.*, $\leqslant \tau$ ulps. The $compress(\cdot)$ procedure merges all the error terms of $\mathcal{A}_{\vec{\delta}}(x)$ that have such small contribution to the total ulp error, into a single error term $a(x)\delta'$, and leaves all the other error terms of $\mathcal{A}_{\vec{\delta}}(x)$ as is. Conceptually, we can set $\tau = \infty$ and merge all δ terms into a single term. But for some cases, *e.g.*, Theorem 5.2 in §5.3, this merging can lead to very imprecise abstractions.

Like all the previous operations on abstractions, $compress(\cdot)$ over-approximates its argument:

Lemma 4.5. $\mathcal{A}_{\vec{\delta}}(x) \sqsubseteq compress(\mathcal{A}_{\vec{\delta}}(x))$

Proof. First, we show that for any i with $\gamma_i < \infty$ we have $b_i(x)\delta_i \sqsubseteq a(x)\delta'_i$, where $|\delta'_i| \leqslant \gamma_i$. Consider any i with $\gamma_i < \infty$ and any $x \in X$. If $a(x) \neq 0$,

$$|b_i(x)\delta_i| = |a(x)| \cdot \left|\frac{b_i(x)}{a(x)}\delta_i\right| \leqslant |a(x)| \cdot \left|\frac{b_i(x)}{a(x)}\right| \Delta_i \leqslant |a(x)| \cdot \gamma_i = |a(x)\gamma_i|$$

which implies $\{b_i(x)\delta_i : |\delta_i| \leqslant \Delta_i\} \subseteq \{a(x)\delta'_i : |\delta'_i| \leqslant \gamma_i\}$. If $a(x) = 0$, $\gamma_i < \infty$ implies $b_i(x) = 0$, so we have $\{b_i(x)\delta_i : |\delta_i| \leqslant \Delta_i\} = \{0\} = \{a(x)\delta'_i : |\delta'_i| \leqslant \gamma_i\}$.

Next, it is easy to see that $\sum_{i \in S} a(x)\delta'_i \sqsubseteq a(x)\delta'$, where $|\delta'| \leqslant \sum_{i \in S} \gamma_i$. Combining the two facts implies $\sum_{i \in S} b_i(x)\delta_i \sqsubseteq a(x)\delta'$. Hence $\sum_i b_i(x)\delta_i = \sum_{i \in S} b_i(x)\delta_i + \sum_{i \notin S} b_i(x)\delta_i \sqsubseteq a(x)\delta' + \sum_{i \notin S} b_i(x)\delta_i$. □

We remark that the previous work on similar abstractions, [Goubault and Putot 2005; Solovyev et al. 2015], does not use the $compress(\cdot)$ operation.

Let us revisit the rules in Figure 3. The rules R3 and R4 use the re-defined operation $(\cdots)\boxtimes(1+\delta')$ and the newly defined operation $compress(\cdot)$, to reduce the number of δ variables in the abstractions of $e_1 \circledast e_2$ and $bit\text{-}mask(e_1, B)$. Using these two operations, the rules can be applied to expressions of practical sizes. We note that these two operations will be re-defined again in §5.3. Finally, we establish the soundness of the rules:

Theorem 4.6. *If* $(\cdot, e) \rhd (\mathcal{K}', \mathcal{A}_{\vec{\delta}})$ *then* $\mathcal{A}_{\vec{\delta}}$ *is a sound abstraction of* e.

Proof. We generalize the above statement as: if $(\mathcal{K}, e) \rhd (\mathcal{K}', \mathcal{A}_{\vec{\delta}})$ and $\mathcal{K}$ is a sound cache, then $\mathcal{K}'$ is a sound cache and $\mathcal{A}_{\vec{\delta}}$ is a sound abstraction of e. We can prove this by induction on the derivation tree of $(\mathcal{K}, e) \rhd (\mathcal{K}', \mathcal{A}_{\vec{\delta}})$ using Theorem 3.2 and Lemmas 4.2, 4.3, 4.4, and 4.5. □

We conclude this section by demonstrating that the abstraction used in [Solovyev et al. 2015] is not sufficient to prove tight ulp error bounds. For example, consider Intel's `sin` implementation of the sine function over the input interval $X = [2^{-252}, \frac{\pi}{64}]$. Since `sin` computes $x - \frac{1}{6}x^3 + \cdots$ for an input $x \in X$, applying the $(1+\epsilon)$-property produces an abstraction of `sin` that contains the error term $-\frac{1}{6}x^3\delta_1\delta_2\delta_3$, where $|\delta_i| < \epsilon$ $(i = 1, 2, 3)$. In Solovyev et al.'s abstraction, the cubic error term is over-approximated by a first-order error term $C\delta'$, where $|\delta'| < \epsilon$ and $C = \epsilon^2 \max\{|-\frac{1}{6}x^3| : x \in X\} \approx 2 \times 10^{-37}$. Hence with Solovyev et al.'s abstraction, a bound on the maximum ulp error of `sin` over X (with respect to $\sin x$) is at least $\frac{1}{\epsilon}\max\{|C\delta'/\sin x| : x \in X, |\delta'| < \epsilon\} \approx 2 \times 10^{39}$ ulps, which is too loose. The culprit is that Solovyev et al.'s abstraction has constant coefficients for higher-order error terms and these terms become significant for inputs near zero.

5 EXPLOITING EXACTNESS PROPERTIES

Although the rules in Figure 3 can be used to construct a sound abstraction of an expression e, the resulting abstraction can over-approximate the behaviors of e too imprecisely and fail to prove a tight error bound of e. Consider a part of the implementation of `log` discussed in §2: $e = (2 \otimes x) \ominus \frac{255}{128}$ with $X = [\frac{4095}{4096}, 1)$. As already explained, the operations $\otimes$ and $\ominus$ in e are exact, *i.e.*, introduces no rounding errors due to the exactness of multiplication by 2 and Sterbenz's theorem (Theorem 5.1), so $\mathcal{A}_{\vec{\delta}}(x) = 2x - \frac{255}{128}$ is a sound abstraction of e. However, the rules in Figure 3 generate $\mathcal{A}'_{\vec{\delta}}(x) = (2x - \frac{255}{128}) + (2x - \frac{255}{128})\delta_1 + \cdots$ as an abstraction of e by simply applying the $(1+\epsilon)$-property to the $\otimes$ and $\ominus$ operation. The proof then uses $\mathcal{A}'_{\vec{\delta}}$ as an abstraction of e, instead of using the more precise abstraction $\mathcal{A}_{\vec{\delta}}$. This imprecision leads to the imprecise error bound of 10^{14} ulps [Lee et al. 2016] for `log` over X.

$$\frac{\begin{array}{l}(\mathcal{K}, e_1) \triangleright (\mathcal{K}_1, \mathcal{A}_{1,\vec{\delta}}) \\ (\mathcal{K}_1, e_2) \triangleright (\mathcal{K}_2, 0) \qquad\qquad * \in \{+, -\}\end{array}}{(\mathcal{K}, e_1 \circledast e_2) \triangleright (\mathcal{K}_2[e_1 \circledast e_2 \mapsto (\mathcal{A}_{1,\vec{\delta}}, \texttt{true})], \mathcal{A}_{1,\vec{\delta}})}\ \text{R5}$$

$$\frac{\begin{array}{l}(\mathcal{K}, e_1) \triangleright (\mathcal{K}_1, \mathcal{A}_{1,\vec{\delta}}) \\ (\mathcal{K}_1, e_2) \triangleright (\mathcal{K}_2, 2^n)\ (n \in \mathbb{Z}) \qquad * \in \{\times, /\}\end{array}}{(\mathcal{K}, e_1 \circledast e_2) \triangleright (\mathcal{K}_2[e_1 \circledast e_2 \mapsto (\mathcal{A}_{1,\vec{\delta}} * 2^n, \texttt{true})], \mathcal{A}_{1,\vec{\delta}} * 2^n)}\ \text{R6}$$

$$\frac{\begin{array}{l}(\mathcal{K}, e_1) \triangleright (\mathcal{K}_1, c_1) \\ (\mathcal{K}_1, e_2) \triangleright (\mathcal{K}_2, c_2) \qquad * \in \{+, -, \times, /\}\end{array}}{(\mathcal{K}, e_1 \circledast e_2) \triangleright (\mathcal{K}_2[e_1 \circledast e_2 \mapsto (c', c' == c_1 * c_2)], c'),\ \text{where } c' = c_1 \circledast c_2}\ \text{R7}$$

Fig. 4. Rules for simple exact operations

To prove a tighter error bound, we construct a more precise abstraction by avoiding the application of the $(1+\epsilon)$-property whenever possible while maintaining soundness (Theorem 4.6). For each floating-point operation $e_1 \circledast e_2$, we first determine whether the operation $\circledast$ is exact or not using some properties of floating-point arithmetic. If the particular operation $\circledast$ is exact, we simply use $\mathcal{A}_{1,\vec{\delta}} \boxast \mathcal{A}_{2,\vec{\delta}}$ as a sound abstraction of $e_1 \circledast e_2$, where $\mathcal{A}_{i,\vec{\delta}}$ is a sound abstraction of e_i $(i = 1, 2)$. In contrast, the $(1+\epsilon)$-property instead yields the less precise abstraction $(\mathcal{A}_{1,\vec{\delta}} \boxast \mathcal{A}_{2,\vec{\delta}}) \boxtimes (1 + \delta') \boxplus \delta''$. The key to this approach is automatically determining whether a given floating-point operation is exact, *i.e.*, produces no rounding errors.

Some of the properties of floating-point that we use are well-known to floating-point experts (§5.1, 5.2, 5.3, 5.6) and some are new (*i.e.*, haven't appeared explicitly in the literature) to the best of our knowledge (§5.4, 5.5). We remark that it was challenging for us to rediscover these properties and infer how to use them in automatic proofs of error bounds for practical floating-point implementations.

5.1 Simple Exact Operations

We start with the simplest situation where a floating-point addition/subtraction or a floating-point multiplication/division is exact: for any $x, y \in \mathbb{F}$ with $y = 2^n$ for some $n \in \mathbb{Z}$, we have $x \circledast 0 = x$ if $* \in \{+, -\}$, and $x \circledast y = x * y$ if $* \in \{\times, /\}$. In other words, floating-point addition by 0 is always exact, and floating-point multiplication by an integer power of 2 is always exact because multiplying x by a power of 2 changes only the exponent of x, not its significand.[2]

Figure 4 presents the rules based on the above property. The rule R5 considers the addition/subtraction case and the rule R6 considers the multiplication/division case[3]; their commutative counterparts are omitted. The rule R7 considers the case where the evaluation result of e_1 and e_2 are exactly known as c_1 and c_2. In such a case, though the floating-point operation $\circledast$ in $e_1 \circledast e_2$ may not be exact, we can know the exact evaluation result of the operation, $c_1 \circledast c_2$, by partial evaluation. Note that all the rules in the figure do not use the $(1+\epsilon)$-property.

[2] Technically, $x \otimes 2^n = x \times 2^n$ may not hold if $x \times 2^n$ is very small (*e.g.*, $x = 2^{-1074}$ and $n = -1$) since the exponent of a double cannot be smaller than -1022.

[3] Strictly speaking, the rule R6 is unsound according to Footnote 2. For a sound version of the rule R6, refer to the rule R6′ (§A.2) which uses quantities $\sigma(e)$ and $\mu(e)$ introduced in §5.4 and §5.5.

5.2 Sterbenz's Theorem

The next situation where a floating-point addition/subtraction is exact is described in Sterbenz's theorem [Sterbenz 1973]:

Theorem 5.1. *Let $x, y \in \mathbb{F}$ with $x, y \geqslant 0$. Then*

$$\frac{x}{2} \leqslant y \leqslant 2x \quad \Longrightarrow \quad x \ominus y = x - y$$

The theorem says that the floating-point subtraction of $x \geqslant 0$ and $y \geqslant 0$ is exact whenever y is within a factor of two of x.

Typical examples that make use of Sterbenz's theorem are from range reduction steps that reduce the computation of $f(x)$ to the computation of $g(r)$ such that the range of r is much smaller than that of x. A range reduction step used to compute $log\, x$ has been discussed in §2 and at the beginning of this section. Another common range reduction is:

$$n = \text{round}(K_{inv} \otimes x), \quad r = x \ominus (K \otimes n)$$

where n is an integer, and $K_{inv}, K \in \mathbb{F}_{>0}$ have a relationship that $K_{inv} \approx 1/K$. For example, if $K = \text{fl}(\pi)$ then this range reduction can reduce the computation of $sin\, x$ to $sin\, r$ for $r \in [-\frac{\pi}{2}, \frac{\pi}{2}]$. This range reduction relies on Sterbenz's theorem to guarantee that the operation $\ominus$ in the computation of r is exact.

Before explaining how to exploit Sterbenz's theorem in our framework, we point out that Theorem 5.1 considers only the case $x, y \geqslant 0$. To cover the case $x, y \leqslant 0$ as well, we extend the theorem in the following way: for any $x, y \in \mathbb{F}$, if they satisfy

$$\frac{x}{2} \leqslant y \leqslant 2x \qquad \text{or} \qquad 2x \leqslant y \leqslant \frac{x}{2} \tag{6}$$

then $x \ominus y = x - y$. From now on, we refer to this extended theorem (instead of Theorem 5.1) as Sterbenz's theorem.

Next, we derive optimization problems, based on Sterbenz's theorem, that can check whether an operation $\ominus$ between two expressions e_1 and e_2 is exact. As e_1 and e_2 are functions of x, we would like to check if the operation $\mathcal{E}(e_1)(x) \ominus \mathcal{E}(e_2)(x)$ is exact for all $x \in X \cap \mathbb{F}$. According to Sterbenz's theorem (Eq. 6), the operation is exact for all $x \in X \cap \mathbb{F}$ if

$$\forall x \in X \cap \mathbb{F}.\ \left(\frac{1}{2}\mathcal{E}(e_1)(x) \leqslant \mathcal{E}(e_2)(x) \leqslant 2\mathcal{E}(e_1)(x)\right) \vee \left(2\mathcal{E}(e_1)(x) \leqslant \mathcal{E}(e_2)(x) \leqslant \frac{1}{2}\mathcal{E}(e_1)(x)\right) \tag{7}$$

However, we do not know $\mathcal{E}(e_i)(x)$ statically; rather we can construct its abstraction as described in §4. Let $\mathcal{A}_{i,\vec{\delta}}$ be a sound abstraction of e_i ($i = 1, 2$). From the definition of a sound abstraction, for any $x \in X \cap \mathbb{F}$, we have $\mathcal{E}(e_1)(x) = \mathcal{A}_{1,\vec{\delta}}(x)$ and $\mathcal{E}(e_2)(x) = \mathcal{A}_{2,\vec{\delta}}(x)$ for some $\vec{\delta} \in \vec{\Delta}$, where $\vec{\Delta} = [-\Delta_1, \Delta_1] \times \cdots \times [-\Delta_n, \Delta_n]$. Using $\mathcal{A}_{1,\vec{\delta}}$ and $\mathcal{A}_{2,\vec{\delta}}$, we strengthen Eq. 7 to Eq. 8:

$$\forall x \in X. \forall \vec{\delta} \in \vec{\Delta}.\ \left[\left(\frac{1}{2}\mathcal{A}_{1,\vec{\delta}}(x) \leqslant \mathcal{A}_{2,\vec{\delta}}(x) \leqslant 2\mathcal{A}_{1,\vec{\delta}}(x)\right) \vee \left(2\mathcal{A}_{1,\vec{\delta}}(x) \leqslant \mathcal{A}_{2,\vec{\delta}}(x) \leqslant \frac{1}{2}\mathcal{A}_{1,\vec{\delta}}(x)\right)\right] \tag{8}$$

Eq. 8 is stronger than Eq. 7 in two aspects: it is quantified over x that ranges over X (instead of over $X \cap \mathbb{F}$), and additionally over $\vec{\delta}$. The first change is motivated by the fact that checking inequalities (and solving optimization problems) over $X \subset \mathbb{R}$ is easier than over the discrete set $X \cap \mathbb{F}$. The second change is necessary since we do not know statically which $\vec{\delta} \in \vec{\Delta}$ would satisfy $\mathcal{E}(e_1)(x) = \mathcal{A}_{1,\vec{\delta}}(x)$ and $\mathcal{E}(e_2)(x) = \mathcal{A}_{2,\vec{\delta}}(x)$ for each x. Although Eq. 8 is easier to handle than

$$\frac{\begin{array}{ll}(\mathcal{K}, e_1) \rhd (\mathcal{K}_1, \mathcal{A}_{1,\vec{\delta}}) & \min_{x,\vec{\delta}}(\mathcal{A}_{2,\vec{\delta}} - \frac{1}{2}\mathcal{A}_{1,\vec{\delta}}) \geqslant 0 \\ (\mathcal{K}_1, e_2) \rhd (\mathcal{K}_2, \mathcal{A}_{2,\vec{\delta}}) & \max_{x,\vec{\delta}}(\mathcal{A}_{2,\vec{\delta}} - 2\mathcal{A}_{1,\vec{\delta}}) \leqslant 0\end{array}}{(\mathcal{K}, e_1 \ominus e_2) \rhd (\mathcal{K}', \mathcal{A}'_{\vec{\delta}}),\ \text{where} \begin{cases} \mathcal{A}'_{\vec{\delta}} = compress(\mathcal{A}_{1,\vec{\delta}} \boxminus \mathcal{A}_{2,\vec{\delta}}) \\ \mathcal{K}' = \mathcal{K}_2[e_1 \ominus e_2 \mapsto (\mathcal{A}'_{\vec{\delta}}, \mathtt{true})]\end{cases}} \ \text{R8}$$

$$\frac{\begin{array}{ll}(\mathcal{K}, e_1) \rhd (\mathcal{K}_1, \mathcal{A}_{1,\vec{\delta}}) & \min_{x,\vec{\delta}}(\mathcal{A}_{2,\vec{\delta}} - 2\mathcal{A}_{1,\vec{\delta}}) \geqslant 0 \\ (\mathcal{K}_1, e_2) \rhd (\mathcal{K}_2, \mathcal{A}_{2,\vec{\delta}}) & \max_{x,\vec{\delta}}(\mathcal{A}_{2,\vec{\delta}} - \frac{1}{2}\mathcal{A}_{1,\vec{\delta}}) \leqslant 0\end{array}}{(\mathcal{K}, e_1 \ominus e_2) \rhd (\mathcal{K}', \mathcal{A}'_{\vec{\delta}}),\ \text{where} \begin{cases} \mathcal{A}'_{\vec{\delta}} = compress(\mathcal{A}_{1,\vec{\delta}} \boxminus \mathcal{A}_{2,\vec{\delta}}) \\ \mathcal{K}' = \mathcal{K}_2[e_1 \ominus e_2 \mapsto (\mathcal{A}'_{\vec{\delta}}, \mathtt{true})]\end{cases}} \ \text{R9}$$

Fig. 5. Rules for applying Sterbenz's theorem

Eq. 7, transforming it directly into optimization problems is still difficult because of the $\vee$ within the quantifiers. We strengthen it further to obtain Eq. 9:

$$\left(\forall x.\forall\vec{\delta}.\ \frac{1}{2}\mathcal{A}_{1,\vec{\delta}}(x) \leqslant \mathcal{A}_{2,\vec{\delta}}(x) \leqslant 2\mathcal{A}_{1,\vec{\delta}}(x)\right) \vee \left(\forall x.\forall\vec{\delta}.\ 2\mathcal{A}_{1,\vec{\delta}}(x) \leqslant \mathcal{A}_{2,\vec{\delta}}(x) \leqslant \frac{1}{2}\mathcal{A}_{1,\vec{\delta}}(x)\right) \tag{9}$$

where x ranges over X and $\vec{\delta}$ over $\vec{\Delta}$. The left clause of Eq. 9 is logically equivalent to

$$\left(\min_{x\in X,\vec{\delta}\in\vec{\Delta}}(\mathcal{A}_{2,\vec{\delta}} - \frac{1}{2}\mathcal{A}_{1,\vec{\delta}}) \geqslant 0\right) \wedge \left(\max_{x\in X,\vec{\delta}\in\vec{\Delta}}(\mathcal{A}_{2,\vec{\delta}} - 2\mathcal{A}_{1,\vec{\delta}}) \leqslant 0\right) \tag{10}$$

involving two optimization problems that are sufficient to ensure $e_1 \ominus e_2$ is exact.

The rules shown in Figure 5 are based on the above derivation. The rule R8 does not apply the $(1+\epsilon)$-property if Eq. 10 holds. The rule R9 is based on the counterpart of Eq. 10 derived from the right clause of Eq. 9. Note that in Figure 5 the rules for $e_1 \oplus e_2$ are omitted: they can be obtained from the rules for $e_1 \ominus e_2$ by negating $\mathcal{A}_{2,\vec{\delta}}$ as $x \oplus y = x \ominus (-y)$ for any x and y.

5.3 Dekker's Theorem

The next property of floating-point arithmetic that we use to construct a more precise abstraction is Dekker's theorem [Dekker 1971]. The theorem suggests a way to compute the rounding error, $(x \oplus y) - (x + y)$, of an operation $x \oplus y$. It is well-known that the rounding error $r = (x \oplus y) - (x + y)$ is in fact a double for any $x, y \in \mathbb{F}$, and Dekker's theorem provides a way to recover r using only floating-point operations on x and y:

Theorem 5.2. *Let* $x, y \in \mathbb{F}$ *with* $|x + y| \leqslant \max\mathbb{F}$ *and* $r = x \oplus y \ominus x \ominus y$. *Then*

$$|x| \geqslant |y| \quad \Longrightarrow \quad r = (x \oplus y) - (x + y)$$

The double $r = x \oplus y \ominus x \ominus y$ in the theorem represents the rounding error of $x \oplus y$.

Let us start with the rule R11 of Figure 6 that constructs a tighter abstraction of an expression $e_r = e_1 \oplus e_2 \ominus e_1 \ominus e_2$ based on Dekker's theorem. The rule first checks whether Dekker's theorem is applicable to the expression e_r, *i.e.*, whether the condition $P_1 \triangleq \forall x \in X \cap \mathbb{F}.\ |\mathcal{E}(e_1)(x)| \geqslant |\mathcal{E}(e_2)(x)|$ is satisfied. However, P_1 cannot be checked statically and the rule actually checks a stronger condition $P_2 \triangleq \min_{x,\vec{\delta}} |\mathcal{A}_{1,\vec{\delta}}(x)| \geqslant \max_{x,\vec{\delta}} |\mathcal{A}_{2,\vec{\delta}}(x)|$ by solving optimization problems on a sound abstraction $\mathcal{A}_{i,\vec{\delta}}$ of e_i $(i = 1, 2)$. The derivation of P_2 from P_1 is similar to the derivation of Eq. 10 from Eq. 7 in §5.2. Once the rule successfully checks that P_2 is true, it constructs a sound abstraction

$$\frac{\begin{array}{c}(\mathcal{K}, e_1) \rhd (\mathcal{K}_1, \mathcal{A}_{1,\vec{\delta}}) \\ (\mathcal{K}_1, e_2) \rhd (\mathcal{K}_2, \mathcal{A}_{2,\vec{\delta}}) \qquad \mathsf{hasDekker}(e_1 \oplus e_2)\end{array}}{(\mathcal{K}, e_1 \oplus e_2) \rhd (\mathcal{K}', \mathcal{A}'_{\vec{\delta}}),\ \text{where} \begin{cases} \delta' = \mathit{fresh}(\epsilon, \texttt{true}) \\ \mathcal{A}'_{\vec{\delta}} = \mathit{compress}((\mathcal{A}_{1,\vec{\delta}} \boxplus \mathcal{A}_{2,\vec{\delta}}) \boxtimes (1 + \delta')) \\ \mathcal{K}' = \mathcal{K}_2[e_1 \oplus e_2 \mapsto (\mathcal{A}'_{\vec{\delta}}, \texttt{false}\langle\delta'\rangle)] \end{cases}} \ \text{R10}$$

$$\frac{\begin{array}{ccc} & \mathcal{K}_1(e_1) = (\mathcal{A}_{1,\vec{\delta}}, \_) & \mathcal{K}_1(e_1 \oplus e_2) = (\_, \texttt{false}\langle\delta'\rangle) \\ (\mathcal{K}, e_1 \oplus e_2) \rhd (\mathcal{K}_1, \_) & \mathcal{K}_1(e_2) = (\mathcal{A}_{2,\vec{\delta}}, \_) & \min_{x,\vec{\delta}} |\mathcal{A}_{1,\vec{\delta}}| \geqslant \max_{x,\vec{\delta}} |\mathcal{A}_{2,\vec{\delta}}| \end{array}}{(\mathcal{K}, e_1 \oplus e_2 \ominus e_1 \ominus e_2) \rhd (\mathcal{K}', \mathcal{A}'_{\vec{\delta}}),\ \text{where} \begin{cases} \mathcal{A}'_{\vec{\delta}} = \mathit{compress}((\mathcal{A}_{1,\vec{\delta}} \boxplus \mathcal{A}_{2,\vec{\delta}}) \boxtimes \delta') \\ \mathcal{K}' = \mathcal{K}_1[e_1 \oplus e_2 \ominus e_1 \ominus e_2 \mapsto (\mathcal{A}'_{\vec{\delta}}, \texttt{false})] \end{cases}} \ \text{R11}$$

$$\frac{(\mathcal{K}, e_1 \oplus e_2) \rhd (\mathcal{K}_1, \_) \qquad \mathcal{K}_1(e_1 \oplus e_2) = (\_, \texttt{true})}{(\mathcal{K}, e_1 \oplus e_2 \ominus e_1 \ominus e_2) \rhd (\mathcal{K}_1[e_1 \oplus e_2 \ominus e_1 \ominus e_2 \mapsto (0, \texttt{true})], 0)} \ \text{R12}$$

Fig. 6. Rules for applying Dekker's theorem

of e_r not by applying the $(1+\epsilon)$-property, but by applying Dekker's theorem which says e_r is the rounding error of $e_1 \oplus e_2$. Note that the rule requires a new operation $\mathcal{A}_{\vec{\delta}}(x) \boxtimes \delta'$:

$$\mathcal{A}_{\vec{\delta}}(x) \boxtimes \delta' \triangleq \mathcal{A}_{\vec{\delta}}(x) \boxtimes \mathcal{A}'_{\vec{\delta}}(x) \qquad \text{where } \mathcal{A}'_{\vec{\delta}}(x) = 0 + 1 \cdot \delta'$$

Although the rule R11 constructs a tighter abstraction of $e_r = e_1 \oplus e_2 \ominus e_1 \ominus e_2$ than the rule R3, it does not fully capture the essence of Dekker's theorem: the possibility of the rounding error of $x \oplus y$ being exactly cancelled out by $x \oplus y \ominus x \ominus y$. Such cancellation may not occur even with the rule R11 because some δ variables can be replaced with or merged into fresh ones by $\mathcal{A}_{\vec{\delta}} \boxtimes (1+\delta')$ and $\mathit{compress}(\cdot)$.

We re-define the operations $\mathcal{A}_{\vec{\delta}} \boxtimes (1+\delta')$ and $\mathit{compress}(\cdot)$ and introduce the rule R10 to ensure that δ variables related to Dekker's theorem are preserved (*i.e.*, not replaced with or merged into fresh variables). As a first step, each variable δ_i is associated with the predicate $\mathsf{preserve}(\delta_i)$ which indicates whether δ_i should be preserved: $\mathsf{preserve}(\delta_i) = \texttt{true}$ denotes that δ_i should be preserved, whereas $\mathsf{preserve}(\delta_i) = \texttt{false}$ denotes that merging δ_i is allowed. Using $\mathsf{preserve}(\cdot)$, we re-define the following operations on abstractions:

$$\mathcal{A}_{\vec{\delta}}(x) \boxtimes (1+\delta') \triangleq a(x) + a(x)\delta' + \sum_{i \in R} b_i(x)\delta'_i + \sum_{i \notin R} (b_i(x)\delta_i + b_i(x)\delta''_i) \quad \text{where } \begin{array}{l} |\delta'_i| \leqslant \Delta_i(1+\Delta'), \\ |\delta''_i| \leqslant \Delta_i \Delta' \end{array}$$

$$\mathit{compress}(\mathcal{A}_{\vec{\delta}}(x)) \triangleq a(x) + a(x)\delta' + \sum_{i \notin R \cap S} b_i(x)\delta_i \qquad \text{where } |\delta'| \leqslant \sum_{i \in R \cap S} \gamma_i$$

Here δ', δ'_i, and δ''_i are fresh variables, S and γ_i are defined as before (Eq. 5), and $R = \{i : \mathsf{preserve}(\delta_i) = \texttt{false}\}$. The re-defined operations preserve any δ_i with $\mathsf{preserve}(\delta_i) = \texttt{true}$. Note that the previous definition of $\mathcal{A}_{\vec{\delta}}(x) \boxtimes (1+\delta')$ and $\mathit{compress}(\cdot)$ is obtained by setting $\mathsf{preserve}(\delta_i) = \texttt{false}$ for all i.

In the rule R10 of Figure 6, the predicate $\mathsf{hasDekker}(e_1 \oplus e_2)$ denotes that a given expression contains $e_r = e_1 \oplus e_2 \ominus e_1 \ominus e_2$ as a subexpression, and the value $\texttt{false}\langle\delta'\rangle$ denotes that the operation $\oplus$ in $e_1 \oplus e_2$ may not be exact and the rounding error from this $\oplus$ operation is modeled by the variable δ'. The function $\mathit{fresh}(\Delta, b)$ returns a fresh variable δ with the constraint $|\delta| \leqslant \Delta$ and $\mathsf{preserve}(\delta) = b$; the previous function $\mathit{fresh}(\Delta)$ now denotes $\mathit{fresh}(\Delta, \Delta > \epsilon)$, which implies that by default only those δ variables from bit-mask operations are preserved. The antecedent of the rule R10 indicates that Dekker's theorem can possibly be applied to $e_1 \oplus e_2$ and e_r. In this case the rule sets $\mathsf{preserve}(\delta')$ to be $\mathtt{true}$, where δ' denotes the rounding error of $e_1 \oplus e_2$, and prevents δ' from being removed. Note that the rule R11 uses this δ' in an abstraction of e_r to make the cancellation possible. The rule R10 does not add an absolute error term δ'' ($|\delta''| \leqslant \epsilon'$) in its consequent by the refined $(1+\epsilon)$-property (Theorem 5.8 in §5.5).

To illustrate how the rules R10 and R11 are applied, consider an expression $e = e_1 \ominus (e_2 \oplus e_3)$ over $X = [1, 2]$, where $e_1 = x \oplus 1$, $e_2 = x \oplus 1 \ominus x \ominus 1$, and $e_3 = 0.01 \otimes x \otimes x$. The expression e accurately computes $1 + x - 0.01x^2$ by subtracting e_2 (which evaluates exactly to the rounding error of e_1 by Dekker's theorem) from e_1. Analyzing e with the rules R10 and R11 produces the following derivation tree:

$$\dfrac{\dfrac{\begin{array}{c}\cdots\\ \mathsf{hasDekker}(e_1)\end{array}}{(\cdot, e_1) \rhd (\mathcal{K}_1, \mathcal{A}_{1,\vec{\delta}})}\,\text{R10} \qquad \dfrac{\dfrac{\dfrac{x \oplus 1 \in dom(\mathcal{K}_1)}{(\mathcal{K}_1, x \oplus 1) \rhd (\mathcal{K}_1, \_)}\,\text{L}{\small\text{OAD}} \quad \begin{array}{r} \mathcal{K}_1(x) = (x, \_) \\ \mathcal{K}_1(1) = (1, \_) \\ \mathcal{K}_1(x \oplus 1) = (\_, \mathtt{false}\langle\delta_1\rangle) \\ \min_{x,\vec{\delta}} |x| \geqslant \max_{x,\vec{\delta}} |1| \end{array}}{(\mathcal{K}_1, e_2) \rhd (\mathcal{K}_2, \mathcal{A}_{2,\vec{\delta}})}\,\text{R11} \qquad (\mathcal{K}_2, e_3) \rhd \cdots}{(\mathcal{K}_1, e_2 \oplus e_3) \rhd \cdots}\,\text{R3}}{(\cdot, e_1 \ominus (e_2 \oplus e_3)) \rhd (\cdots, \mathcal{A}'_{\vec{\delta}})}\,\text{R3}$$

Here $\mathcal{A}_{1,\vec{\delta}}(x) = (x+1)+(x+1)\delta_1, \mathcal{K}_1 = [1 \mapsto (1, \mathtt{false}), x \mapsto (x, \mathtt{false}), e_1 \mapsto (\mathcal{A}_{1,\vec{\delta}}, \mathtt{false}\langle\delta_1\rangle)]$, $\mathcal{A}_{2,\vec{\delta}}(x) = (x+1)\delta_1, \mathcal{K}_2 = \mathcal{K}_1[e_2 \mapsto (\mathcal{A}_{2,\vec{\delta}}, \mathtt{false})]$, and $\mathcal{A}'_{\vec{\delta}}(x) = (1 + x - 0.01x^2) + (1 + x - 0.01x^2)\delta'$, where $|\delta_1| \leqslant \epsilon$ and $|\delta'| \leqslant 1.041\epsilon$. The above derivation tree states that an abstraction of e_1 and of e_2 are constructed as $\mathcal{A}_{1,\vec{\delta}}$ and $\mathcal{A}_{2,\vec{\delta}}$. Note that the abstraction $\mathcal{A}_{2,\vec{\delta}}$ of e_2 is $(x+1)\delta_1$, the error term of $\mathcal{A}_{1,\vec{\delta}}$ which models the rounding error of $x \oplus 1$. Hence the final abstraction $\mathcal{A}'_{\vec{\delta}}$ of e does not contain the error term $(x+1)\delta_1$ due to its cancellation, which is what we desired.

The rule R12 in Figure 6, based on Lemma 5.3, deals with the specific case when $e_1 \oplus e_2$ is exact. The lemma says that if $x \oplus y$ is exact then $x \oplus y \ominus x \ominus y = 0$ regardless of the ordering between x and y.

Lemma 5.3. *Let $x, y \in \mathbb{F}$ and $r = x \oplus y \ominus x \ominus y$. Then*

$$x \oplus y = x + y \qquad \Longrightarrow \qquad r = 0$$

Note that there are several variants of Theorem 5.2 and Lemma 5.3, and Figure 6 omits the corresponding variants of the rules R10, R11, and R12 for brevity. For instance, one variant of Theorem 5.2 is: $|y| \geqslant |x|$ implies $r = -((x \ominus y) - (x - y))$ where $r = x \ominus (x \ominus y \oplus y)$. The rules based on each such variant of Theorem 5.2 and Lemma 5.3 can be designed analogously to the rules R10, R11, and R12 by focusing on different expressions (*e.g.*, $x \ominus y$ and $x \ominus (x \ominus y \oplus y)$) and extending the definition of $\mathsf{hasDekker}(\cdot)$ accordingly.

5.4 Nonzero Significand Bits

The next floating-point property we exploit is based on $\sigma(d)$, the number of the significand bits of $d \in \mathbb{F}$ that are not trailing zeros. To formally define $\sigma(\cdot)$ over a subset of $\mathbb{R}$, we define the exponent function $\mathsf{expnt}(\cdot)$ as:

$$\mathsf{expnt}(r) \triangleq \begin{cases} k & \text{for } |r| \in [2^k, 2^{k+1}) \quad \text{where } k \in [-1022, 1023] \cap \mathbb{Z} \\ -1022 & \text{for } |r| \in [0, 2^{-1022}) \end{cases}$$

Using $\text{expnt}(\cdot)$, we define the function $\sigma : [-\max\mathbb{F}, \max\mathbb{F}] \to \mathbb{Z}_{\geqslant 0} \cup \{\infty\}$ as follows: for $r \neq 0$,

$$\sigma(r) \triangleq \begin{cases} \max\{i \in \mathbb{Z}_{\geqslant 1} : f_i \neq 0\} & \text{if defined} \\ \infty & \text{otherwise} \end{cases}$$

where $f_1.f_2f_3\cdots_{(2)}$ is the binary representation of $r/2^{\text{expnt}(r)}$ ($f_i \in \{0,1\}$ for all i), and $\sigma(0) \triangleq 0$. For example, $\sigma(5/8) = 3$ since $\text{expnt}(5/8) = -1$ and $5/8 = 2^{-1} \times 1.01_{(2)}$, and $\sigma(1/5) = \infty$ since $1/5 = 2^{-3} \times 1.10011001\cdots_{(2)}$.

The following theorem uses $\sigma(\cdot)$ to determine if a floating-point operation is exact:

Theorem 5.4. *Let $x, y \in \mathbb{F}$ with $|x * y| \leqslant \max\mathbb{F}$ where $* \in \{+, -, \times, /\}$. Then*

$$\sigma(x * y) \leqslant 53 \qquad \Longrightarrow \qquad x \circledast y = x * y$$

The theorem follows directly from the observation: $\sigma(r) \leqslant 53$ iff $r \in \mathbb{F}$ for any $r \in [-\max\mathbb{F}, \max\mathbb{F}]$. It holds because every double has a 53-bit significand and every real number with a 53-bit significand is representable as a double.

To make use of this theorem, we must compute $\sigma(x * y)$. The following two lemmas can be used to bound $\sigma(x * y)$, given $\sigma(x)$ and $\sigma(y)$ (or their upper bounds). First, Lemma 5.5 handles multiplication:

Lemma 5.5. *Let $x, y \in \mathbb{F}$ with $|xy| \leqslant \max\mathbb{F}$. Assume $x, y \neq 0$. Then*

$$\sigma(x \times y) \leqslant \sigma(x) + \sigma(y) + k$$

where $k = \min\{n \in \mathbb{Z}_{\geqslant 0} : |xy| \geqslant 2^{-1022-n}\}$.

The intuition is that multiplying two integers of n and m digits produces an integer of at most $n + m$ digits. The integer k in the lemma is necessary to consider the case when $|xy| < 2^{-1022}$.

Second, Lemma 5.6 handles addition and subtraction:

Lemma 5.6. *Let $x, y \in \mathbb{F}$ with $|x + y| \leqslant \max\mathbb{F}$. Assume $x, y > 0$ and $\text{expnt}(x) \geqslant \text{expnt}(y)$. Then*

$$\sigma(x + y) \leqslant \max\{\sigma(x), \sigma(y) + \Delta e\} + k$$
$$\sigma(x - y) \leqslant \max\{\max\{\sigma(x), \sigma(y) + \Delta e\} - \min\{l, \text{expnt}(x) + 1022\}, 0\}$$

where $\Delta e = \text{expnt}(x) - \text{expnt}(y)$, $k = \min\{n \in \mathbb{Z}_{\geqslant 0} : |x + y| < 2^{\text{expnt}(x)+1+n}\}$, and $l = \max\{n \in \mathbb{Z}_{\geqslant 0} : |x - y| < 2^{\text{expnt}(x)+1-n}\}$.

The lemma says that when $\sigma(x)$ and $\sigma(y)$ are fixed, $\sigma(x + y)$ and $\sigma(x - y)$ decrease as x and y get closer to each other (since it makes Δe smaller and l larger). In the lemma, the integer k represents whether there is a carry-over during the addition $x + y$, as $k = 0$ if no carry-over and $k = 1$ otherwise. The integer l is subtracted from the upper bound on $\sigma(x - y)$ to consider the case when x and y are close: if they are close enough, some of x's most significant bits can be cancelled out by y's corresponding significant bits during the subtraction $x - y$, thereby reducing $\sigma(x - y)$. The term $\min\{\cdots, \text{expnt}(x) + 1022\}$ is necessary to consider the case when $|x - y| < 2^{-1022}$.

Note that Lemma 5.6 is a generalization of Sterbenz's theorem. We are unaware of any previous work that proves this lemma. Moreover, this lemma is a general fact about floating-point that may have applicability beyond this paper.

We present the rule based on Theorem 5.4 in Figure 7. To apply Theorem 5.4, we need to track $\sigma(e)$ for each expression e, which is defined as:

$$\sigma(e) \triangleq \max\{\sigma(\mathcal{E}(e)(x)) : x \in X \cap \mathbb{F}\}$$

$$\frac{\begin{array}{ccc} & \mathcal{K}_1(e_1) = (\_,\_,\sigma_1) & \\ (\mathcal{K}, e_1) \rhd (\mathcal{K}_1, \mathcal{A}_{1,\vec{\delta}}) & \mathcal{K}_2(e_2) = (\_,\_,\sigma_2) & * \in \{+,-,\times,/\} \\ (\mathcal{K}_1, e_2) \rhd (\mathcal{K}_2, \mathcal{A}_{2,\vec{\delta}}) & \sigma' = \mathit{bound}\text{-}\sigma(*, \mathcal{A}_{1,\vec{\delta}}, \mathcal{A}_{2,\vec{\delta}}, \sigma_1, \sigma_2) & \sigma' \leqslant 53 \end{array}}{(\mathcal{K}, e_1 \circledast e_2) \rhd (\mathcal{K}', \mathcal{A}'_{\vec{\delta}}),\ \text{where} \begin{cases} \mathcal{A}'_{\vec{\delta}} = \mathit{compress}(\mathcal{A}_{1,\vec{\delta}} \boxed{*} \mathcal{A}_{2,\vec{\delta}}) \\ \mathcal{K}' = \mathcal{K}_2[e_1 \circledast e_2 \mapsto (\mathcal{A}'_{\vec{\delta}}, \mathtt{true}, \sigma')] \end{cases}} \ \text{R13}$$

Fig. 7. Rule for using $\sigma(\cdot)$

Algorithm 1 *bound-*$\sigma(*, f_1, f_2, \sigma_1, \sigma_2)$

```
Let $D = dom(f_1)$
if $* = +$ then
    Compute $\Delta e \in \mathbb{Z}_{\geqslant 0}$ such that $\forall \vec{x} \in D.\ |\mathrm{expnt}(f_1(\vec{x})) - \mathrm{expnt}(f_2(\vec{x}))| \leqslant \Delta e$
    if $\forall \vec{x} \in D.\ \mathrm{expnt}(f_1(\vec{x})) \geqslant \mathrm{expnt}(f_2(\vec{x}))$ then
        if $\forall \vec{x} \in D.\ f_1(\vec{x}) f_2(\vec{x}) \geqslant 0$ then
            Compute $k \in \mathbb{Z}_{\geqslant 0}$ such that $\forall \vec{x} \in D.\ |f_1(\vec{x}) + f_2(\vec{x})| < 2^{\mathrm{expnt}(f_1(\vec{x}))+1+k}$
            return $\max\{\sigma_1, \sigma_2 + \Delta e\} + k$
        if $\forall \vec{x} \in D.\ f_1(\vec{x}) f_2(\vec{x}) < 0$ then
            Compute $\begin{cases} l \in \mathbb{Z}_{\geqslant 0} \text{ such that } \forall \vec{x} \in D.\ |f_1(\vec{x}) + f_2(\vec{x})| < 2^{\mathrm{expnt}(f_1(\vec{x}))+1-l} \\ em_1 \in \mathbb{Z} \text{ such that } \forall \vec{x} \in D.\ \mathrm{expnt}(f_1(\vec{x})) \geqslant em_1 \end{cases}$
            return $\max\{\max\{\sigma_1, \sigma_2 + \Delta e\} - \min\{l, em_1 + 1022\}, 0\}$
    if $\forall \vec{x} \in D.\ \mathrm{expnt}(f_1(\vec{x})) \leqslant \mathrm{expnt}(f_2(\vec{x}))$ then
        [symmetric to the above case]
    return $\max\{\sigma_1, \sigma_2\} + \Delta e + 1$
if $* = -$ then
    [similar to the case $* = +$]
if $* = \times$ then
    Compute $k \in \mathbb{Z}_{\geqslant 0}$ such that $\forall \vec{x} \in D.\ |f_1(\vec{x}) f_2(\vec{x})| \geqslant 2^{-1022-k}$
    return $\sigma_1 + \sigma_2 + k$
if $* = /$ then
    return $\infty$
```

The cache $\mathcal{K}$ is extended to store an upper bound of $\sigma(e)$ for each e. The rule R13 computes σ' that upper bounds $\sigma(e_1 * e_2)$ via Algorithm 1, and then checks whether $\sigma' \leqslant 53$. If the check passes, the rule constructs an abstraction of $e_1 \circledast e_2$ without applying the $(1+\epsilon)$-property.

Next, we discuss Algorithm 1 in more detail. For brevity, the algorithm uses the notation f_i to represent an abstraction $\mathcal{A}_{i,\vec{\delta}}$ $(i = 1, 2)$; D represents $X \times [-\Delta_1, \Delta_1] \times \cdots \times [-\Delta_n, \Delta_n]$ and $\vec{x}$ represents $(x, \delta_1, \cdots, \delta_n)$. With this notation Algorithm 1 upper bounds $\sigma(f_1 * f_2)$, given upper bounds on $\sigma(f_1)$ and $\sigma(f_2)$. Formally, the algorithm meets the following specification:

Lemma 5.7. *Consider* $* \in \{+,-,\times,/\}$, $f_i : D \to \mathbb{R}$, *and* $\sigma_i \in \mathbb{Z}_{\geqslant 0}$ *for* $i \in \{1,2\}$ *and* $D \subset \mathbb{R}^m$. *Let* $\sigma' = \mathit{bound}\text{-}\sigma(*, f_1, f_2, \sigma_1, \sigma_2)$. *Then for any* $\vec{x} \in D$ *such that* $\forall i \in \{1,2\}.\ \sigma(f_i(\vec{x})) \leqslant \sigma_i$,

$$|f_1(\vec{x}) * f_2(\vec{x})| \leqslant \max \mathbb{F} \qquad \Longrightarrow \qquad \sigma(f_1(\vec{x}) * f_2(\vec{x})) \leqslant \sigma'$$

Algorithm 1 (conservatively) returns ∞ for division because the result of dividing two doubles often has no representation with finite binary digits (*e.g.*, $1/5 = 2^{-3} \times 1.10011001\cdots_{(2)}$).

Algorithm 1 solves multiple optimization problems to compute the final bound. For example, to obtain Δe, we first compute $[em_i, eM_i]$ ($i = 1, 2$), an interval bounding the range of $\text{expnt}(f_i(\vec{x}))$ over $\vec{x} \in D$, by solving optimization problems: $em_i = \text{expnt}(\min\{|f_i(\vec{x})| : \vec{x} \in D\})$ and $eM_i = \text{expnt}(\max\{|f_i(\vec{x})| : \vec{x} \in D\})$. Next, Δe is set to $\Delta e = \max\{eM_1 - em_2, eM_2 - em_1\}$. For another example, consider the **if** condition on line 5. The condition can be conservatively checked by deciding whether $(fm_1 \geqslant 0 \wedge fm_2 \geqslant 0) \vee (fM_1 \leqslant 0 \wedge fM_2 \leqslant 0)$ holds, where $fm_i = \min\{f_i(\vec{x}) : \vec{x} \in D\}$ and $fM_i = \max\{f_i(\vec{x}) : \vec{x} \in D\}$.

Now that the cache has been extended to store an upper bound of $\sigma(e)$, we need to extend all the previous rules accordingly and ensure that Theorem 4.6 holds. For instance, the rule R4 is extended to

$$\frac{(\mathcal{K}, \textit{bit-mask}(e_1, B)) \rhd (\mathcal{K}_1, \mathcal{A}_{1,\vec{\delta}}) \qquad \mathcal{K}_1(e_1) = (\_, \_, \sigma_1)}{(\mathcal{K}, \textit{bit-mask}(e_1, B)) \rhd (\mathcal{K}', \mathcal{A}'_{\vec{\delta}}), \text{ where } \begin{cases} \delta' = \textit{fresh}(2^{-52+B}),\ \delta'' = \textit{fresh}(2^{-1074+B}) \\ \mathcal{A}'_{\vec{\delta}} = \textit{compress}(\mathcal{A}_{1,\vec{\delta}} \boxtimes (1+\delta') \boxplus \delta'') \\ \sigma' = \min\{\sigma_1, 53 - B\} \\ \mathcal{K}' = \mathcal{K}_1[\textit{bit-mask}(e_1, B) \mapsto (\mathcal{A}'_{\vec{\delta}}, \texttt{false}, \sigma')] \end{cases}} \ \text{R4}'$$

because $\textit{bit-mask}(e_1, B)$ masks out B least significant bits of e_1's significand. We can prove that Theorem 4.6 still holds, using the extended definition of a sound cache: a cache $\mathcal{K}$ is sound if for any $e \in \textit{dom}(\mathcal{K})$ with $\mathcal{K}(e) = (\mathcal{A}_{\vec{\delta}}, b, \sigma)$, $\mathcal{A}_{\vec{\delta}}$ and b satisfy the previous condition and $\sigma(e) \leqslant \sigma$.

5.5 Refined $(1 + \epsilon)$-property

In some cases, the absolute error term δ' in Theorem 3.2 can be soundly removed according to the following *refined* $(1 + \epsilon)$-property:

Theorem 5.8. *Let $x, y \in \mathbb{F}$ and $* \in \{+, -, \times, /\}$. Assume $|x * y| \leqslant \max \mathbb{F}$. For any $* \in \{+, -\}$, and for any $* \in \{\times, /\}$ with either $x * y = 0$ or $|x * y| \geqslant 2^{-1022}$, we have*

$$x \circledast y = (x * y)(1 + \delta)$$

for some $|\delta| < \epsilon$.

The theorem states that the absolute error term δ' is always unnecessary for addition and subtraction, and for the other operations it is unnecessary if the exact result of the operation is not in the subnormal range. The theorem is standard and follows from three properties of floating-point: $\text{ErrRel}(r, \text{fl}(r)) < \epsilon$ for any $r \in \mathbb{R}$ not in the subnormal range, every double is a multiple of $\text{ulp}(0)$, and any multiple of $\text{ulp}(0)$ is a double if it is in the subnormal range.

To use Theorem 5.8 in constructing an abstraction of an expression e, we need to know whether e can evaluate to a number between 0 and $\pm 2^{-1022}$. To this end, define a function $\mu(e) > 0$ over expressions, which denotes how close a non-zero evaluation result of e can be to 0:

$$\mu(e) \triangleq \min\{|\mathcal{E}(e)(x)| \neq 0 : x \in X \cap \mathbb{F}\}$$

where $\min \varnothing = \infty$. An important property related to $\mu(e)$ is the following lemma:

Lemma 5.9. *Let $\mu_1, \mu_2 > 0$. Consider any $d_1, d_2 \in \mathbb{F}$ such that $|d_i| \geqslant \mu_i$ ($i = 1, 2$). Then*

$$d_1 + d_2 \neq 0 \qquad \Longrightarrow \qquad |d_1 + d_2| \geqslant \frac{1}{2}\text{ulp}(\max\{\mu_1, \mu_2\})$$

The lemma states that if the sum of two doubles is non-zero, then its magnitude cannot be smaller than some (small) number. The lemma holds because there is a finite gap between any two consecutive doubles. To illustrate an application of the lemma, consider $X = [0, 2]$ and $e = x \ominus 1$.

$$\frac{\begin{array}{ccc} & \mathcal{K}_1(e_1) = (\_,\_,\sigma_1,\mu_1) & \\ & \mathcal{K}_2(e_2) = (\_,\_,\sigma_2,\mu_2) & \\ (\mathcal{K}, e_1) \rhd (\mathcal{K}_1, \mathcal{A}_{1,\vec{\delta}}) & \sigma' = \mathit{bound}\text{-}\sigma(*, \mathcal{A}_{1,\vec{\delta}}, \mathcal{A}_{2,\vec{\delta}}, \sigma_1, \sigma_2) & \\ (\mathcal{K}_1, e_2) \rhd (\mathcal{K}_2, \mathcal{A}_{2,\vec{\delta}}) & \mu' = \mathit{bound}\text{-}\mu(*, \mathcal{A}_{1,\vec{\delta}}, \mathcal{A}_{2,\vec{\delta}}, \mu_1, \mu_2) & * \in \{+,-\} \end{array}}{(\mathcal{K}, e_1 \circledast e_2) \rhd (\mathcal{K}', \mathcal{A}'_{\vec{\delta}}),\ \text{where} \begin{cases} \delta' = \mathit{fresh}(\epsilon) \\ \mathcal{A}'_{\vec{\delta}} = \mathit{compress}((\mathcal{A}_{1,\vec{\delta}} \boxast \mathcal{A}_{2,\vec{\delta}}) \boxtimes (1+\delta')) \\ \sigma'' = \min\{\sigma', 53\},\ \mu'' = \max\{\mathrm{fl}^-(\mu'), 2^{-1074}\} \\ \mathcal{K}' = \mathcal{K}_2[e_1 \circledast e_2 \mapsto (\mathcal{A}'_{\vec{\delta}}, \mathtt{false}, \sigma'', \mu'')] \end{cases}} \ \text{R14}$$

$$\frac{\begin{array}{ccc} & \mathcal{K}_1(e_1) = (\_,\_,\sigma_1,\mu_1) & \\ & \mathcal{K}_2(e_2) = (\_,\_,\sigma_2,\mu_2) & \\ (\mathcal{K}, e_1) \rhd (\mathcal{K}_1, \mathcal{A}_{1,\vec{\delta}}) & \sigma' = \mathit{bound}\text{-}\sigma(*, \mathcal{A}_{1,\vec{\delta}}, \mathcal{A}_{2,\vec{\delta}}, \sigma_1, \sigma_2) & * \in \{\times, /\} \\ (\mathcal{K}_1, e_2) \rhd (\mathcal{K}_2, \mathcal{A}_{2,\vec{\delta}}) & \mu' = \mathit{bound}\text{-}\mu(*, \mathcal{A}_{1,\vec{\delta}}, \mathcal{A}_{2,\vec{\delta}}, \mu_1, \mu_2) & \mu' \geqslant 2^{-1022} \end{array}}{(\mathcal{K}, e_1 \circledast e_2) \rhd (\mathcal{K}', \mathcal{A}'_{\vec{\delta}}),\ \text{where} \begin{cases} \delta' = \mathit{fresh}(\epsilon) \\ \mathcal{A}'_{\vec{\delta}} = \mathit{compress}((\mathcal{A}_{1,\vec{\delta}} \boxast \mathcal{A}_{2,\vec{\delta}}) \boxtimes (1+\delta')) \\ \sigma'' = \min\{\sigma', 53\},\ \mu'' = \max\{\mathrm{fl}^-(\mu'), 2^{-1074}\} \\ \mathcal{K}' = \mathcal{K}_2[e_1 \circledast e_2 \mapsto (\mathcal{A}'_{\vec{\delta}}, \mathtt{false}, \sigma'', \mu'')] \end{cases}} \ \text{R15}$$

Fig. 8. Rules for applying the refined $(1+\epsilon)$-property

Algorithm 2 *bound-*$\mu(*, f_1, f_2, \mu_1, \mu_2)$

```
Let $D = dom(f_1)$
if $0 \notin \{f_1(\vec{x}) * f_2(\vec{x}) : \vec{x} \in D\}$ then
    return $\mu' \in \mathbb{R}$ such that $0 < \mu' \leqslant \min\{|f_1(\vec{x}) * f_2(\vec{x})| : \vec{x} \in D\}$
else
    if $* = \{+,-\}$ then
        return $\min\{\frac{1}{2}\mathrm{ulp}(\max\{\mu_1,\mu_2\}), \mu_1, \mu_2\}$
    else if $* = \times$ then
        return $\mu_1\mu_2$
    else if $* = /$ then
        Compute $M_2 \in \mathbb{R}$ such that $M_2 \geqslant \max\{|f_2(\vec{x})| : \vec{x} \in D\}$
        return $\mu_1/M_2$
```

Clearly e can evaluate to 0 (for an input $x = 1$). However, from the lemma we can conclude $\mu(e) \geqslant \frac{1}{2}\mathrm{ulp}(1) = 2^{-53}$, *i.e.*, e can never evaluate to any value in $(0, 2^{-53})$.

The rules based on Theorem 5.8 are given in Figure 8. To keep track of $\mu(e)$, the cache $\mathcal{K}$ is extended to store a lower bound of $\mu(e)$ for each e. Both the rules R14 and R15 first compute a lower bound μ' on $\mu(e_1 * e_2)$ using Algorithm 2. The rule R15 then checks whether $\mu' \geqslant 2^{-1022}$; if the check passes, the rule constructs an abstraction of $e_1 \circledast e_2$ without adding an absolute error term δ'', based on Theorem 5.8. On the other hand, the rule R14 does not add the absolute error term δ'' regardless of whether $\mu' \geqslant 2^{-1022}$, also based on Theorem 5.8. Note that both rules set a lower bound of $\mu(e_1 \circledast e_2)$ to $\max\{\mathrm{fl}^-(\mu'), 2^{-1074}\}$ because the smallest positive double is 2^{-1074}, where $\mathrm{fl}^-(r) \triangleq \max\{d \in \mathbb{F} : d \leqslant r\}$.

Consider Algorithm 2. For brevity, the algorithm uses the notation of Algorithm 1 (*e.g.*, f_i to represent an abstraction $\mathcal{A}_{i,\vec{\delta}}$). Given lower bounds on $\mu(f_1)$ and $\mu(f_2)$, the algorithm finds a lower bound on $\mu(f_1 * f_2)$ using Lemma 5.9:

Lemma 5.10. *Consider* $* \in \{+, -, \times, /\}$, $f_i : D \to \mathbb{R}$, *and* $\mu_i \in \mathbb{R}_{>0}$ *for* $i \in \{1, 2\}$ *and* $D \subset \mathbb{R}^m$. *Let* $\mu' = \textit{bound-}\mu(*, f_1, f_2, \mu_1, \mu_2)$. *Then for any* $\vec{x} \in D$ *such that* $\forall i \in \{1, 2\}.\ f_i(\vec{x}) = 0 \lor |f_i(\vec{x})| \geqslant \mu_i$,

$$f_1(\vec{x}) * f_2(\vec{x}) \neq 0 \qquad \Longrightarrow \qquad |f_1(\vec{x}) * f_2(\vec{x})| \geqslant \mu'$$

Like Algorithm 1, Algorithm 2 requires solving optimization problems to obtain the final answer. For instance, the **if** condition on line 2 can be conservatively checked by deciding whether $0 \notin [fm, fM]$, where $fm = \min\{f_1(\vec{x}) * f_2(\vec{x}) : \vec{x} \in D\}$ and $fM = \max\{f_1(\vec{x}) * f_2(\vec{x}) : \vec{x} \in D\}$.

Since the cache has been extended to store a lower bound of $\mu(e)$, we need to extend all the previous rules and check Theorem 4.6 again. For example, the rules R1 and R2 are extended to

$$\frac{}{(\mathcal{K}, c) \rhd (\mathcal{K}[c \mapsto (c, \texttt{false}, \sigma(c), \mu(c))], c)}\ \text{R1}' \qquad \frac{}{(\mathcal{K}, x) \rhd (\mathcal{K}[x \mapsto (x, \texttt{false}, 53, \mu(x))], x)}\ \text{R2}'$$

We can prove that Theorem 4.6 is still true, by extending the definition of a sound cache: a cache $\mathcal{K}$ is sound if for any $e \in dom(\mathcal{K})$ with $\mathcal{K}(e) = (\mathcal{A}_{\vec{\delta}}, b, \sigma, \mu)$, $\mathcal{A}_{\vec{\delta}}$, b, and σ satisfy the previous condition and $\mu(e) \geqslant \mu$. The complete rules appear in §A.2.

5.6 Bounding Ulp Error

The main goal of this paper is to find an ulp error bound Θ_{ulp} of an expression e with respect to a mathematical specification $f(x)$, *i.e.*, to find Θ_{ulp} such that

$$\text{ErrUlp}(f(x), \mathcal{E}(e)(x)) \leqslant \Theta_{\text{ulp}} \quad \text{for all } x \in X \cap \mathbb{F} \tag{11}$$

To achieve the goal, we first construct a sound abstraction $\mathcal{A}_{\vec{\delta}}(x) = a(x) + \sum_i b_i(x)\delta_i$ of e by applying the rules discussed so far, and then compute a relative error bound Θ_{rel} of e with respect to $f(x)$ by solving the following optimization problems over $x \in X$:

$$\Theta_{\text{rel}} = \max_{x \in X} \left| \frac{f(x) - a(x)}{f(x)} \right| + \sum_i \max_{x \in X} \left| \frac{b_i(x)}{f(x)} \right| \cdot \Delta_i$$

We can prove that Θ_{rel} is an upper bound on the relative error of e with respect to $f(x)$, *i.e.*, $\text{ErrRel}(f(x), \mathcal{E}(e)(x)) \leqslant \Theta_{\text{rel}}$ for all $x \in X \cap \mathbb{F}$, using the triangle inequality and the soundness of $\mathcal{A}_{\vec{\delta}}(x)$. Finally, Theorem 3.3 enables us to obtain $\Theta_{\text{ulp}} = \Theta_{\text{rel}}/\epsilon$ that satisfies Eq. 11.

However, the above approach often cannot prove an ulp error bound less than 1 ulp. To illustrate, consider $X = [1, 2]$, $e = x \oplus 1$, and $f(x) = x + 1$. Applying the rules R1′, R2′, and R14 to e gives an abstraction $\mathcal{A}_{\vec{\delta}}(x) = (x + 1) + (x + 1)\delta$ of e with $|\delta| \leqslant \epsilon$, and we obtain $\Theta_{\text{rel}} = 0 + 1 \cdot \epsilon = \epsilon$ and $\Theta_{\text{ulp}} = \epsilon/\epsilon = 1$. But the tightest ulp error bound of e is in fact 0.5 ulps, as a single floating-point operation always has a rounding error of $\leqslant 0.5$ ulps.

To obtain an ulp error bound less than 1 ulp, we use the following property about ulp errors which has been used to prove very precise ulp error bounds in [Harrison 2000b]:

Theorem 5.11. *Let* $r \in [-\max \mathbb{F}, \max \mathbb{F}]$, $d_1, d_2 \in \mathbb{F}$, *and* $* \in \{+, -, \times, /\}$. *Assume* $|d_1 * d_2| \leqslant \max \mathbb{F}$. *Then* $\text{ErrUlp}(r, d_1 * d_2) \leqslant 1$ *implies*

$$\text{ErrUlp}(r, d_1 \circledast d_2) \leqslant \text{ErrUlp}(r, d_1 * d_2) + \frac{1}{2} \tag{12}$$

The theorem states that the ulp error of a floating-point operation is upper bounded by the ulp error of the corresponding exact operation plus 1/2. Note that the condition $\text{ErrUlp}(r, d_1 * d_2) \leqslant 1$ in the theorem is necessary; Eq. 12 may not hold if $\text{ErrUlp}(r, d_1 * d_2) > 1$. For the case when

$\mathrm{ErrUlp}(r, d_1 * d_2) > 1$, we can use the following similar statement: $\mathrm{ErrUlp}(r, d_1 * d_2) \leqslant 2^{53}$ implies $\mathrm{ErrUlp}(r, d_1 \circledast d_2) \leqslant \mathrm{ErrUlp}(r, d_1 * d_2) + 1$.

Using Theorem 5.11, we compute an ulp error bound $\Theta_{\mathrm{ulp,new}}$ of e tighter than Θ_{ulp} as follows. We first construct an abstraction $\mathcal{A}'_{\vec{\delta}}$ of e by applying the previous rules as before, but with the assumption that the last operation of e is exact. Then we compute an ulp error bound Θ'_{ulp} from $\mathcal{A}'_{\vec{\delta}}$ (not from $\mathcal{A}_{\vec{\delta}}$) by following the exactly same steps as above. Finally, we obtain a new, tighter ulp error bound of e by $\Theta_{\mathrm{ulp,new}} = \Theta'_{\mathrm{ulp}} + 1/2$ if $\Theta'_{\mathrm{ulp}} \leqslant 1$, or by $\Theta_{\mathrm{ulp,new}} = \Theta'_{\mathrm{ulp}} + 1$ if $1 < \Theta'_{\mathrm{ulp}} \leqslant 2^{53}$. Using Theorem 5.11, we can show that $\Theta_{\mathrm{ulp,new}}$ satisfies Eq. 11.

6 IMPLEMENTATION

We implement the techniques described in §4 and §5 using Mathematica 11.0.1. To solve optimization problems that appear in constructing abstractions and computing error bounds, we use Mathematica's built-in functions `MaxValue[···]` and `MinValue[···]` that find the global maximum/minimum of an objective function soundly using analytical optimization (not numerical optimization).

Most optimization problems occurring in our analysis involve abstractions, *i.e.*, the minimization or the maximization of an abstraction $\mathcal{A}_{\vec{\delta}}$ or its magnitude $|\mathcal{A}_{\vec{\delta}}|$. However, these optimization problems are multi-variate and are difficult to solve in general. Hence, our implementation computes sound lower/upper bounds of these optimization objectives via Eq. (13)-(16), and uses these instead of the exact minimization/maximization results as in [Lee et al. 2016; Solovyev et al. 2015].

$$\max_{x \in X, |\delta_i| \leqslant \Delta_i} \mathcal{A}_{\vec{\delta}}(x) \leqslant \max_{x \in X} a(x) + \sum_i \max_{x \in X} |b_i(x)| \cdot \Delta_i \tag{13}$$

$$\min_{x \in X, |\delta_i| \leqslant \Delta_i} \mathcal{A}_{\vec{\delta}}(x) \geqslant \min_{x \in X} a(x) - \sum_i \max_{x \in X} |b_i(x)| \cdot \Delta_i \tag{14}$$

$$\max_{x \in X, |\delta_i| \leqslant \Delta_i} |\mathcal{A}_{\vec{\delta}}(x)| \leqslant \max_{x \in X} |a(x)| + \sum_i \max_{x \in X} |b_i(x)| \cdot \Delta_i \tag{15}$$

$$\min_{x \in X, |\delta_i| \leqslant \Delta_i} |\mathcal{A}_{\vec{\delta}}(x)| \geqslant \min_{x \in X} |a(x)| - \sum_i \max_{x \in X} |b_i(x)| \cdot \Delta_i \tag{16}$$

In Eq. (13)-(16), the RHS represents a lower/upper bound of the LHS. As each RHS is a collection of uni-variate optimization problems, it is much easier to solve.

Our implementation checks that the evaluation of an expression e does not introduce $\pm\infty$ or NaNs (§3). For proving the absence of overflows, the inequality $\max_{x,\vec{\delta}} |\mathcal{A}'_{\vec{\delta}}| \leqslant \max \mathbb{F}$ is checked for every subexpression e' of e, where $\mathcal{A}'_{\vec{\delta}}$ is an abstraction of e'. For proving the absence of divide-by-0 errors, the inequality $0 \notin [\min_{x,\vec{\delta}} \mathcal{A}'_{\vec{\delta}}, \max_{x,\vec{\delta}} \mathcal{A}'_{\vec{\delta}}]$ is checked for every e' such that $e'' \oslash e'$ is a subexpression of e, where $\mathcal{A}'_{\vec{\delta}}$ is an abstraction of e'. Our implementation checks these conditions by solving additional optimization problems.

For some expressions e and input intervals $X = [l, r]$, our technique might produce imprecise results. In such scenarios, typically we can subdivide the interval into two (or more) subintervals $[l_1, r_1] \cup [l_2, r_2] = [l, r]$ such that separate analysis of the subintervals does yield tight bounds. This situation arises because the preconditions of different exactness properties are satisfied on different subintervals, but few or no such properties hold for the entire interval.

To prove tighter error bounds in such scenarios, our implementation works as follows. Let e be an expression, X be an input interval, and $\Theta_{\mathrm{ulp,goal}}$ be an ulp error bound that we aim to prove (we use $\Theta_{\mathrm{ulp,goal}} = 0.53$ in the evaluation). We first compute an ulp error bound Θ_{ulp} by applying our technique to e and X. If $\Theta_{\mathrm{ulp}} \leqslant \Theta_{\mathrm{ulp,goal}}$ or if we are out of computation budget, return Θ_{ulp}. Otherwise, we bisect X into two subintervals X_1 and X_2, recursively compute an ulp error bound

Table 1. Summary of results. For each implementation (column 1) and for each input interval (column 2), column 3 shows the ulp error bound of the implementation over the input interval. Column 4 is the number of the initial input intervals from column 2, and column 5 is the number of disjoint intervals obtained by repeatedly bisecting the initial input intervals (until the ulp error bound of column 3 is obtained). Column 6 shows the total wall clock time taken to obtain the ulp error bound of column 3 (in minutes), and column 7 shows the maximum time taken to verify an initial input interval (in seconds). Here $^\diamond$ denotes that we used 16 instances of Mathematica in parallel; by default we run only one instance of Mathematica.

	Input Interval	Ulp Error Bound	# of Intervals Before Bisections	# of Intervals After Bisections	Verification Time (m)	Max Time per Interval (s)
exp	$[-4, 4]$	7.552	13	13	0.52	2.5
sin	$[-\pi, \pi] \setminus (-2^{-252}, 2^{-252})$	0.530	66	142	68	446
tan	$[\frac{13}{128}, \frac{17\pi}{64})$	0.595	9	22	40	495
	$[\frac{17\pi}{64}, \frac{\pi}{2})$	13.33	8	8	10	81
log	$[2^{-1022}, \max \mathbb{F}]$	0.583	4.2×10^6	4.2×10^6	461 hrs$^\diamond$	24

$\Theta_{\text{ulp},\text{i}}$ of e over X_i ($i = 1, 2$), and return the maximum of $\Theta_{\text{ulp},1}$ and $\Theta_{\text{ulp},2}$. Such approaches that bisect input intervals are well-known and are a part of existing commercial tools [Delmas et al. 2009].

7 CASE STUDIES

We evaluate our technique on the benchmarks of [Lee et al. 2016; Schkufza et al. 2014] which consist of implementations (exp, sin, tan, and log) of four different transcendental functions (e^x, $\mathit{sin}\, x$, $\mathit{tan}\, x$, and $\mathit{log}\, x$). The code in exp is a custom implementation used in S3D [Chen et al. 2009], a combustion chemistry simulator; sin, tan, and log are taken from Intel® Math Library libimf which is Intel's implementation of the C library math.h and contains "highly optimized and very accurate mathematical functions."[4] All these implementations are loop-free programs, and have been written directly in x86 assembly for the best performance. We remark that analyzing these x86 implementations involves substantial engineering effort beyond what is described in this paper or in [Lee et al. 2016], as modeling the semantics of the more complex x86 instructions correctly and in detail is itself a significant undertaking and, at least so far, the overlap in instructions used among the benchmarks we have studied has not been as much as might be hoped.

We find an ulp error bound of each x86 implementation $P \in \{\texttt{exp}, \texttt{sin}, \texttt{tan}, \texttt{log}\}$ as follows. We first apply the technique from [Lee et al. 2016] that eliminates bit-level and integer arithmetic computations intermingled with floating-point operations using partial evaluation. The result is that for each P with an input interval X, the method yields k different expressions $e_1, \cdots, e_k$ in our core language (Figure 2), corresponding input intervals $X_1, \cdots, X_k$, and a (small) set H of individual doubles (typically $|H| < 250$) such that $\forall i \in \{1, \cdots, k\}. \forall x \in X_i \cap \mathbb{F}.\ P(x) = \mathcal{E}(e_i)(x)$ and $X \cap \mathbb{F} = \bigcup_{1 \leqslant i \leqslant k} (X_i \cap \mathbb{F}) \cup H$. Let us call $X_1, \cdots, X_k$ the *initial* input intervals from X. We then find an ulp error bound $\Theta_{\text{ulp},i}$ of e_i over X_i with respect to the exact mathematical function $f \in \{e^x, \mathit{sin}\, x, \mathit{tan}\, x, \mathit{log}\, x\}$ for each $1 \leqslant i \leqslant k$. Finally, we obtain an ulp error bound of P over X with respect to f by taking the maximum of $\max\{\Theta_{\text{ulp},i} : 1 \leqslant i \leqslant k\}$ and $\max\{\text{ErrUlp}(f(x), P(x)) : x \in H\}$. Although the procedure of [Lee et al. 2016] relies on procedures that may need manual intervention, these procedures have been automated for our benchmarks.

[4] https://software.intel.com/en-us/node/522653

The results of applying our technique to these implementations are summarized in Table 1. Columns 1 and 2 represent P and X, and column 3 represents the proved ulp error bound of P over X with respect to f. Column 4 shows k, the number of the initial input intervals $X_1, \cdots, X_k$, while column 5 shows the number of disjoint intervals after bisecting these initial input intervals (§6). Column 6 shows the total wall clock time taken to obtain the bounds in column 3, and column 7 shows the maximum time taken to verify an initial input interval. In particular, the maximum time taken by our analysis (for a given expression e_i and an initial input interval X_i) is less than 10 minutes.

Although the benchmarks of [Lee et al. 2016] include `fdim` (Intel's implementation of $\mathit{fdim}(x, y) \triangleq x - y$ if $x > y$ and 0 otherwise), this benchmark is easy to verify and its correctness follows from Theorem 5.11. On the other hand, verifying the other implementations `exp`, `sin`, `tan`, and `log` require the exactness results discussed in §5, as explained below.

For the `exp` implementation, our technique finds an error bound of 7.552 ulps over the interval $[-4, 4]$ in 31 seconds. An important step for proving this error bound is the application of rule R8/R9. The error bound of 7.552 ulps is larger than Intel's implementations as the developer sacrificed precision for performance in this custom implementation; even assuming every floating-point operation in `exp` is exact, the ulp error bound is 4.06 ulps over $[-4, 4]$, and indeed we typically observe error of 3-4 ulps on concrete inputs. The documentation of `exp` specifies that the implementation is supposed to compute e^x with “small errors” for inputs between -2.6 and 0.12, and we have successfully quantified the maximum precision loss formally.

In sharp contrast to such custom implementations, standard math libraries such as `libimf` claim that the maximum precision loss is below one ulp. For `sin`, our technique finds an error bound of 0.530 ulps over the interval $X = [-\pi, \pi] \backslash (-2^{-252}, 2^{-252})$ in 68 minutes. We exclude the interval $(-2^{-252}, 2^{-252})$ because the `sin` implementation we analyze is executed only for $|x| \in [2^{-252}, 90112)$. For inputs outside this range, different implementations are used. To prove the error bound, rules R8 and R9 (related to Sterbenz's theorem, §5.2), rules R10, R11, and R12 (related to Dekker's theorem, §5.3), and rules R14 and R15 (related to the refined $(1 + \epsilon)$-property, §5.5) are crucial.

Proving the error bound of `sin` shown in Table 1 requires us to analyze `sin` over 142 disjoint intervals, the result of repeatedly bisecting the 66 initial input intervals. In particular, to verify `sin` over $X_{66} = [63\pi/64, \pi]$, we need to repeatedly bisect X_{66} to have 13 disjoint subintervals $[63\pi/64, y_1)$, $[y_1, y_2)$, $\cdots$, $[y_{12}, \pi]$, where $y_0 = 63\pi/64$, $y_{13} = \pi$, and $y_i = (y_{i-1} + y_{13})/2$ $(i = 1, \cdots, 12)$. We require many subintervals because the antecedents of the rules R11 and R12 are valid only over small intervals.

For the `tan` implementation, our technique finds an error bound of 0.595 and 13.33 ulps over the intervals $[13/128, 17\pi/64)$ and $[17\pi/64, \pi/2)$, respectively, in 50 minutes. We exclude the interval $[0, 13/128)$ because our benchmark implementation is supposed to compute $\tan x$ precisely only for $|x| \in [13/128, 12800)$. To obtain the error bounds, it is crucial to apply all the rules used in verifying `sin` multiple times (R8, R9, R10, R11, R12, R14, and R15). Additionally, the rules R4′ and R13 (related to $\sigma(\cdot)$, §5.4) are used to precisely abstract bit-mask operations (which are absent in `sin`).

For `tan` over the input interval $X = [17\pi/64, \pi/2)$, we obtain the error bound of 13.33 ulps. The main culprit is the requirement that, though our abstractions can be non-linear in x, they must be linear in each δ variable. For example, consider the following expressions that appear in `tan`:

$$e_1 = \mathit{bit\text{-}mask}(e_0, 18), \quad e_2 = \mathit{bit\text{-}mask}(1 \oslash e_1, 35), \quad e = 1 \ominus e_1 \otimes e_2$$

For simplicity, assume that the expression e_0 has no rounding error, *i.e.*, $a(x)$ is a sound abstraction of e_0, and we suppress any δ variable with $|\delta| \leqslant \epsilon$. First, by a precise manual analysis, we show that e

computes the rounding error of the bit-mask operation in e_2: from Lemma 4.3, $g_1(x, \vec{\delta}) = a(x)(1+\delta_1)$ and $g_2(x, \vec{\delta}) = \frac{1}{a(x)(1+\delta_1)}(1 + \delta_2)$ are over-approximations of e_1 and e_2, where $|\delta_1| \leqslant 2^{-34}$ and $|\delta_2| \leqslant 2^{-17}$; thus the following is an over-approximation of e (the operations $\ominus$ and $\otimes$ of e are exact by Theorem 5.1 and 5.4).

$$g(x, \vec{\delta}) = 1 - g_1(x, \vec{\delta}) \times g_2(x, \vec{\delta}) = 1 - a(x)(1 + \delta_1) \times \frac{1}{a(x)(1 + \delta_1)}(1 + \delta_2) = -\delta_2$$

Note that the term $(1 + \delta_1)$ in g_1 and g_2 is exactly cancelled out in computing g. On the other hand, the analysis of e using our abstractions proceeds as follows. First, $\mathcal{A}_{1,\vec{\delta}}(x) = g_1(x, \vec{\delta})$ is a sound abstraction of e_1. However, g_2 is non-linear in δ_1; by the rules R3 and R4, $\mathcal{A}_{2,\vec{\delta}}(x) = \frac{1}{a(x)}(1 + \delta_1')(1 + \delta_2)$ is a sound abstraction of e_2, where $|\delta_1'| \leqslant 2^{-34}$. Given $\mathcal{A}_{1,\vec{\delta}}$ and $\mathcal{A}_{2,\vec{\delta}}$, the following is a sound abstraction of e:

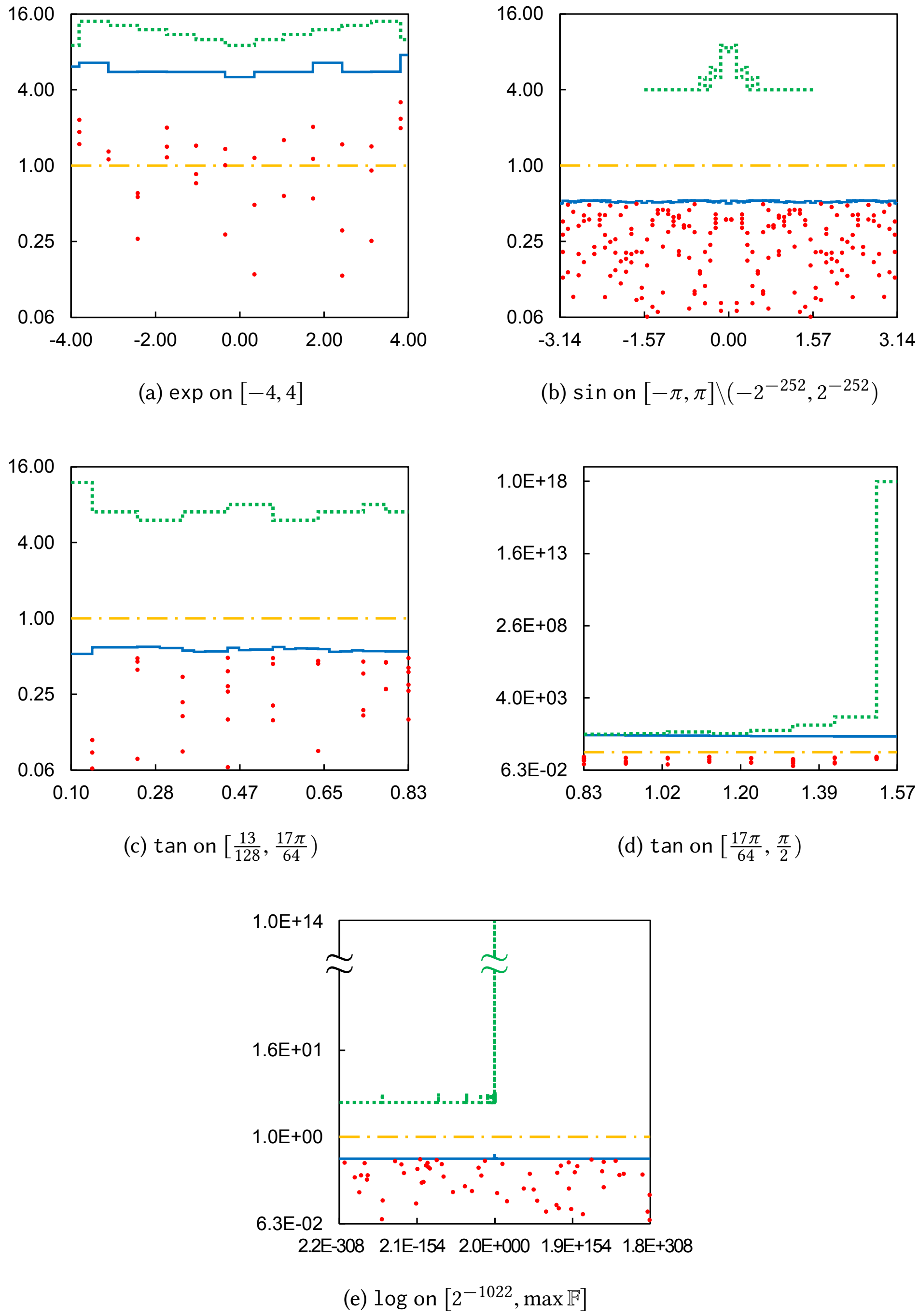
$$\mathcal{A}_{\vec{\delta}}(x) = 1 - \mathcal{A}_{1,\vec{\delta}}(x) \times \mathcal{A}_{2,\vec{\delta}}(x) = 1 - a(x)(1+\delta_1) \times \frac{1}{a(x)}(1+\delta_1')(1+\delta_2) \approx -\delta_2 - (\delta_1 + \delta_1' + \delta_{1,2} + \delta_{1,2}')$$

where $|\delta_{1,2}| \leqslant 2^{-51}$ and $|\delta_{1,2}'| \leqslant 2^{-51}$. These additional δ terms (compared to g) contribute significantly to the error bound of 13.33 ulps for `tan` over X (since $\Delta_{1,2}$ and $\Delta_{1,2}'$ are $2^{-51} = 4\epsilon$).

For the `log` implementation, we apply our technique to its complete input interval $X = [2^{-1022}, \max \mathbb{F}]$ and obtain an error bound of 0.583 ulps. The error bound implies that `log` mostly returns the nearest double to the mathematically exact results. This verification requires all the rules presented in §5. For `log`, we used 16 instances of Mathematica in parallel and required 461 hours of wall clock time to verify all four million cases. We note that this verification is highly parallelizable as analyses over distinct input intervals can be run independently. From Table 1, we observe that the average time and the maximum time taken to verify `log` over each initial input interval are 6 seconds and 24 seconds, respectively.

Figure 9 shows our results graphically. For each graph, the x-axis represents the input values and the y-axis represents the bounds on ulp error (in log scale) between an implementation and the exact mathematical function. The ulp error bounds we prove are shown as solid blue lines, while the ulp error bounds from [Lee et al. 2016] are shown by dotted green lines. Dashed yellow lines in (b)-(e) denote the one ulp bound that we must prove to verify the correctness of `sin`, `tan`, and `log`. The actual ulp errors on concrete inputs (from the set H and random samples) are shown by the red dots. These ulp errors are calculated by comparing the output of an implementation with the exact result computed by Mathematica.

In these graphs, lower bounds are tighter and the bounds proven by our analysis are much better than that of [Lee et al. 2016] across the board. There are some inputs that we analyze but [Lee et al. 2016] do not, *e.g.*, `sin` near $\pm\pi$ and $\log(x)$ for $x \geqslant 4$. For such inputs, the green dotted lines are missing. In (b), (c), and (e), our bounds are below one ulp and we successfully establish the correctness of these implementations. Except for (d), the bounds we infer are tight and the observed ulp errors on concrete inputs are close to the statically inferred bounds. In (d), although our bounds are not tight enough to establish correctness, they are still sound. However, the bounds from [Lee et al. 2016] in this graph are obtained by numerical optimization (as opposed to analytical optimization) and are not guaranteed to be sound. Finally, although we only compare our approach with [Lee et al. 2016] in Figure 9, other generic tools for floating-point verification such as [Darulova and Kuncak 2014; Delmas et al. 2009; Solovyev et al. 2015] would meet a similar fate due to the absence of the relevant exactness results in their analyses.



(a) exp on $[-4, 4]$

(b) sin on $[-\pi, \pi] \backslash (-2^{-252}, 2^{-252})$

(c) tan on $[\frac{13}{128}, \frac{17\pi}{64})$

(d) tan on $[\frac{17\pi}{64}, \frac{\pi}{2})$

(e) log on $[2^{-1022}, \max \mathbb{F}]$

Fig. 9. Each graph shows the ulp error (y-axis in log scale) of each implementation over an input interval (x-axis). Solid blue lines represent our ulp error bounds (Table 1), dotted green lines represent the ulp error bounds from [Lee et al. 2016], and dashed yellow lines represent 1 ulp. Red dots represent actual ulp errors on concrete inputs. The x-axis in (a)-(d) is linear. Because of the large input interval, x-axis in (e) is log-scale.

8 RELATED WORK

An obvious approach to obtain a provably correct mathematical library involves using verified routines for arbitrary precision arithmetic during computations and then rounding the results (*e.g.*, the libmcr library by Sun). However, the resulting implementations have vastly inferior performance compared to the math libraries that exclusively use 64-bit arithmetic. Libraries such as CRLibm aim to keep the maximum error below 0.5 ulps while maintaining performance. The correctness is ensured by a mixture of “pen-and-paper" proofs [Daramy-Loirat et al. 2005] and machine-checkable proofs in Gappa [Daumas and Melquiond 2010; de Dinechin et al. 2011] and Coq [Melquiond 2012; Ramananandro et al. 2016]. The tightness of error bounds and the peculiar structure of rounding errors coupled with optimization tricks make such high performance libraries difficult to verify. Furthermore, industry standard libraries such as Intel’s math library lose precision to have better performance. Harrison proved a tight error bound of an algorithm for computing $sin\, x$ for $|x| \leqslant 2^{63}$ (which is slightly different from Intel’s sin implementation) in HOL Light [Harrison 2000b]. In general, the libimf documentation claims, without any formal proofs, that the maximum error in the routines is always below one ulp[5]. In this work, we have validated this claim fully automatically for log (for all valid inputs), sin (for inputs between $-\pi$ and π), and tan (for inputs between $13/128$ and $17\pi/64$). We are unaware of any prior technique that can prove such tight bounds for math libraries automatically.

The existing work closest to ours is our own previous paper [Lee et al. 2016], and this work builds on those techniques to achieve the results presented here. In [Lee et al. 2016], error bounds are proven by first decomposing the math.h implementations into simple expressions and then proving error bounds of those expressions using Mathematica. The primary research contribution of the 2016 paper is the first step which performs the decomposition, and we used a standard error analysis in the second step. The work in the present paper reuses the decomposition step and adds a novel automatic error analysis that leverages results about exact floating-point arithmetic systematically. No prior analysis, including the one of [Lee et al. 2016] and those mentioned below, uses all the exactness results we discussed in §5 and all of these would fail to prove that the error bounds are below one ulp for the benchmarks we consider.

Automatic tools that can provide formal guarantees on error bounds include Gappa [Daumas and Melquiond 2010], Fluctuat [Delmas et al. 2009; Goubault et al. 2007], MathSAT [Haller et al. 2012], Rosa [Darulova and Kuncak 2014], FPTaylor [Solovyev et al. 2015], and Astree [Blanchet et al. 2003; Miné 2012]. In contrast to [Lee et al. 2016], none of these provide support to bound the error between expressions in our core language and exact transcendentals. For example, these techniques do not handle bit-masking. Although some of these can handle some exactness results about floating-point, they do not provide a general framework like ours. For example, Gappa automatically applies some of the exactness results described in §5, but not all of them (*e.g.*, Dekker’s theorem (§5.3) and the refined $(1+\epsilon)$-property (§5.5) for $* \in \{\times, /\}$). Moreover, Gappa uses interval arithmetic to soundly bound the max/min of some expressions, when checking preconditions of exactness results. Interval arithmetic can often cause imprecision (because it does not preserve dependencies between variables) and fail to discharge the preconditions; our optimization-based technique is more precise.

There are techniques that check whether two floating-point programs produce exactly equivalent results [Collingbourne et al. 2011; Nötzli and Brown 2016]. These do not produce any bound on the maximum deviation between the implementations. Debugging tools such as [Barr et al. 2013; Benz et al. 2012; Chiang et al. 2014; Lakhotia et al. 2010] are complementary to our work and can

[5] See maxerror=1.0 at https://software.intel.com/en-us/cpp-compiler-18.0-developer-guide-and-reference-fimf-precision-qimf-precision

help detect incorrect implementations. In particular, [Fu and Su 2017] use optimization to find inputs that achieve high branch coverage. Other techniques that provide statistical (and not formal) guarantees include [Misailovic et al. 2011; Necula and Gulwani 2005; Schkufza et al. 2014].

9 CONCLUSION

A major source of imprecision in generic verification techniques for floating-point stems from modeling every floating-point operation as having a rounding error about which worst-case assumptions must be made. However, floating-point operations do not always introduce rounding errors. We identify floating-point computations that are exact and thus avoid introducing unneeded potential rounding errors into the modeling of those computations. Our main technical contribution is a reduction from the problem of checking whether an operation is exact to a set of mathematical optimization problems that are solved soundly and automatically by off-the-shelf computer algebra systems. We introduce transformations, also involving optimization problems, to control the size of our abstractions while maintaining precision. Our analysis successfully proves the correctness of x86 implementations from an industry standard math library. We are unaware of any prior formal correctness proofs of these widely used implementations.

A APPENDIX

A.1 Definition of Operations on Abstractions

In this subsection, assume that $\mathcal{A}_{\vec{\delta}}(x)$ denotes $a(x) + \sum_i b_i(x)\delta_i$ and δ_i ranges over $[-\Delta_i, \Delta_i]$.

$\boxed{\mathcal{A}_{\vec{\delta}}(x) \boxast \mathcal{A}'_{\vec{\delta}}(x)}$ $(* \in \{+, -, \times, /\})$

$$\begin{aligned}
\mathcal{A}_{\vec{\delta}}(x) \boxplus \mathcal{A}'_{\vec{\delta}}(x) &\triangleq \mathcal{A}_{\vec{\delta}}(x) + \mathcal{A}'_{\vec{\delta}}(x)\\
\mathcal{A}_{\vec{\delta}}(x) \boxminus \mathcal{A}'_{\vec{\delta}}(x) &\triangleq \mathcal{A}_{\vec{\delta}}(x) - \mathcal{A}'_{\vec{\delta}}(x)\\
\mathcal{A}_{\vec{\delta}}(x) \boxtimes \mathcal{A}'_{\vec{\delta}}(x) &\triangleq \mathit{linearize}(\mathcal{A}_{\vec{\delta}}(x) \times \mathcal{A}'_{\vec{\delta}}(x))\\
\mathcal{A}_{\vec{\delta}}(x) \boxslash \mathcal{A}'_{\vec{\delta}}(x) &\triangleq \mathcal{A}_{\vec{\delta}}(x) \boxtimes \mathit{inv}(\mathcal{A}'_{\vec{\delta}}(x))
\end{aligned}$$

$\mathit{linearize}(\cdot)$ and $\mathit{inv}(\cdot)$ are defined as:

$$\mathit{linearize}\left(a(x) + \sum_i b_i(x)\delta_i + \sum_{i,j} b_{i,j}(x)\delta_i\delta_j\right) \triangleq a(x) + \sum_i b_i(x)\delta_i + \sum_{i,j} b_{i,j}(x)\delta'_{i,j}$$

$$\mathit{inv}(\mathcal{A}_{\vec{\delta}}(x)) \triangleq \frac{1}{a(x)} + \frac{1}{a(x)}\delta'' \quad (\text{assumes } \Delta' < 1)$$

where $\delta'_{i,j} = \mathit{fresh}(\Delta_i\Delta_j)$ and $\delta'' = \mathit{fresh}(\frac{\Delta'}{1-\Delta'})$, and $\Delta' \in \mathbb{R}_{\geq 0}$ is computed as:

$$\Delta' = \sum_i \max_{x \in X} \left|\frac{b_i(x)}{a(x)}\right| \cdot \Delta_i$$

$\boxed{\mathcal{A}_{\vec{\delta}}(x) \boxast \delta'}$ $(* \in \{+, \times\})$

$$\begin{aligned}
\mathcal{A}_{\vec{\delta}}(x) \boxplus \delta' &\triangleq \mathcal{A}_{\vec{\delta}}(x) \boxplus \mathcal{A}'_{\vec{\delta}}(x), \quad \text{where } \mathcal{A}'_{\vec{\delta}}(x) = 0 + 1 \cdot \delta'\\
\mathcal{A}_{\vec{\delta}}(x) \boxtimes \delta' &\triangleq \mathcal{A}_{\vec{\delta}}(x) \boxtimes \mathcal{A}'_{\vec{\delta}}(x), \quad \text{where } \mathcal{A}'_{\vec{\delta}}(x) = 0 + 1 \cdot \delta'
\end{aligned}$$

$\boxed{\mathcal{A}_{\vec{\delta}}(x) \boxtimes (1 + \delta')}$

$$\mathcal{A}_{\vec{\delta}}(x) \boxtimes (1 + \delta') \triangleq a(x) + a(x)\delta' + \sum_{i \in R} b_i(x)\delta'_i + \sum_{i \notin R} (b_i(x)\delta_i + b_i(x)\delta''_i)$$

where δ' ranges over $[-\Delta', \Delta']$, $R = \{i : \text{preserve}(\delta_i) = \texttt{false}\}$, $\delta'_i = \mathit{fresh}(\Delta_i(1+\Delta'))$ $(i \in R)$, and $\delta''_i = \mathit{fresh}(\Delta_i\Delta')$ $(i \notin R)$.

$\boxed{\mathit{compress}(\mathcal{A}_{\vec{\delta}}(x))}$

$$\mathit{compress}(\mathcal{A}_{\vec{\delta}}(x)) \triangleq a(x) + a(x)\delta' + \sum_{i \notin R \cap S} b_i(x)\delta_i, \quad \text{where } \delta' = \mathit{fresh}\left(\sum_{i \in R \cap S} \gamma_i\right)$$

Here $R = \{i : \text{preserve}(\delta_i) = \texttt{false}\}$, and $\gamma_i \in \mathbb{R}_{\geqslant 0} \cup \{\infty\}$ and the set S are computed as:

$$\gamma_i = \max_{x \in X} \left|\frac{b_i(x)}{a(x)}\right| \cdot \Delta_i \qquad \text{and} \qquad S = \left\{i : \frac{\gamma_i}{\epsilon} \leqslant \tau\right\}$$

where $\tau \in \mathbb{R}_{\geqslant 0}$ is a constant.

A.2 Rules for Constructing Abstractions

Basic rules:

$$\frac{e \in \mathit{dom}(\mathcal{K}) \qquad \mathcal{K}(e) = (\mathcal{A}_{\vec{\delta}}, \_, \_, \_)}{(\mathcal{K}, e) \rhd (\mathcal{K}, \mathcal{A}_{\vec{\delta}})}\ \textsc{Load}'$$

$$\frac{}{(\mathcal{K}, c) \rhd (\mathcal{K}[c \mapsto (c, \texttt{false}, \sigma(c), \mu(c))], c)}\ \text{R1}' \qquad \frac{}{(\mathcal{K}, x) \rhd (\mathcal{K}[x \mapsto (x, \texttt{false}, 53, \mu(x))], x)}\ \text{R2}'$$

$$\frac{\begin{array}{ccc} & \mathcal{K}_1(e_1) = (\_, \_, \sigma_1, \mu_1) & \\ & \mathcal{K}_2(e_2) = (\_, \_, \sigma_2, \mu_2) & \\ (\mathcal{K}, e_1) \rhd (\mathcal{K}_1, \mathcal{A}_{1,\vec{\delta}}) & \sigma = \mathit{bound}\text{-}\sigma(*, \mathcal{A}_{1,\vec{\delta}}, \mathcal{A}_{2,\vec{\delta}}, \sigma_1, \sigma_2) & \\ (\mathcal{K}_1, e_2) \rhd (\mathcal{K}_2, \mathcal{A}_{2,\vec{\delta}}) & \mu = \mathit{bound}\text{-}\mu(*, \mathcal{A}_{1,\vec{\delta}}, \mathcal{A}_{2,\vec{\delta}}, \mu_1, \mu_2) & * \in \{+, -\} \end{array}}{(\mathcal{K}, e_1 \circledast e_2) \rhd (\mathcal{K}', \mathcal{A}'_{\vec{\delta}}),\ \text{where} \begin{cases} \delta' = \mathit{fresh}(\epsilon) \\ \mathcal{A}'_{\vec{\delta}} = \mathit{compress}((\mathcal{A}_{1,\vec{\delta}} \boxast \mathcal{A}_{2,\vec{\delta}}) \boxtimes (1 + \delta')) \\ \sigma' = \min\{\sigma, 53\},\ \mu' = \max\{\text{fl}^-(\mu), 2^{-1074}\} \\ \mathcal{K}' = \mathcal{K}_2[e_1 \circledast e_2 \mapsto (\mathcal{A}'_{\vec{\delta}}, \texttt{false}, \sigma', \mu')] \end{cases}}\ \text{R14}$$

$$\frac{\begin{array}{ccc} & \mathcal{K}_1(e_1) = (\_, \_, \sigma_1, \mu_1) & \\ & \mathcal{K}_2(e_2) = (\_, \_, \sigma_2, \mu_2) & \\ (\mathcal{K}, e_1) \rhd (\mathcal{K}_1, \mathcal{A}_{1,\vec{\delta}}) & \sigma = \mathit{bound}\text{-}\sigma(*, \mathcal{A}_{1,\vec{\delta}}, \mathcal{A}_{2,\vec{\delta}}, \sigma_1, \sigma_2) & * \in \{\times, /\} \\ (\mathcal{K}_1, e_2) \rhd (\mathcal{K}_2, \mathcal{A}_{2,\vec{\delta}}) & \mu = \mathit{bound}\text{-}\mu(*, \mathcal{A}_{1,\vec{\delta}}, \mathcal{A}_{2,\vec{\delta}}, \mu_1, \mu_2) & \mu \geqslant 2^{-1022} \end{array}}{(\mathcal{K}, e_1 \circledast e_2) \rhd (\mathcal{K}', \mathcal{A}'_{\vec{\delta}}),\ \text{where} \begin{cases} \delta' = \mathit{fresh}(\epsilon) \\ \mathcal{A}'_{\vec{\delta}} = \mathit{compress}((\mathcal{A}_{1,\vec{\delta}} \boxast \mathcal{A}_{2,\vec{\delta}}) \boxtimes (1 + \delta')) \\ \sigma' = \min\{\sigma, 53\},\ \mu' = \max\{\text{fl}^-(\mu), 2^{-1074}\} \\ \mathcal{K}' = \mathcal{K}_2[e_1 \circledast e_2 \mapsto (\mathcal{A}'_{\vec{\delta}}, \texttt{false}, \sigma', \mu')] \end{cases}}\ \text{R15}$$

$$\frac{\begin{array}{ccc} & \mathcal{K}_1(e_1) = (\_, \_, \sigma_1, \mu_1) & \\ & \mathcal{K}_2(e_2) = (\_, \_, \sigma_2, \mu_2) & \\ (\mathcal{K}, e_1) \rhd (\mathcal{K}_1, \mathcal{A}_{1,\vec{\delta}}) & \sigma = \mathit{bound}\text{-}\sigma(*, \mathcal{A}_{1,\vec{\delta}}, \mathcal{A}_{2,\vec{\delta}}, \sigma_1, \sigma_2) & * \in \{\times, /\} \\ (\mathcal{K}_1, e_2) \rhd (\mathcal{K}_2, \mathcal{A}_{2,\vec{\delta}}) & \mu = \mathit{bound}\text{-}\mu(*, \mathcal{A}_{1,\vec{\delta}}, \mathcal{A}_{2,\vec{\delta}}, \mu_1, \mu_2) & \mu < 2^{-1022} \end{array}}{(\mathcal{K}, e_1 \circledast e_2) \rhd (\mathcal{K}', \mathcal{A}'_{\vec{\delta}}),\ \text{where} \begin{cases} \delta' = \mathit{fresh}(\epsilon),\ \delta'' = \mathit{fresh}(\epsilon') \\ \mathcal{A}'_{\vec{\delta}} = \mathit{compress}((\mathcal{A}_{1,\vec{\delta}} \boxast \mathcal{A}_{2,\vec{\delta}}) \boxtimes (1 + \delta') \boxplus \delta'') \\ \sigma' = \min\{\sigma, 53\},\ \mu' = \max\{\text{fl}^-(\mu), 2^{-1074}\} \\ \mathcal{K}' = \mathcal{K}_2[e_1 \circledast e_2 \mapsto (\mathcal{A}'_{\vec{\delta}}, \texttt{false}, \sigma', \mu')] \end{cases}}\ \text{R3}'$$

$$\dfrac{\begin{array}{c} \mathcal{K}_1(e_1) = (\_,\_,\sigma_1,\mu_1) \\ (\mathcal{K}, \textit{bit-mask}(e_1,B)) \rhd (\mathcal{K}_1, \mathcal{A}_{1,\vec{\delta}}) \qquad \mu = \mathcal{E}(\textit{bit-mask}(x,B))(\text{fl}^-(\mu_1)) \qquad \mu \geqslant 2^{-1022} \end{array}}{(\mathcal{K}, \textit{bit-mask}(e_1,B)) \rhd (\mathcal{K}', \mathcal{A}'_{\vec{\delta}}),\ \text{where} \begin{cases} \delta' = \textit{fresh}(2^{-52+B}) \\ \mathcal{A}'_{\vec{\delta}} = \textit{compress}(\mathcal{A}_{1,\vec{\delta}} \boxtimes (1+\delta')) \\ \sigma' = \min\{\sigma_1, 53-B\},\ \mu' = \max\{\mu, 2^{-1074}\} \\ \mathcal{K}' = \mathcal{K}_1[\textit{bit-mask}(e_1,B) \mapsto (\mathcal{A}'_{\vec{\delta}}, \texttt{false}, \sigma', \mu')] \end{cases}}\ \text{R4}'\text{-1}$$

$$\dfrac{\begin{array}{c} \mathcal{K}_1(e_1) = (\_,\_,\sigma_1,\mu_1) \\ (\mathcal{K}, \textit{bit-mask}(e_1,B)) \rhd (\mathcal{K}_1, \mathcal{A}_{1,\vec{\delta}}) \qquad \mu = \mathcal{E}(\textit{bit-mask}(x,B))(\text{fl}^-(\mu_1)) \qquad \mu < 2^{-1022} \end{array}}{(\mathcal{K}, \textit{bit-mask}(e_1,B)) \rhd (\mathcal{K}', \mathcal{A}'_{\vec{\delta}}),\ \text{where} \begin{cases} \delta' = \textit{fresh}(2^{-52+B}),\ \delta'' = \textit{fresh}(2^{-1074+B}) \\ \mathcal{A}'_{\vec{\delta}} = \textit{compress}(\mathcal{A}_{1,\vec{\delta}} \boxtimes (1+\delta') \boxplus \delta'') \\ \sigma' = \min\{\sigma_1, 53-B\},\ \mu' = \max\{\mu, 2^{-1074}\} \\ \mathcal{K}' = \mathcal{K}_1[\textit{bit-mask}(e_1,B) \mapsto (\mathcal{A}'_{\vec{\delta}}, \texttt{false}, \sigma', \mu')] \end{cases}}\ \text{R4}'\text{-2}$$

Rules for simple exact operations:

$$\dfrac{\begin{array}{l} (\mathcal{K}, e_1) \rhd (\mathcal{K}_1, \mathcal{A}_{1,\vec{\delta}}) \\ (\mathcal{K}_1, e_2) \rhd (\mathcal{K}_2, 0) \qquad \mathcal{K}_1(e_1) = (\_,\_,\sigma_1,\mu_1) \qquad * \in \{+,-\} \end{array}}{(\mathcal{K}, e_1 \circledast e_2) \rhd (\mathcal{K}', \mathcal{A}_{1,\vec{\delta}}),\ \text{where}\ \mathcal{K}' = \mathcal{K}_2[e_1 \circledast e_2 \mapsto (\mathcal{A}_{1,\vec{\delta}}, \texttt{true}, \sigma_1, \mu_1)]}\ \text{R5}'$$

$$\dfrac{\begin{array}{ccc} & \mathcal{K}_1(e_1) = (\_,\_,\sigma_1,\mu_1) & \\ & \mathcal{K}_2(e_2) = (\_,\_,\sigma_2,\mu_2) & \\ (\mathcal{K}, e_1) \rhd (\mathcal{K}_1, \mathcal{A}_{1,\vec{\delta}}) & \sigma = \textit{bound-}\sigma(\times, \mathcal{A}_{1,\vec{\delta}}, \mathcal{A}_{2,\vec{\delta}}, \sigma_1, \sigma_2) & \exists n \in \mathbb{Z}. \forall x \in X.\ \mathcal{A}_{2,\vec{\delta}}(x) = 2^n \\ (\mathcal{K}_1, e_2) \rhd (\mathcal{K}_2, \mathcal{A}_{2,\vec{\delta}}) & \mu = \textit{bound-}\mu(\times, \mathcal{A}_{1,\vec{\delta}}, \mathcal{A}_{2,\vec{\delta}}, \mu_1, \mu_2) & n \geqslant -1075 - \text{expnt}(\mu_1) + \sigma_1 \end{array}}{(\mathcal{K}, e_1 \otimes e_2) \rhd (\mathcal{K}', \mathcal{A}'_{\vec{\delta}}),\ \text{where} \begin{cases} \mathcal{A}'_{\vec{\delta}} = \mathcal{A}_{1,\vec{\delta}} \times 2^n \\ \sigma' = \min\{\sigma, 53\},\ \mu' = \max\{\mu, 2^{-1074}\} \\ \mathcal{K}' = \mathcal{K}_2[e_1 \otimes e_2 \mapsto (\mathcal{A}'_{\vec{\delta}}, \texttt{true}, \sigma', \mu')] \end{cases}}\ \text{R6}'$$

$$\dfrac{\begin{array}{l} (\mathcal{K}, e_1) \rhd (\mathcal{K}_1, c_1) \\ (\mathcal{K}_1, e_2) \rhd (\mathcal{K}_2, c_2) \qquad c' = c_1 \circledast c_2 \qquad * \in \{+,-,\times,/\} \end{array}}{(\mathcal{K}, e_1 \circledast e_2) \rhd (\mathcal{K}', c'),\ \text{where}\ \mathcal{K}' = \mathcal{K}_2[e_1 \circledast e_2 \mapsto (c', c' == c_1 * c_2, \sigma(c'), \mu(c'))]}\ \text{R7}'$$

Rules for applying Sterbenz's theorem:

$$\dfrac{\begin{array}{ccc} & \mathcal{K}_1(e_1) = (\_,\_,\sigma_1,\mu_1) & \\ & \mathcal{K}_2(e_2) = (\_,\_,\sigma_2,\mu_2) & \\ (\mathcal{K}, e_1) \rhd (\mathcal{K}_1, \mathcal{A}_{1,\vec{\delta}}) & \sigma = \textit{bound-}\sigma(-, \mathcal{A}_{1,\vec{\delta}}, \mathcal{A}_{2,\vec{\delta}}, \sigma_1, \sigma_2) & \min_{x,\vec{\delta}}(\mathcal{A}_{2,\vec{\delta}} - \frac{1}{2}\mathcal{A}_{1,\vec{\delta}}) \geqslant 0 \\ (\mathcal{K}_1, e_2) \rhd (\mathcal{K}_2, \mathcal{A}_{2,\vec{\delta}}) & \mu = \textit{bound-}\mu(-, \mathcal{A}_{1,\vec{\delta}}, \mathcal{A}_{2,\vec{\delta}}, \mu_1, \mu_2) & \max_{x,\vec{\delta}}(\mathcal{A}_{2,\vec{\delta}} - 2\mathcal{A}_{1,\vec{\delta}}) \leqslant 0 \end{array}}{(\mathcal{K}, e_1 \ominus e_2) \rhd (\mathcal{K}', \mathcal{A}'_{\vec{\delta}}),\ \text{where} \begin{cases} \mathcal{A}'_{\vec{\delta}} = \textit{compress}(\mathcal{A}_{1,\vec{\delta}} \boxminus \mathcal{A}_{2,\vec{\delta}}) \\ \sigma' = \min\{\sigma, 53\},\ \mu' = \max\{\mu, 2^{-1074}\} \\ \mathcal{K}' = \mathcal{K}_2[e_1 \ominus e_2 \mapsto (\mathcal{A}'_{\vec{\delta}}, \texttt{true}, \sigma', \mu')] \end{cases}}\ \text{R8}'$$

$$\frac{\begin{array}{ccc} & \mathcal{K}_1(e_1) = (\_,\_,\sigma_1,\mu_1) & \\ & \mathcal{K}_2(e_2) = (\_,\_,\sigma_2,\mu_2) & \\ (\mathcal{K},e_1) \triangleright (\mathcal{K}_1,\mathcal{A}_{1,\vec{\delta}}) & \sigma = \mathit{bound}\text{-}\sigma(-,\mathcal{A}_{1,\vec{\delta}},\mathcal{A}_{2,\vec{\delta}},\sigma_1,\sigma_2) & \min_{x,\vec{\delta}}(\mathcal{A}_{2,\vec{\delta}} - 2\mathcal{A}_{1,\vec{\delta}}) \geqslant 0 \\ (\mathcal{K}_1,e_2) \triangleright (\mathcal{K}_2,\mathcal{A}_{2,\vec{\delta}}) & \mu = \mathit{bound}\text{-}\mu(-,\mathcal{A}_{1,\vec{\delta}},\mathcal{A}_{2,\vec{\delta}},\mu_1,\mu_2) & \max_{x,\vec{\delta}}(\mathcal{A}_{2,\vec{\delta}} - \tfrac{1}{2}\mathcal{A}_{1,\vec{\delta}}) \leqslant 0 \end{array}}{(\mathcal{K},e_1 \ominus e_2) \triangleright (\mathcal{K}',\mathcal{A}'_{\vec{\delta}}),\ \text{where} \begin{cases} \mathcal{A}'_{\vec{\delta}} = \mathit{compress}(\mathcal{A}_{1,\vec{\delta}} \boxminus \mathcal{A}_{2,\vec{\delta}}) \\ \sigma' = \min\{\sigma,53\},\ \mu' = \max\{\mu,2^{-1074}\} \\ \mathcal{K}' = \mathcal{K}_2[e_1 \ominus e_2 \mapsto (\mathcal{A}'_{\vec{\delta}},\texttt{true},\sigma',\mu')] \end{cases}}\ \text{R9}'$$

Rules for applying Dekker's theorem:

$$\frac{\begin{array}{ccc} & \mathcal{K}_1(e_1) = (\_,\_,\sigma_1,\mu_1) & \\ & \mathcal{K}_2(e_2) = (\_,\_,\sigma_2,\mu_2) & \\ (\mathcal{K},e_1) \triangleright (\mathcal{K}_1,\mathcal{A}_{1,\vec{\delta}}) & \sigma = \mathit{bound}\text{-}\sigma(+,\mathcal{A}_{1,\vec{\delta}},\mathcal{A}_{2,\vec{\delta}},\sigma_1,\sigma_2) & \\ (\mathcal{K}_1,e_2) \triangleright (\mathcal{K}_2,\mathcal{A}_{2,\vec{\delta}}) & \mu = \mathit{bound}\text{-}\mu(+,\mathcal{A}_{1,\vec{\delta}},\mathcal{A}_{2,\vec{\delta}},\mu_1,\mu_2) & \mathsf{hasDekker}(e_1 \oplus e_2) \end{array}}{(\mathcal{K},e_1 \oplus e_2) \triangleright (\mathcal{K}',\mathcal{A}'_{\vec{\delta}}),\ \text{where} \begin{cases} \delta' = \mathit{fresh}(\epsilon,\texttt{true}) \\ \mathcal{A}'_{\vec{\delta}} = \mathit{compress}((\mathcal{A}_{1,\vec{\delta}} \boxplus \mathcal{A}_{2,\vec{\delta}}) \boxtimes (1+\delta')) \\ \sigma' = \min\{\sigma,53\},\ \mu' = \max\{\mathrm{fl}^{-}(\mu),2^{-1074}\} \\ \mathcal{K}' = \mathcal{K}_2[e_1 \oplus e_2 \mapsto (\mathcal{A}'_{\vec{\delta}},\texttt{false}\langle\delta'\rangle,\sigma',\mu')] \end{cases}}\ \text{R10}'$$

$$\frac{\begin{array}{ccc} & \mathcal{K}_1(e_1) = (\mathcal{A}_{1,\vec{\delta}},\_,\_,\_) & \mathcal{K}_1(e_1 \oplus e_2) = (\_,\texttt{false}\langle\delta'\rangle,\_,\_) \\ (\mathcal{K},e_1 \oplus e_2) \triangleright (\mathcal{K}_1,\_) & \mathcal{K}_1(e_2) = (\mathcal{A}_{2,\vec{\delta}},\_,\_,\_) & \min_{x,\vec{\delta}} |\mathcal{A}_{1,\vec{\delta}}| \geqslant \max_{x,\vec{\delta}} |\mathcal{A}_{2,\vec{\delta}}| \end{array}}{(\mathcal{K},e_1 \oplus e_2 \ominus e_1 \ominus e_2) \triangleright (\mathcal{K}',\mathcal{A}'_{\vec{\delta}}),\ \text{where} \begin{cases} \mathcal{A}'_{\vec{\delta}} = \mathit{compress}((\mathcal{A}_{1,\vec{\delta}} \boxplus \mathcal{A}_{2,\vec{\delta}}) \boxtimes \delta') \\ \mathcal{K}' = \mathcal{K}_1[e_1 \oplus e_2 \ominus e_1 \ominus e_2 \mapsto (\mathcal{A}'_{\vec{\delta}},\texttt{false},53,2^{-1074})] \end{cases}}\ \text{R11}'$$

$$\frac{(\mathcal{K},e_1 \oplus e_2) \triangleright (\mathcal{K}_1,\_) \qquad \mathcal{K}_1(e_1 \oplus e_2) = (\_,\texttt{true},\_,\_)}{(\mathcal{K},e_1 \oplus e_2 \ominus e_1 \ominus e_2) \triangleright (\mathcal{K}',0),\ \text{where}\ \mathcal{K}' = \mathcal{K}_1[e_1 \oplus e_2 \ominus e_1 \ominus e_2 \mapsto (0,\texttt{true},0,\infty)]}\ \text{R12}'$$

Rule for using $\sigma(\cdot)$:

$$\frac{\begin{array}{ccc} & \mathcal{K}_1(e_1) = (\_,\_,\sigma_1,\mu_1) & \\ & \mathcal{K}_2(e_2) = (\_,\_,\sigma_2,\mu_2) & \\ (\mathcal{K},e_1) \triangleright (\mathcal{K}_1,\mathcal{A}_{1,\vec{\delta}}) & \sigma = \mathit{bound}\text{-}\sigma(*,\mathcal{A}_{1,\vec{\delta}},\mathcal{A}_{2,\vec{\delta}},\sigma_1,\sigma_2) & * \in \{+,-,\times,/\} \\ (\mathcal{K}_1,e_2) \triangleright (\mathcal{K}_2,\mathcal{A}_{2,\vec{\delta}}) & \mu = \mathit{bound}\text{-}\mu(*,\mathcal{A}_{1,\vec{\delta}},\mathcal{A}_{2,\vec{\delta}},\mu_1,\mu_2) & \sigma \leqslant 53 \end{array}}{(\mathcal{K},e_1 \circledast e_2) \triangleright (\mathcal{K}',\mathcal{A}'_{\vec{\delta}}),\ \text{where} \begin{cases} \mathcal{A}'_{\vec{\delta}} = \mathit{compress}(\mathcal{A}_{1,\vec{\delta}} \boxast \mathcal{A}_{2,\vec{\delta}}) \\ \mu' = \max\{\mu,2^{-1074}\} \\ \mathcal{K}' = \mathcal{K}_2[e_1 \circledast e_2 \mapsto (\mathcal{A}'_{\vec{\delta}},\texttt{true},\sigma,\mu')] \end{cases}}\ \text{R13}'$$

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful comments. Wonyeol Lee was supported by Samsung Scholarship. This work was also supported by NSF grants CCF-1160904 and CCF-1409813.

REFERENCES

Earl T. Barr, Thanh Vo, Vu Le, and Zhendong Su. 2013. Automatic detection of floating-point exceptions. In *POPL*. 549–560.

Florian Benz, Andreas Hildebrandt, and Sebastian Hack. 2012. A dynamic program analysis to find floating-point accuracy problems. In *PLDI*. 453–462.

Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2003. A static analyzer for large safety-critical software. In *PLDI*. 196–207.

J H Chen, A Choudhary, B de Supinski, M DeVries, E R Hawkes, S Klasky, W K Liao, K L Ma, J Mellor-Crummey, N Podhorszki, R Sankaran, S Shende, and C S Yoo. 2009. Terascale direct numerical simulations of turbulent combustion using S3D. *Computational Science and Discovery* (2009), 015001.

Wei-Fan Chiang, Ganesh Gopalakrishnan, Zvonimir Rakamaric, and Alexey Solovyev. 2014. Efficient search for inputs causing high floating-point errors. In *PPoPP*. 43–52.

Peter Collingbourne, Cristian Cadar, and Paul H. J. Kelly. 2011. Symbolic crosschecking of floating-point and SIMD code. In *EuroSys*. 315–328.

Catherine Daramy-Loirat, David Defour, Florent de Dinechin, Matthieu Gallet, Nicolas Gast, and Jean-Michel Muller. 2005. CR-Libm, a library of correctly rounded elementary functions in double-precision. Available at http://lipforge.ens-lyon.fr/www/crlibm.

Eva Darulova and Viktor Kuncak. 2014. Sound Compilation of Reals. In *POPL*. 235–248.

Marc Daumas and Guillaume Melquiond. 2010. Certification of bounds on expressions involving rounded operators. *ACM Trans. Math. Software* 37, 1 (2010), 2:1–2:20.

Florent de Dinechin, Christoph Quirin Lauter, and Guillaume Melquiond. 2011. Certifying the Floating-Point Implementation of an Elementary Function Using Gappa. *IEEE Trans. Computers* 60, 2 (2011), 242–253.

T. J. Dekker. 1971. A Floating-point Technique for Extending the Available Precision. *Numer. Math.* 18, 3 (1971), 224–242.

David Delmas, Eric Goubault, Sylvie Putot, Jean Souyris, Karim Tekkal, and Franck Védrine. 2009. Towards an Industrial Use of FLUCTUAT on Safety-Critical Avionics Software. In *FMICS*. 53–69.

Zhoulai Fu and Zhendong Su. 2017. Achieving high coverage for floating-point code via unconstrained programming. In *PLDI*. 306–319.

David Goldberg. 1991. What Every Computer Scientist Should Know About Floating Point Arithmetic. *Comput. Surveys* 23, 1 (1991), 5–48.

Eric Goubault and Sylvie Putot. 2005. Weakly relational domains for floating-point computation analysis. In *NSAD*.

Eric Goubault, Sylvie Putot, Philippe Baufreton, and Jean Gassino. 2007. Static Analysis of the Accuracy in Control Systems: Principles and Experiments. In *FMICS*. 3–20.

Leopold Haller, Alberto Griggio, Martin Brain, and Daniel Kroening. 2012. Deciding floating-point logic with systematic abstraction. In *FMCAD*. 131–140.

John Harrison. 1999. A machine-checked theory of floating-point arithmetic. In *TPHOLs*. 113–130.

John Harrison. 2000a. Floating-point verification in HOL Light: the exponential function. *Formal Methods in System Design* 16, 3 (2000), 271–305.

John Harrison. 2000b. Formal verification of floating point trigonometric functions. In *FMCAD*. 217–233.

William Kahan. 2004. A logarithm too clever by half. Available at http://http.cs.berkeley.edu/~wkahan/LOG10HAF.TXT.

Kiran Lakhotia, Nikolai Tillmann, Mark Harman, and Jonathan de Halleux. 2010. FloPSy - Search-Based Floating Point Constraint Solving for Symbolic Execution. In *ICTSS*. 142–157.

Wonyeol Lee, Rahul Sharma, and Alex Aiken. 2016. Verifying Bit Manipulations of Floating-Point. In *PLDI*. 70–84.

Guillaume Melquiond. 2012. Floating-point arithmetic in the Coq system. *Inf. Comput.* 216 (2012), 14–23.

Antoine Miné. 2012. Abstract Domains for Bit-Level Machine Integer and Floating-point Operations. In *ATx/WInG*. 55–70.

Sasa Misailovic, Daniel M. Roy, and Martin C. Rinard. 2011. Probabilistically Accurate Program Transformations. In *SAS*. 316–333.

Jean-Michel Muller. 2005. On the definition of ulp(x). Available at http://ljk.imag.fr/membres/Carine.Lucas/TPScilab/JMMuller/ulp-toms.pdf.

George C. Necula and Sumit Gulwani. 2005. Randomized Algorithms for Program Analysis and Verification. In *CAV*. 1.

Andres Nötzli and Fraser Brown. 2016. LifeJacket: verifying precise floating-point optimizations in LLVM. In *SOAP@PLDI*. 24–29.

Tahina Ramananandro, Paul Mountcastle, Benoît Meister, and Richard Lethin. 2016. A unified Coq framework for verifying C programs with floating-point computations. In *CPP*. 15–26.

Eric Schkufza, Rahul Sharma, and Alex Aiken. 2014. Stochastic Optimization of Floating-Point Programs using Tunable Precision. In *PLDI*. 53–64.

Alexey Solovyev, Charles Jacobsen, Zvonimir Rakamaric, and Ganesh Gopalakrishnan. 2015. Rigorous Estimation of Floating-Point Round-off Errors with Symbolic Taylor Expansions. In *FM*. 532–550.

Pat H. Sterbenz. 1973. *Floating-point computation.* Prentice Hall, Englewood Cliffs, NJ.