Numerical Fuzz: A Type System for Rounding Error Analysis

ARIEL E. KELLISON, Cornell University, USA
JUSTIN HSU, Cornell University, USA

Algorithms operating on real numbers are implemented as floating-point computations in practice, but floating-point operations introduce *roundoff errors* that can degrade the accuracy of the result. We propose $\Lambda_{\mathbf{num}}$, a functional programming language with a type system that can express quantitative bounds on roundoff error. Our type system combines a sensitivity analysis, enforced through a linear typing discipline, with a novel graded monad to track the accumulation of roundoff errors. We prove that our type system is sound by relating the denotational semantics of our language to the exact and floating-point operational semantics.

To demonstrate our system, we instantiate $\Lambda_{\mathbf{num}}$ with error metrics proposed in the numerical analysis literature and we show how to incorporate rounding operations that faithfully model aspects of the IEEE 754 floating-point standard. To show that $\Lambda_{\mathbf{num}}$ can be a useful tool for automated error analysis, we develop a prototype implementation for $\Lambda_{\mathbf{num}}$ that infers error bounds that are competitive with existing tools, while often running significantly faster. Finally, we consider semantic extensions of our graded monad to bound error under more complex rounding behaviors, such as non-deterministic and randomized rounding.

CCS Concepts: • **Mathematics of computing** → **Numerical analysis**; • **General and reference** → **Verification**; • **Software and its engineering** → *Functional languages*.

Additional Key Words and Phrases: Floating point, Roundoff error, Linear type systems

1 INTRODUCTION

Floating-point numbers serve as discrete, finite approximations of continuous real numbers. Since computation on floating-point numbers is designed to approximate a computation on ideal real numbers, a key goal is reducing the *roundoff error*: the difference between the floating-point and the ideal results. To address this challenge, researchers in numerical analysis have developed techniques to measure, analyze, and ultimately reduce the approximation error in floating-point computations.

Prior work: formal methods for numerical software. While numerical error analysis provides a well-established set of tools for bounding roundoff error, it requires manual effort and tedious calculation. To automate this process, researchers have developed verification methods based on *abstract interpretation* and *optimization*. In the first approach, the analysis approximates floating-point numbers with roundoff error by *intervals* of real numbers, which are propagated through the computation. In the second approach, the analysis approximates the floating-point program by a more well-behaved function (e.g., a polynomial), and then uses global optimization to find the maximum error between the approximation and the ideal computation over all possible realizations of the rounding error.

While these tools are effective, they share some drawbacks. First, there is the issue of limited scalability: analyses that rely on global optimization are not compositional, and analyses based on interval arithmetic are compositional, but when the intervals are composed naively, the analysis produces bounds that are too large to be useful in practice. Second, existing tools largely focus on small simple expressions, such as straight-line programs, and it is unclear how to extend existing methods to more full-featured programming languages.

Authors' addresses: Ariel E. Kellison, Cornell University, Ithaca, USA, ak2485@cornell.edu; Justin Hsu, Cornell University, Ithaca, USA, justin@cs.cornell.edu.

Analyzing floating-point error: basics and challenges. To get a glimpse of the challenges in analyzing roundoff error, we begin with some basics. At a high level, floating-point numbers are a finite subset of the continuous real numbers. Arithmetic operations (e.g., addition, multiplication) have floating-point counterparts, which are specified following a common principle: the output of a floating-point operation applied to some arguments is the result of the *ideal* operation, followed by *rounding* to a representable floating-point number. In symbols:

$$\widetilde{op}(x, y) \triangleq \widetilde{op(x, y)}$$

where the tilde denotes the approximate operation and rounded value, respectively.

Though simple to state, this principle leads to several challenges in analyzing floating point error. First, many properties of exact arithmetic do not hold for floating-point arithmetic. For example, floating-point addition is not associative: $(x\tilde{+}y)\tilde{+}z \neq x\tilde{+}(y\tilde{+}z)$. Second, the floating point error accumulates in complex ways through the computation. For instance, the floating point error of a computation cannot be directly estimated from just the *number* of rounding operations—the details of the specific computation are important, since some operations may amplify error, while other operations may reduce error.

Our work: a type system for error analysis. To address these challenges, we propose $\Lambda_{\mathbf{num}}$, a novel type system for error analysis. Our approach is inspired by the specification of floating-point operations as an exact (ideal) operation, followed by a rounding step. The key idea is to separate the error analysis into two distinct components: a *sensitivity* analysis, which describes how errors propagate through the computation in the absence of rounding, and a *rounding* analysis, which tracks how errors accumulate due to rounding the results of operations.

Our type system is based on Fuzz [47], a family of bounded-linear type systems for sensitivity analysis originally developed for verifying differential privacy. To track errors due to rounding, we extend the language with a graded monadic type $M_u\tau$. Intuitively, $M_u\tau$ is the type of computations that produce τ while possibly performing rounding, and u is a real constant that upper-bounds the rounding error. In this way, we view rounding as an *effect*, and model rounding computations with a monadic type like other kinds of computational effects [42]. We interpret our monadic type as a novel graded monad on the category of metric spaces, which may be of independent interest.

As far as we know, our work is the first type system to provide bounds on roundoff error. Our type-based approach has several advantages compared to prior work. First, our system can be instantiated to handle different kinds of error metrics; our leading application bounds the *relative* error, using a metric due to Olver [44]. Second, $\Lambda_{\mathbf{num}}$ is an expressive, higher-order language; by using a primitive operation for rounding, we are able to precisely describe where rounding is applied. Finally, the analysis in $\Lambda_{\mathbf{num}}$ is compositional and does not require global optimization.

Outline of paper. After presenting background on floating-point arithmetic and giving an overview of our system (Section 2), we present our technical contributions:

- The design of $\Lambda_{\mathbf{num}}$, a language and type system for error analysis (Section 3).
- A denotational semantics for $\Lambda_{\mathbf{num}}$, along with metatheoretic properties establishing soundness of the error bound. A key ingredient is the *neighborhood monad*, a novel monad on the category of metric spaces (Section 4).
- A range of case studies showing how to instantiate our language for different kinds of error analyses and rounding operations described by the floating-point standard. We demonstrate how to use our system to establish bounds for various programs through typing (Section 5).
- A prototype implementation for $\Lambda_{\mathbf{num}}$, capable of inferring types capturing roundoff error bounds. We translate a variety of floating-point benchmarks into $\Lambda_{\mathbf{num}}$, and show that our

implementation infers error bounds that are competitive with error bounds produced by other tools, while often running substantially faster (Section 6).
- Extensions of the neighborhood monad to model more complex rounding behavior, e.g., rounding with underflows/overflows, non-deterministic rounding, state-dependent rounding, and probabilistic rounding (Section 7).

Finally, we discuss related work (Section 8) and conclude with future directions (Section 9).

2 A TOUR OF $\Lambda_{\mathbf{num}}$

2.1 Floating-Point Arithmetic

To set the stage, we first recall some basic properties of floating-point arithmetic. For the interested reader, we point to excellent expositions by Goldberg [26], Higham [31], and Boldo et al. [8].

Floating-Point Number Systems. A floating-point number x in a floating-point number system $\mathbb{F} \subseteq \mathbb{R}$ has the form

$$x = (-1)^s \cdot m \cdot \beta^{e-p+1}, \tag{1}$$

where $\beta \in \{b \in \mathbb{N} \mid b \geq 2\}$ is the *base*, $p \in \{prec \in \mathbb{N} \mid prec \geq 2\}$ is the *precision*, $m \in \mathbb{N} \cap [0, \beta^p)$ is the *significand*, $e \in \mathbb{Z} \cap [\text{emin}, \text{emax}]$ is the *exponent*, and $s \in \{0, 1\}$ is the *sign* of x. For IEEE binary64 (double-precision), $p = 53$ and emax = 1023; for binary32, $p = 24$ and emax = 127.

Many real numbers cannot be represented exactly in a floating-point format. For example, the number 0.1 cannot be exactly represented in binary64. Furthermore, the result of most elementary operations on floating-point numbers cannot be represented exactly and must be *rounded* back to a representable value, leading to one of the most distinctive features of floating-point arithmetic: roundoff error.

Rounding Operators. Given a real number x and a floating point format $\mathbb{F}$, a *rounding operator* $\rho : \mathbb{R} \to \mathbb{F}$ is a function that takes x and returns a (nearby) floating-point number. The IEEE standard specifies that the basic arithmetic operations $(+, -, *, \div, \sqrt{})$ behave as if they first computed a correct, infinitely precise result, and then rounded the result using one of four rounding functions (referred to as modes): round towards $+\infty$, round towards $-\infty$, round towards 0, and round towards nearest (with defined tie-breaking schemes).

The Standard Model. By clearly defining floating-point formats and rounding functions, the floating-point standard provides a mathematical model for reasoning about roundoff error: If we write op for an ideal, exact arithmetic operation, and $\widetilde{op}$ for the correctly rounded, floating-point version of op, then then for any floating-point numbers x and y we have [31]

$$x \mathbin{\widetilde{op}} y \triangleq \rho(x \mathbin{op} y) = (x \mathbin{op} y)(1 + \delta), \quad |\delta| \leq u, \quad op \in \{+, -, *, \div\}, \tag{2}$$

where ρ is an IEEE rounding operator and u is the *unit roundoff*, which is upper bounded by 2^{1-p} for a binary floating-point format with precision p. Equation (2) is only valid in the absence of underflow and overflow. We discuss how $\Lambda_{\mathbf{num}}$ can handle underflow and overflow in in Section 7.

Absolute and Relative Error. The most common measures of the accuracy of a floating-point approximation $\tilde{x}$ to an exact value x are *absolute error* (er_{abs}) and *relative error* (er_{rel}):

$$er_{abs}(x, \tilde{x}) = |\tilde{x} - x| \quad \text{and} \quad er_{rel}(x, \tilde{x}) = |(\tilde{x} - x)/x| \quad \text{if } x \neq 0. \tag{3}$$

From Equation (2), we see that the relative error of the basic floating-point operations is at most unit roundoff.

The relative and absolute error do not apply uniformly to all values: the absolute error is well-behaved for small values, while the relative error is well-behaved for large values. Alternative error

measures that can uniformly represent floating-point error on both large and small values are the *units in the last place* (ULP) error, which measures the number of floating-point values between an approximate and exact value, and its logarithm, *bits of error* [16]:

$$er_{\text{ULP}}(x, \tilde{x}) = |\mathbb{F} \cap [\min(x, \tilde{x}), \max(x, \tilde{x})]| \quad \text{and} \quad er_{bits}(x, \tilde{x}) = \log_2 er_{\text{ULP}}(x, \tilde{x}). \tag{4}$$

While static analysis tools that provide sound, worst-case error bound guarantees for floating-point programs compute relative or absolute error bounds (or both), the ULP error and its logarithm are often used in tools that optimize either the performance or accuracy of floating-point programs, like Herbie [45] and Stoke [48].

Propagation of Rounding Errors. In addition to bounding the rounding error produced by a floating-point computation, a comprehensive rounding error analysis must also quantify how a computation propagates rounding errors from inputs to outputs. The tools Rosa [18, 19], Fluctuat [27], and SATIRE [20] account for the propagation of rounding errors using Taylor-approximations, abstract interpretation, and automatic differentiation, respectively. In our work, we take a different approach: our language $\Lambda_{\mathbf{num}}$ tracks the propagation of rounding errors using a *sensitivity type system.*

2.2 Sensitivity Type Systems: An Overview

The core of our type system is based on Fuzz [47], a family of linear type systems. The central idea in Fuzz is that each type τ can be viewed as a metric space with a *metric* d_τ, a notion of distance on values of type τ. Then, function types describe functions of bounded sensitivity.

Definition 2.1. A function $f : X \to Y$ between metric spaces is said to be *c-sensitive* (or *Lipschitz continuous* with constant c) *iff* $d_Y(f(x), f(y)) \leq c \cdot d_X(x, y)$ for all $x, y \in X$.

In other words, a function is c-sensitive if it can magnify distances between inputs by a factor of at most c. In Fuzz, and in our system, the type $\tau \multimap \sigma$ describes functions that are *non-expansive*, or 1-sensitive functions. Intuitively, varying the input of a non-expansive function by distance δ cannot change the output by more than distance δ. Functions that are r-sensitive for some constant r are captured by the type $!_r\tau \multimap \sigma$; the type $!_r\tau$ scales the metric of τ by a factor of r.

To get a sense of how the type system works, we first introduce a metric on real numbers proposed by Olver [44] to capture relative error in numerical analysis.

Definition 2.2 (The Relative Precision (RP) Metric). Let $\tilde{x}$ and x be nonzero real numbers of the same sign. Then the relative precision (RP) of $\tilde{x}$ as an approximation to x is given by

$$RP(x, \tilde{x}) = |ln(x/\tilde{x})|. \tag{5}$$

While Definition 2.2 is a true metric, satisfying the usual axioms of zero-self distance, symmetry, and the triangle inequality, the relative error (Equation (3)) and the ULP error (Equation (4)) are not. Rewriting Definition 2.2 and the relative error as follows, and by considering the Taylor expansion of the exponential function, we can see that the relative precision is a close approximation to the relative error so long as $\delta \ll 1$:

$$er_{rel}(x, \tilde{x}) = |\delta|; \quad \tilde{x} = (1 + \delta)x, \tag{6}$$

$$RP(x, \tilde{x}) = |\delta|; \quad \tilde{x} = e^{\delta}x. \tag{7}$$

Moreover, if $\tilde{x}$ as an approximation to x of relative precision $0 \leq \alpha < 1$, then $\tilde{x}$ approximates x with relative error [cf. 44, eq 3.28 95]

$$\epsilon = e^{\alpha} - 1 \leq \alpha/(1 - \alpha). \tag{8}$$

In $\Lambda_{\mathbf{num}}$, we can write a function **pow2** that squares its argument and assign it the following type:

$$\mathbf{pow2} \triangleq \lambda x.\ \mathbf{mul}\ (x, x) : !_2\mathbf{num} \multimap \mathbf{num}.$$

The type **num** is the numeric type in $\Lambda_{\mathbf{num}}$; for now, we can think of it as just the ideal real numbers $\mathbb{R}$. The type $!_2\mathbf{num} \multimap \mathbf{num}$ states that **pow2** is 2-sensitive under the RP metric. More generally, this type reflects that the sensitivity of a function to its inputs is dependent on how many times the input is used in the function body. Now, spelling all of this out, if we have two inputs v and $v \cdot e^{\delta}$ at distance δ under the RP metric, then applying **pow2** leads to outputs v^2 and $(v \cdot e^{\delta})^2 = v^2 \cdot e^{2\cdot\delta}$, which are at distance (at most) $2 \cdot \delta$ under the RP metric.

2.3 Roundoff Error Analysis in $\Lambda_{\mathbf{num}}$: A Motivating Example

So far, we have not considered roundoff error: **pow2** simply squares its argument without performing any rounding. Next, we give an idea of how rounding is modeled in $\Lambda_{\mathbf{num}}$, and how sensitivity interacts with roundoff error.

The purpose of a rounding error analysis is to derive an a priori bound on the effects of rounding errors on an algorithm [31]. Suppose we are tasked with performing a rounding error analysis on a function **pow2′**, which squares a real number and rounds the result. Using the standard model for floating-point arithmetic (Equation (2)), the analysis is simple: the result of the function is

$$\mathbf{pow2}'(x) = \rho(x * x) = (x * x)(1 + \epsilon), \quad |\epsilon| \leq u, \tag{9}$$

and the relative error is bounded by the unit roundoff, u. Our insight is that a type system can be used to perform this analysis, by modeling rounding as an *error producing* effectful operation.

To see how this works, the function **pow2′** can be defined in $\Lambda_{\mathbf{num}}$ as follows:

$$\mathbf{pow2}' \triangleq \lambda x.\ \mathbf{rnd}\ (\mathbf{mul}\ (x, x)) : !_2\mathbf{num} \multimap M_u\mathbf{num}.$$

Here, **rnd** is an effectful operation that applies rounding to its argument and produces values of monadic type $M_\epsilon\mathbf{num}$; intuitively, this type describes computations that produce numeric results while performing rounding, and incurring at most ϵ in rounding error. Thus, our type for **pow2′** captures the desired error bound from Equation (9): for any input $v \in \mathbb{R}$, $\mathbf{pow2}'(v)$ approximates its ideal, infinitely precise counterpart $\mathbf{pow2}(v)$ within RP distance at most u, the unit roundoff.

To formalize this guarantee, programs of type $M_\epsilon\tau$ can be executed in two ways: under an ideal semantics where rounding operations act as the identity function, and under an approximate (floating-point) semantics where rounding operations round their arguments following some prescribed rounding strategy. We formalize these semantics in Section 4, and show our main soundness result: for programs with monadic type $M_\epsilon\mathbf{num}$, the result of the ideal computation differs from the result of the approximate computation by at most ϵ.

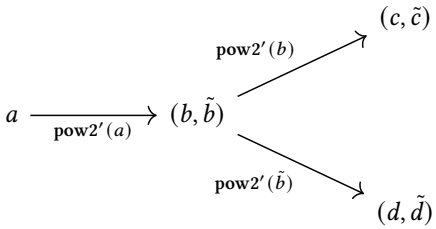
Composing Error Bounds. The type for $\mathbf{pow2}' : !_2\mathbf{num} \multimap M_u\mathbf{num}$ actually guarantees a bit more than just a bound on the roundoff: it also guarantees that the function is 2-sensitive under the *ideal* semantics, just like for **pow2**. (Under the approximate semantics, on the other hand, the function does *not* necessarily enjoy this guarantee.) It turns out that this added piece of information is crucial when analyzing how rounding errors propagate.

To see why, suppose we define a function that maps any number v to its fourth power: v^4. We can implement this function by using **pow2′** twice, like so:

$$\mathbf{pow4}\ x \triangleq \mathbf{let\text{-}bind}\ y = \mathbf{pow2}'\ x\ \mathbf{in}\ \mathbf{pow2}'\ y : M_{3u}\mathbf{num}.$$

The **let-bind**$-$**in** $-$ construct sequentially composes two monadic, effectful computations; to keep this example readable, we have elided some of the other syntax in $\Lambda_{\mathbf{num}}$. Thus, **pow4** first squares its argument, rounds the result, then squares again, rounding a second time.

The bound $3u$ on the total roundoff error deserves some explanation. In the typing rules for $\Lambda_{\mathbf{num}}$ we will see in Section 3, this index is computed as the sum $2u + u$, where the first term $2u$ is the error u from the first rounding operation *amplified by* 2 *since this error is fed into the second call of* **pow2′**, *a* 2*-sensitive function*, and the second term u is the roundoff error from the second rounding operation. If we think of **pow2′** as mapping a numeric value a to a pair of outputs $(b, \tilde{b})$, where b is the result under the exact semantics and $\tilde{b}$ is the result under the approximate semantics, we can visualize the computation **pow4**(a) as the following composition:



From left-to-right, the ideal and approximate results of **pow2′**(a) are b and $\tilde{b}$, respectively; the grade u on the monadic return type of **pow2′** ensures that these values are at distance at most u. The ideal result of **pow4**(a) is c, while the approximate result of **pow4**(a) is $\tilde{d}$. (The values $\tilde{c}$ and d arise from mixing ideal and approximate computations, and do not fully correspond to either the ideal or approximate semantics.) The 2-sensitivity guarantee of **pow2′** ensures that the distance between c and d is at most twice the distance between b and $\tilde{b}$—leading to the $2u$ term in the error—while the distance between d and $\tilde{d}$ is at most u. By applying the triangle inequality, the overall error bound is at most $2u + u = 3u$.

Error Propagation. So far, we have described how to bound the rounding error of a single computation applied to a single input. In practice, it is also often useful to analyze how errors *propagate* through a computation: given inputs with some roundoff error u, how does the output roundoff error depend on u? Our system can also be used for this kind of analysis; we give detailed examples in Section 5, but can use our running example of **pow4** to illustrate the idea here

If we denote by f the interpretation of an infinitely precise program, and by g an interpretation of a finite-precision program that approximates f, then we can use the triangle inequality to derive an upper bound on the relative precision of g with respect to f on distinct inputs:

$$RP(g(x), f(y)) \leq RP(g(x), f(x)) + RP(f(x), f(y)). \tag{10}$$

This upper bound is the sum of two terms: the first reflects the *local* rounding error—how much error is produced by the approximate function, and the second reflects by how much the function magnifies errors in the inputs—the sensitivity of the function.

Now, given that the full signature of **pow4** is $\mathbf{pow4} : !_4\mathbf{num} \multimap M_{3u}\mathbf{num}$, if we denote by g and f the approximate and ideal interpretations of **pow4**, from Equation (10) we expect our type system to produce the following bound on the propagation of error in **pow4**. For any exact number x and its approximation $\tilde{x}$ at distance at most u', we have:

$$RP(g(\tilde{x}), f(x)) \leq 3u + 4u'. \tag{11}$$

The term $4u'$ reflects that **pow4** is 4-sensitive in its argument, and that the (approximate) input value $\tilde{x}$ differs from its ideal value x by at most u'. In fact, we can use **pow4** to implement a function

$$
\begin{aligned}
\text{Types } \sigma, \tau &::= \mathbf{unit} \mid \mathbf{num} \mid \sigma \times \tau \mid \sigma \otimes \tau \mid \sigma + \tau \mid \sigma \multimap \tau \mid !_s\sigma \mid M_u\tau \\
\text{Values } v, w &::= x \mid \langle\rangle \mid k \in R \mid \langle v, w\rangle \mid (v, w) \mid \mathbf{inl}\ v \mid \mathbf{inr}\ v \\
&\quad \mid \lambda x.\, e \mid [v] \mid \mathbf{rnd}\ v \mid \mathbf{ret}\ v \mid \mathbf{let\text{-}bind}(\mathbf{rnd}\ v, x.f) \\
\text{Terms } e, f &::= v \mid v\ w \mid \pi_i\ v \mid \mathbf{let}\ (x, y) = v\ \mathbf{in}\ e \mid \mathbf{case}\ v\ \mathbf{of}\ (\mathbf{inl}\ x.e \mid \mathbf{inr}\ y.f) \\
&\quad \mid \mathbf{let}\ [x] = v\ \mathbf{in}\ e \mid \mathbf{let\text{-}bind}(v, x.f) \mid \mathbf{let}\ x = e\ \mathbf{in}\ f \mid \mathbf{op}(v) \quad \mathbf{op} \in O
\end{aligned}
$$

Fig. 1. Types, values, and terms.

pow4′ with the following type:

$$\mathbf{pow4'} : M_{u'}\mathbf{num} \multimap M_{3u+4u'}\mathbf{num}.$$

The type describes the error propagation: roundoff error at most u' in the input leads to roundoff error at most $3u + 4u'$ in the output.

3 THE LANGUAGE $\Lambda_{\mathbf{num}}$

3.1 Syntax

Figure 1 presents the syntax of types and terms. $\Lambda_{\mathbf{num}}$ is based on Fuzz [47], a linear call-by-value λ-calculus. For simplicity we do not treat recursive types, and $\Lambda_{\mathbf{num}}$ does not have general recursion.

Types. Some of the types in Figure 1 have already been mentioned in Section 2, including the linear function type $\tau \multimap \sigma$, the metric scaled $!_s\sigma$ type, and the monadic $M_\epsilon\mathbf{num}$ type. The base types are **unit** and numbers **num**. Following Fuzz, $\Lambda_{\mathbf{num}}$ has sum types $\sigma + \tau$ and two product types, $\tau \otimes \sigma$ and $\tau \times \sigma$, which are interpreted as pairs with different metrics.

Values and Terms. Our language requires that all computations are explicitly sequenced by let-bindings, $\mathbf{let}\ x = v\ \mathbf{in}\ e$, and term constructors and eliminators are restricted to values (including variables). This refinement of Fuzz better supports extensions to effectful languages [15]. In order to sequence monadic and metric scaled types, $\Lambda_{\mathbf{num}}$ provides the eliminators $\mathbf{let\text{-}bind}(v, x.e)$ and $\mathbf{let}\ [x] = v\ \mathbf{in}\ e$, respectively. The constructs $\mathbf{rnd}\ v$ and $\mathbf{ret}\ v$ lift values of plain type to monadic type; for metric types, the construct $[v]$ indicates scaling the metric of the type by a constant.

$\Lambda_{\mathbf{num}}$ is parameterized by a set R of numeric constants with type **num**. In Section 5, we will instantiate R and interpret **num** as a concrete set of numbers with a particular metric. $\Lambda_{\mathbf{num}}$ is also parameterized by a signature Σ: a set of operation symbols $\mathbf{op} \in O$, each with a type $\sigma \multimap \tau$, and a function $op : CV(\sigma) \to CV(\tau)$ mapping closed values of type σ to closed values of type τ. We write $\{\mathbf{op} : \sigma \multimap \tau\}$ in place of the tuple $(\sigma \multimap \tau, op : CV(\sigma) \to CV(\tau), \mathbf{op})$. For now, we make no assumptions on the functions op; in Section 4 we will need further assumptions.

3.2 Static Semantics

The static semantics of $\Lambda_{\mathbf{num}}$ is given in Figure 2. Before stepping through the details of each rule, we require some definitions regarding typing judgments and typing environments.

Terms in $\Lambda_{\mathbf{num}}$ are typed with judgments of the form $\Gamma \vdash e : \sigma$ where Γ is a typing environment and σ is a type. Environments are defined by the syntax $\Gamma, \Delta ::= \cdot \mid \Gamma, x :_s \sigma$. We can also view a typing environment Γ as a partial map from variables to types and sensitivities, where $(\sigma, s) = \Gamma(x)$ when $x :_s \sigma \in \Gamma$. Intuitively, if the environment Γ has a binding $x :_s \sigma \in \Gamma$, then a term e typed under Γ has sensitivity s to perturbations in the variable x; 0-sensitivity means that the term does

$$\frac{s \geq 1}{\Gamma, x :_s \sigma, \Delta \vdash x : \sigma}\ (\text{Var}) \quad \frac{\Gamma, x :_1 \sigma \vdash e : \tau}{\Gamma \vdash \lambda x.e : \sigma \multimap \tau}\ (\multimap \text{I}) \quad \frac{\Gamma \vdash v : \sigma \multimap \tau \qquad \Theta \vdash w : \sigma}{\Gamma + \Theta \vdash vw : \tau}\ (\multimap \text{E})$$

$$\frac{}{\Gamma \vdash \langle\rangle : \mathbf{unit}}\ (\text{Unit}) \quad \frac{\Gamma \vdash v : \sigma \qquad \Gamma \vdash w : \tau}{\Gamma \vdash \langle v, w\rangle : \sigma \times \tau}\ (\times \text{I}) \quad \frac{\Gamma \vdash v : \tau_1 \times \tau_2}{\Gamma \vdash \pi_i\, v : \tau_i}\ (\times \text{E})$$

$$\frac{\Gamma \vdash v : \sigma \qquad \Theta \vdash w : \tau}{\Gamma + \Theta \vdash (v, w) : \sigma \otimes \tau}\ (\otimes \text{I}) \quad \frac{\Gamma \vdash v : \sigma \otimes \tau \qquad \Theta, x :_s \sigma, y :_s \tau \vdash e : \rho}{s * \Gamma + \Theta \vdash \mathbf{let}\ (x, y)\ =\ v\ \mathbf{in}\ e : \rho}\ (\otimes \text{E})$$

$$\frac{\Gamma \vdash v : \sigma}{\Gamma \vdash \mathbf{inl}\ v : \sigma + \tau}\ (+\text{I}_L) \quad \frac{\Gamma \vdash v : \tau}{\Gamma \vdash \mathbf{inr}\ v : \sigma + \tau}\ (+\text{I}_R) \quad \frac{\Gamma \vdash v : !_s\sigma \qquad \Theta, x :_{t*s} \sigma \vdash e : \tau}{t * \Gamma + \Theta \vdash \mathbf{let}\ [x] = v\ \mathbf{in}\ e : \tau}\ (!\ \text{E})$$

$$\frac{\Gamma \vdash v : \sigma + \tau \qquad \Theta, x :_s \sigma \vdash e : \rho \qquad \Theta, y :_s \tau \vdash f : \rho \qquad s > 0}{s * \Gamma + \Theta \vdash \mathbf{case}\ v\ \mathbf{of}\ (\mathbf{inl}\ x.e \mid \mathbf{inr}\ y.f) : \rho}\ (+\text{E}) \quad \frac{\Gamma \vdash v : \sigma}{s * \Gamma \vdash [v] : !_s\sigma}\ (!\ \text{I})$$

$$\frac{\Gamma \vdash e : \tau \qquad \Theta, x :_s \tau \vdash f : \sigma \qquad s > 0}{s * \Gamma + \Theta \vdash \mathbf{let}\ x = e\ \mathbf{in}\ f : \sigma}\ (\text{Let}) \quad \frac{k \in R}{\Gamma \vdash k : \mathbf{num}}\ (\text{Const})$$

$$\frac{\Gamma \vdash e : M_q\tau \qquad r \geq q}{\Gamma \vdash e : M_r\tau}\ (\text{Subsumption}) \quad \frac{\Gamma \vdash v : \tau}{\Gamma \vdash \mathbf{ret}\ v : M_0\tau}\ (\text{Ret}) \quad \frac{\Gamma \vdash v : \mathbf{num}}{\Gamma \vdash \mathbf{rnd}\ v : M_q\mathbf{num}}\ (\text{Rnd})$$

$$\frac{\Gamma \vdash v : M_r\sigma \qquad \Theta, x :_s \sigma \vdash f : M_q\tau}{s * \Gamma + \Theta \vdash \mathbf{let\text{-}bind}(v, x.f) : M_{s*r+q}\tau}\ (M_u\ \text{E}) \quad \frac{\Gamma \vdash v : \sigma \qquad \{\mathbf{op} : \sigma \multimap \mathbf{num}\} \in \Sigma}{\Gamma \vdash \mathbf{op}(v) : \mathbf{num}}\ (\text{Op})$$

Fig. 2. Typing rules for $\Lambda_{\mathbf{num}}$, with $s, t, q, r \in \mathbb{R}^{\geq 0} \cup \{\infty\}$.

not depend on x, while infinite sensitivity means that any perturbation in x can lead to arbitrarily large changes in e. Well-typed expressions of the form $x :_s \sigma \vdash e : \tau$ represent computations that have permission to be s-sensitive in the variable x.

Many of the typing rules for $\Lambda_{\mathbf{num}}$ involve summing and scaling typing environments. The notation $s * \Gamma$ denotes scalar multiplication of the variable sensitivities in Γ by s, and is defined as

$$s * \cdot = \cdot \qquad\qquad s * (\Gamma, x :_t \sigma) = s * \Gamma, x :_{s*t},$$

where we require that $0 \cdot \infty = \infty \cdot 0 = 0$. The sum $\Gamma + \Delta$ of two typing environments is defined if they assign the same types to variables that appear in both environments. All typing rules that involve summing environments ($\Gamma + \Delta$) implicitly require that Γ and Δ are *summable*.

Definition 3.1. The environments Γ and Δ are *summable iff* for any $x \in dom(\Gamma) \cap dom(\Delta)$, if $(\sigma, s) = \Gamma(x)$, then there exists an element $t \in \mathbb{R}^{\geq 0} \cup \{\infty\}$ such that $(\sigma, t) = \Delta(x)$.

Under this condition, we can define the sum $\Gamma + \Delta$ as follows.

$$\cdot + \cdot = \cdot \qquad\qquad (\Gamma, x :_s \sigma) + \Delta = (\Gamma + \Delta), x :_s \sigma \text{ if } x \notin \Delta$$
$$\Gamma + (\Delta, x :_s \sigma) = (\Gamma + \Delta), x :_s \sigma \text{ if } x \notin \Gamma \qquad (\Gamma + \Delta), x :_{s+t} \sigma = (\Gamma, x :_s \sigma) + (\Delta, x :_t \sigma)$$

We now consider the rules in Figure 2. The simplest are (Const) and (Var), which allow any constant to be used under any environment, and allow a variable from the environment to be used so long as its sensitivity is at least 1.

The introduction and elimination rules for the products $\otimes$ and $\times$ are similar to those given in Fuzz. In ($\otimes$ I), introducing the pair requires summing the environments in which the individual elements were defined, while in ($\times$ I), the elements of the pair share the same environment.

The typing rules for sequencing (Let) and case analysis (+ E) both require that the sensitivity s is strictly positive. While the restriction in (Let) is not needed for a terminating calculus, like ours, it is required for soundness in the presence of non-termination [15]. The restriction in (+ E) is needed for soundness (we discuss this detail in Section 8).

The remaining interesting rules are those for metric scaling and monadic types. In the (! I) rule, the box constructor $[-]$ indicates scalar multiplication of an environment. The (! E) rule is similar to ($\otimes$ E), but includes the scaling on the let-bound variable.

The rules (Subsumption), (Ret), (Rnd), and (M_u E) are the core rules for performing rounding error analysis in $\Lambda_{\mathbf{num}}$. Intuitively, the monadic type $M_\epsilon \mathbf{num}$ describes computations that produce numeric results while performing rounding, and incur at most ϵ in rounding error. The subsumption rule states that rounding error bounds can be loosened. The (Ret) rule states that we can lift terms of plain type to monadic type without introducing rounding error. The (Rnd) rule types the primitive rounding operation, which introduces roundoff errors. Here, q is a fixed numeric constant describing the roundoff error incurred by a rounding operation. The precise value of this constant depends on the precision of the format and the specified rounding mode; we leave q unspecified for now. In Section 5, we will illustrate how to instantiate our language to different settings.

The monadic elimination rule (M_u E) allows sequencing two rounded computations together. This rule formalizes the interaction between sensitivities and rounding, as we illustrated in Section 2: the rounding error of the body of the let-binding $\mathbf{let\text{-}bind}(v, x.f)$ is upper bounded by the sum of the roundoff error of the value v scaled by the sensitivity of f to x, and the roundoff error of f.

Our type system satisfies the usual properties of weakening and substitution.

3.3 Dynamic Semantics

We use a small-step operational semantics adapted from Fuzz [47], extended with rules for the monadic let-binding. We show here the evaluation rules that are unique to $\Lambda_{\mathbf{num}}$.

If the judgment $e \mapsto e'$ indicates that the expression e takes a single step, resulting in the expression e', then for the **let-bind** construct we have the following evaluation rules.

$$\mathbf{let\text{-}bind}(\mathbf{ret}\ v, x.e) \mapsto e[v/x]$$

$$\mathbf{let\text{-}bind}(\mathbf{let\text{-}bind}(v, x.f), y.g) \mapsto \mathbf{let\text{-}bind}(v, x.\mathbf{let\text{-}bind}(f, y.g)) \quad x \notin FV(g)$$

Although our language does not have recursive types, the **let-bind** construct makes it somewhat less obvious that the calculus is terminating: the evaluation rules for **let-bind** rearrange the term but do not reduce its size. Even so, a standard logical relations argument can be used to show that well-typed programs are terminating.

4 DENOTATIONAL SEMANTICS AND ERROR SOUNDNESS

In this section, we show two central guarantees of $\Lambda_{\mathbf{num}}$: bounded sensitivity and bounded error.

4.1 Categorical Preliminaries

We provide a denotational semantics for our language based on the categorical semantics of Fuzz, due to Azevedo de Amorim et al. [5]. Our language has many similarities to Fuzz, with some key differences needed for our application—most notably, our language does not have recursive types and non-termination, but it does have a novel graded monad which we will soon discuss. We emphasize that we use category theory as a concise language for defining our semantics—we are ultimately interested in a specific, concrete interpretation of our language. The general categorical semantics of Fuzz-like languages has been studied in prior work [24].

Basic concepts. To begin, we quickly review some basic concepts from category theory; the interested reader should consult a textbook for a more gentle introduction [3, 37]. We will introduce more specialized concepts as we go along. A *category* $\mathbf{C}$ consists of a collection Ob of objects, and a collection of morphisms $Hom_{\mathbf{C}}(A, B)$ for every pair of objects $A, B \in Ob_{\mathbf{C}}$. For every pair of morphisms $f \in Hom_{\mathbf{C}}(A, B)$ and $g \in Hom_{\mathbf{C}}(B, C)$, the *composition* $g \circ f$ is defined to be a morphism in $Hom_{\mathbf{C}}(A, C)$. There is an *identity* morphism $id_A \in Hom_{\mathbf{C}}(A, A)$ corresponding to object A; this morphism acts as the identity under composition: $f \circ id = id \circ f = f$.

A *functor* F from category $\mathbf{C}$ to category $\mathbf{D}$ consists of a function on objects $F : Ob_{\mathbf{C}} \to Ob_{\mathbf{D}}$, and a function on morphisms $F : Hom_{\mathbf{C}}(A, B) \to Hom_{\mathbf{D}}(A, B)$. The mapping on morphisms should preserve identities and composition: $F(id_A) = id_A$, and $F(g \circ f) = F(g) \circ F(f)$. Finally, a *natural transformation* α from a functor $F : \mathbf{C} \to \mathbf{D}$ to a functor $G : \mathbf{C} \to \mathbf{D}$ consists of a family of morphisms $\alpha_A \in Hom_{\mathbf{D}}(F(A), G(A))$, one per object $A \in Ob_{\mathbf{C}}$, that commutes with the functor F and G applied to any morphism: for every $f \in Hom_{\mathbf{C}}(A, B)$, we have $\alpha_B \circ F(f) = G(f) \circ \alpha_A$.

The category **Met**. Our type system is designed to bound the distance between various kinds of program outputs. Intuitively, types should be interpreted as *metric spaces*, which are sets equipped with a distance function satisfying several standard axioms. Azevedo de Amorim et al. [5] identified the following slight generalization of metric spaces as a suitable category to interpret Fuzz.

Definition 4.1. An *extended pseudo-metric space* (A, d_A) consists of a *carrier* set A and a *distance* $d_A : A \times A \to \mathbb{R}^{\geq 0} \cup \{\infty\}$ satisfying (i) reflexivity: $d(a, a) = 0$; (ii) symmetry: $d(a, b) = d(b, a)$; and (iii) triangle inequality: $d(a, c) \leq d(a, b) + d(b, c)$ for all $a, b, c, \in A$. We write $|A|$ for the carrier set.

A *non-expansive map* $f : (A, d_a) \to (B, d_B)$ between extended pseudo-metric spaces consists of a set-map $f : A \to B$ such that $d_B(f(a), f(a')) \leq d_A(a, a')$. The identity function is a non-expansive map, and non-expansive maps are closed under composition. Therefore, extended pseudo-metric spaces and non-expansive maps form a category **Met**.

Extended pseudo-metric spaces differ from standard metric spaces in two respects. First, their distance functions can assign infinite distances (*extended* real numbers). Second, their distance functions are only *pseudo*-metrics because they can assign distance zero to pairs of distinct points. Since we will only be concerned with extended pseudo-metric spaces, we will refer to them as metric spaces for short.

The category **Met** supports several constructions that are useful for interpreting linear type systems. First, there are products and coproducts on **Met**. The Cartesian product $(A, d_A) \times (B, d_B)$ has carrier $A \times B$ and distance given by the max: $d_{A \times B}((a, b), (a', b')) = \max(d_A(a, a'), d_B(b, b'))$. The tensor product $(A, d_A) \otimes (B, d_B)$ also has carrier $A \times B$, but with distance given by the sum: $d_{A \otimes B}((a, b), (a', b')) = d_A(a, a') + d_B(b, b')$. Both products are useful for modeling natural metrics on pairs and tuples. The category **Met** also has coproducts $(A, d_A) + (B, d_B)$, where the carrier is disjoint union $A \uplus B$ and the metric d_{A+B} assigns distance ∞ to pairs of elements in different injections, and distance d_A or d_B to pairs of elements in A or B, respectively.

Second, non-expansive functions can be modeled in **Met**. The function space $(A, d_A) \multimap (B, d_B)$ has carrier set $\{f : A \to B \mid f \text{ non-expansive}\}$ and distance given by the supremum norm: $d_{A \multimap B}(f, g) = \sup_{a \in A} d_B(f(a), g(a))$. Moreover, the functor $(- \otimes B)$ is left-adjoint to the functor $(B \multimap -)$, so maps $f : A \otimes B \to C$ can be curried to $\lambda(f) : A \to (B \multimap C)$, and uncurried. These constructions, plus a few additional pieces of data, make $(\mathbf{Met}, I, \otimes, \multimap)$ a *symmetric monoidal closed category* (SMCC), where the unit object I is the metric space with a single element.

A graded comonad on **Met**. Languages like Fuzz are based on *bounded linear logic* [25], where the exponential type $!A$ is refined into a family of bounded exponential types $!_s A$ where s is drawn from a pre-ordered semiring $\mathcal{S}$. The grade s can be used to track more fine-grained, possibly quantitative

aspects of well-typed terms, such as function sensitivities. These bounded exponential types can be modeled by a categorical structure called a $\mathcal{S}$-*graded exponential comonad* [10, 24]. Given any metric space (A, d_A) and non-negative number r, there is an evident operation that scales the metric by r: $(A, r \cdot d_A)$. This operation can be extended to a graded comonad.

Definition 4.2. Let the pre-ordered semiring $\mathcal{S}$ be the extended non-negative real numbers $\mathbb{R}^{\geq 0} \cup \{\infty\}$ with the usual order, addition, and multiplication; $0 \cdot \infty$ and $\infty \cdot 0$ are defined to be 0. We define functors $\{D_s : \mathbf{Met} \to \mathbf{Met} \mid s \in \mathcal{S}\}$ such that $D_s : \mathbf{Met} \to \mathbf{Met}$ takes metric spaces (A, d_A) to metric spaces $(A, s \cdot d_A)$, and non-expansive maps $f : A \to B$ to $D_s f : D_s A \to D_s B$, with the same underlying map.

We get a graded comonad by defining associated natural transformations.

4.2 A Graded Monad on Met

The categorical structures we have seen so far are enough to interpret the non-monadic fragment of our language, which is essentially the core of the Fuzz language [5]. As proposed by Gaboardi et al. [24], this core language can model effectful computations using a graded monadic type, which can be modeled categorically by (i) a *graded strong monad*, and (ii) a *distributive law* modeling the interaction of the graded comonad and the graded monad.

The neighborhood monad. Recall the intuition behind our system: closed programs e of type $M_\epsilon \mathbf{num}$ are computations producing outputs in **num** that may perform rounding operations. The index ϵ should bound the distance between the output under the *ideal* semantics, where rounding is the identity, and the *floating-point (FP)* semantics, where rounding maps a real number to a representable floating-point number following a prescribed rounding procedure. Accordingly, the interpretation of the graded monad should track *pairs* of values—the ideal value, and the FP value.

This perspective points towards the following graded monad on **Met**, which we call the *neighborhood monad*. While the definition appears quite natural mathematically, we are not aware of this graded monad appearing in prior work.

Definition 4.3. Let the pre-ordered monoid $\mathcal{R}$ be the extended non-negative real numbers $\mathbb{R}^{\geq 0} \cup \{\infty\}$ with the usual order and addition. The *neighborhood monad* is defined by the functors $\{T_r : \mathbf{Met} \to \mathbf{Met} \mid r \in \mathcal{R}\}$ and associated natural transformations as follows:

- The functor $T_r : \mathbf{Met} \to \mathbf{Met}$ takes a metric space M to a metric space with underlying set:
$$|T_r M| \triangleq \{(x, y) \in M \mid d_M(x, y) \leq r\}$$
and the metric is: $d_{T_r M}((x, y), (x', y')) \triangleq d_M(x, x')$.
- The functor T_r takes a non-expansive function $f : A \to B$ to $T_r f : T_r A \to T_r B$ with
$$(T_r f)((x, y)) \triangleq (f(x), f(y))$$
- For $r, q \in \mathcal{R}$ and $q \leq r$, the map $(q \leq r)_A : T_q A \to T_r A$ is the identity.
- The unit map $\eta_A : A \to T_0 A$ is defined via: $\eta_A(x) \triangleq (x, x)$.
- The graded multiplication map $\mu_{q,r,A} : T_q(T_r A) \to T_{r+q} A$ is defined via:
$$\mu_{q,r,A}((x, y), (x', y')) \triangleq (x, y').$$

The definitions of T_r are evidently functors. The associated maps are natural transformations, and define a graded monad [22, 33]. The neighborhood monad is a graded *strong* monad [42], and the scaling comonad distributes over the neighborhood monad.

4.3 Interpreting the Language

We are now ready to interpret our language in **Met**.

Interpreting types. We interpret each type τ as a metric space $[\![\tau]\!]$, using constructions in **Met**.

Definition 4.4. Define the type interpretation by induction on the type syntax:

$$[\![\mathbf{unit}]\!] \triangleq I = (\{\star\}, 0) \qquad [\![\mathbf{num}]\!] \triangleq (R, d_R) \qquad [\![A \otimes B]\!] \triangleq [\![A]\!] \otimes [\![B]\!] \qquad [\![A \times B]\!] \triangleq [\![A]\!] \times [\![B]\!]$$

$$[\![A + B]\!] \triangleq [\![A]\!] + [\![B]\!] \qquad [\![A \multimap B]\!] \triangleq [\![A]\!] \multimap [\![B]\!] \qquad [\![!_s A]\!] \triangleq D_s[\![A]\!] \qquad [\![M_r A]\!] \triangleq T_r[\![A]\!]$$

We do not fix the interpretation of the base type **num**: (R, d_R) can be any metric space.

Interpreting judgments. We will interpret each typing derivation showing a typing judgment $\Gamma \vdash e : \tau$ as a morphism in **Met** from the metric space $[\![\Gamma]\!]$ to the metric space $[\![\tau]\!]$. Since all morphisms in this category are non-expansive, this will show (a version of) metric preservation. We first define the metric space $[\![\Gamma]\!]$:

$$[\![\cdot]\!] \triangleq I = (\{\star\}, 0) \qquad [\![\Gamma, x :_s \tau]\!] \triangleq [\![\Gamma]\!] \otimes D_s[\![\tau]\!]$$

Given any binding $x :_r \tau \in \Gamma$, there is a non-expansive map from $[\![\Gamma]\!]$ to $[\![\tau]\!]$ projecting out the x-th position; we sometimes use notation that treats an element $\gamma \in [\![\Gamma]\!]$ as a function, so that $\gamma(x) \in [\![\tau]\!]$.

We are now ready to define our interpretation of typing judgments. Our definition is parametric in the interpretation of three things: the numeric type $[\![\mathbf{num}]\!] = (R, d_R)$, the rounding operation ρ, and the operations in the signature Σ.

Definition 4.5. Fix $\rho : R \to R$ to be a (set) function such that for every $r \in R$ we have $d_R(r, \rho(r)) \leq \epsilon$, and for every operation $\{\mathbf{op} : \sigma \multimap \tau\} \in \Sigma$ in the signature fix an interpretation $[\![\mathbf{op}]\!] : [\![\sigma]\!] \to [\![\tau]\!]$ such that for every closed value $\cdot \vdash v : \sigma$, we have $[\![\mathbf{op}]\!]([\![v]\!]) = [\![op(v)]\!]$.

Then we can interpret each well-typed program $\Gamma \vdash e : \tau$ as a non-expansive map $[\![\Gamma \vdash e : \tau]\!] : [\![\Gamma]\!] \to [\![\tau]\!]$, by induction on the typing derivation, via case analysis on the last rule.

Soundness of operational semantics. Now, we can show that the operational semantics from Section 3 is sound with respect to the metric space semantics: stepping a well-typed term does not change its denotational semantics.

Lemma 4.6 (Substitution). *Let $\Gamma, \Delta, \Gamma' \vdash e : \tau$ be a well-typed term, and let $\vec{v} : \Delta$ be a well-typed substitution of closed values, i.e., we have derivations $\cdot \vdash v_x : \Delta(x)$. Then there is a derivation of*

$$\Gamma, \Gamma' \vdash e[\vec{v}/dom(\Delta)] : \tau$$

with semantics $[\![\Gamma, \Gamma' \vdash e[\vec{v}/dom(\Delta)] : \tau]\!] = (id_{[\![\Gamma]\!]} \otimes [\![\cdot \vdash \vec{v} : \Delta]\!] \otimes id_{[\![\Gamma']\!]}); [\![\Gamma, \Delta, \Gamma' \vdash e : \tau]\!]$.

Lemma 4.7 (Preservation). *Let $\cdot \vdash e : \tau$ be a well-typed closed term, and suppose $e \mapsto e'$. Then there is a derivation of $\cdot \vdash e' : \tau$, and the semantics of both derivations are equal: $[\![\vdash e : \tau]\!] = [\![\vdash e' : \tau]\!]$.*

4.4 Error Soundness

The metric semantics interprets each program as a non-expansive map. We aim to show that values of monadic type $M_r\sigma$ are interpreted as pairs of values, where the first value is the result under an ideal operational semantics and the second value is the result under an approximate, or finite-precision (FP) operational semantics.

To make this connection precise, we first define the ideal and FP operational semantics of our programs, refining our existing operational semantics so that the rounding operation steps to a number. Then, we define two denotational semantics of our programs capturing the ideal and FP behaviors of programs, and show that the ideal and FP operational semantics are sound with respect to this denotation. Finally, we relate our metric semantics with our ideal and FP semantics, showing how well-typed programs of monadic type satisfy the error bound indicated by their type.

Ideal and FP operational semantics. We first refine our operational semantics to capture ideal and FP behaviors.

Definition 4.8. We define two step relations $e \mapsto_{id} e'$ and $e \mapsto_{fp} e'$ by augmenting the operational semantics with the following rules:

$$\mathbf{rnd}\ k \mapsto_{id} \mathbf{ret}\ k \qquad \text{and} \qquad \mathbf{rnd}\ k \mapsto_{fp} \mathbf{ret}\ \rho(k)$$

Note that $\mathbf{let\text{-}bind}(\mathbf{rnd}\ k, x.f)$ is no longer a value under these semantics, since $\mathbf{rnd}\ k$ can step. Also note that these semantics are deterministic, and by a standard logical relations argument, all well-typed terms normalize.

Ideal and FP denotational semantics. Much like our approach in **Met**, we next define a denotational semantics of our programs so that we can abstract away from the step relation. We develop both the ideal and approximate semantics in **Set**, where maps are not required to be non-expansive.

Definition 4.9. Let $\Gamma \vdash e : \tau$ be a well-typed program. We can define two semantics in **Set**:

$$(\!|\Gamma \vdash e : \tau|\!)_{id} : (\!|\Gamma|\!)_{id} \to (\!|\tau|\!)_{id} \qquad\qquad (\!|\Gamma \vdash e : \tau|\!)_{fp} : (\!|\Gamma|\!)_{fp} \to (\!|\tau|\!)_{fp}$$

We take the graded comonad D_s and the graded monad T_r to both be the identity functor on **Set**:

$$(\!|M_u\ \tau|\!)_{id} = (\!|!_s\ \tau|\!)_{id} \triangleq (\!|\tau|\!)_{id} \qquad\qquad (\!|M_u\ \tau|\!)_{fp} = (\!|!_s\ \tau|\!)_{fp} \triangleq (\!|\tau|\!)_{fp}$$

The ideal and floating point interpretations of well-typed programs are straightforward, by induction on the derivation of the typing judgment. The only interesting case is for Round:

$$(\!|\Gamma \vdash \mathbf{rnd}\ k : M_\epsilon \mathbf{num}|\!)_{id} \triangleq (\!|\Gamma \vdash k : \mathbf{num}|\!)_{id} \qquad (\!|\Gamma \vdash \mathbf{rnd}\ k : M_\epsilon \mathbf{num}|\!)_{fp} \triangleq (\!|\Gamma \vdash k : \mathbf{num}|\!)_{fp}; \rho$$

where $\rho : R \to R$ is the rounding function.

Following the same approach as in Lemma 4.7, it is straightforward to prove that these denotational semantics are sound for their respective operational semantics.

Lemma 4.10 (Preservation). *Let $\cdot \vdash e : \tau$ be a well-typed closed term, and suppose $e \mapsto_{id} e'$. Then there is a derivation of $\cdot \vdash e' : \tau$ and the semantics of both derivations are equal: $(\!|\vdash e : \tau|\!)_{id} = (\!|\vdash e' : \tau|\!)_{id}$. The same holds for the FP denotational and operational semantics.*

Establishing error soundness. Finally, we connect the metric semantics with the ideal and FP semantics. Let $U : \mathbf{Met} \to \mathbf{Set}$ be the forgetful functor mapping each metric space to its underlying set, and each morphism of metric spaces to its underlying function on sets. We have:

Lemma 4.11 (Pairing). *Let $\cdot \vdash e : M_r \mathbf{num}$. Then we have: $U[\![e]\!] = \langle (\!|e|\!)_{id}, (\!|e|\!)_{fp} \rangle$ in* **Set***: the first projection of $U[\![e]\!]$ is $(\!|e|\!)_{id}$, and the second projection is $(\!|e|\!)_{fp}$.*

As a corollary, we have soundness of the error bound for programs with monadic type.

Corollary 4.12 (Error soundness). *Let $\cdot \vdash e : M_r \mathbf{num}$ be a well-typed program. Then $e \mapsto^*_{id} \mathbf{ret}\ v_{id}$ and $e \mapsto^*_{fp} \mathbf{ret}\ v_{fp}$ such that $d_{[\![\mathbf{num}]\!]}([\![v_{id}]\!], [\![v_{fp}]\!]) \leq r$.*

5 CASE STUDIES

To illustrate how $\Lambda_{\mathbf{num}}$ can be used to bound the sensitivity and roundoff error of numerical programs, we must fix our interpretation of the numeric type $[\![\mathbf{num}]\!]$ using an appropriate metric space (R, d_R) and augment our language to include primitive arithmetic operations over the set R.

If we interpret our numeric type **num** as the set of strictly positive real numbers $\mathbb{R}^{>0}$ with the relative precision (RP) metric (Definition 2.2), then we can use $\Lambda_{\mathbf{num}}$ to perform a relative

```
add : (num × num)—∘ num
mul : (num ⊗ num)—∘ num
div : (num ⊗ num)—∘ num
sqrt : ![0.5]num—∘ num
```

Fig. 3. Primitive operations in $\Lambda_{\mathbf{num}}$, typed using the relative precision (RP) metric.

```
addfp : (num × num)—∘ M[eps]num
mulfp : (num ⊗ num)—∘ M[eps]num
divfp : (num ⊗ num)—∘ M[eps]num
sqrtfp : ![0.5]num—∘ M[eps]num
```

Fig. 4. Type signatures of defined operations that perform rounding in $\Lambda_{\mathbf{num}}$; eps denotes the unit roundoff.

```
function mulfp (xy: (num, num))
  : M[eps]num {
  s = mul xy;
  rnd s
}
```

```
function addfp (xy: <num, num>)
  : M[eps]num {
  s = add xy;
  rnd s
}
```

Fig. 5. Example defined operations that perform rounding in $\Lambda_{\mathbf{num}}$. We denote the unit roundoff by eps.

error analysis as described by Olver [44]. Using this metric, we extend the language with the four primitive arithmetic operations shown in Figure 3.

Recall that our metric semantics interprets each program as a non-expansive map. If we take the semantics of the arithmetic operations as being the standard addition and multiplication of positive real numbers, then `add` and `mul` as defined in Figure 3 are non-expansive functions [44, Corollary 1 & Property V]; recall that the two product types have different metrics (Section 4.3).

Using the primitive operations in Figure 3 and the **rnd** construct, we can write functions for the basic arithmetic operations in $\Lambda_{\mathbf{num}}$ that perform concrete rounding when interpreted according to the FP semantics. The type signature of these functions is shown in Figure 4, and implementations of a multiplication and addition that perform rounding are shown in Figure 5.

The examples presented in this section use the actual syntax of an implementation of $\Lambda_{\mathbf{num}}$, which is introduced in Section 6. The implementation closely follows the syntax of the language as presented in Figure 2, with some additional syntactic sugar. For instance, we write `(x = v; e)` to denote **let** $x = v$ **in** e, and `(let x = v; f)` to denote **let-bind**$(v, x.f)$. For top level programs, we write `(function ID args {`v`} e)` to denote **let** ID $= v$ **in** e, where v is a lambda term with arguments `args`. We write pairs of type $-\times-$ and $-\otimes-$ as `(|-,-|)` and `(-,-)`, respectively. Finally, for types, we write `M[u]num` to represent monadic types with a numeric grade `u` and we write `![s]` to represent exponential types with a numeric grade `s`.

Choosing the RP Rounding Function. Recall that we require the rounding function ρ to be a function such that for every $x \in R^{>0}$, we have $RP(x, \rho(x)) \leq \epsilon$; that is, the rounding function must satisfy an accuracy guarantee with respect to the metric RP on the set $R^{>0}$. If we choose $\rho_{RU} : R^{>0} \to R^{>0}$ to be rounding towards $+\infty$, then by eq. (7) we have that $RP(x, \rho(x)) \leq$ `eps` where `eps` is the unit roundoff. Error soundness (Corollary 4.12) implies that for the functions `mulfp` and `addfp`, the results of the ideal and approximate computations differ by at most `eps`.

Underflow and overflow. In the following examples, *we assume that the results of computations do not overflow or underflow.* Recall from Section 2 that the standard model for floating-point arithmetic given in eq. (2) is only valid under this assumption. In Section 7, we discuss how $\Lambda_{\mathbf{num}}$ can be extended to handle overflow, underflow, and exceptional values.

Example: The Fused Multiply-Add Operation. We warm up with a simple example of a *multiply-add* (MA) operation: given x, y, z, we want to compute $x * y + z$. The $\Lambda_{\mathbf{num}}$ implementation of `MA` is

```
function MA (x: num) (y: num) (z: num)
  : M[2*eps]num {
  s = mulfp (x,y);
  let a = s;
  addfp (|a,z|)
}
```

```
function FMA (x: num) (y: num) (z: num)
  : M[eps]num {
  a = mul (x,y);
  b = add (|a,z|);
  rnd b
}
```

Fig. 6. Multiply-add and fused multiply-add in $\Lambda_{\mathbf{num}}$.

```
function Horner2
  (a0: num) (a1: num)
  (a2: num) (x: ![2.0]num)
  : M[2*eps]num {
  let [x1] = x ;
  s1 = FMA a2 x' a1;
  let z = s1;
  FMA z x1 a0
}
```

```
function Horner2_with_error
  (a0: M[eps]num) (a1: M[eps]num)
  (a2: M[eps]num) (x: ![2.0]M[eps]num)
  : (M[7*eps]num) {
  let [x1] = x ;
  let a0' = a0; let a1' = a1;
  let a2' = a2; let x' = x1;
  s1 = FMA a2' x' a1';
  let z = s1;
  FMA z x' a0'
}
```

Fig. 7. Horner's scheme for evaluating a second order polynomial in $\Lambda_{\mathbf{num}}$ with (`Horner2_with_error`) and without (`Horner2`) input error.

given in Figure 6. The index `2*eps` on the return type indicates that the roundoff error is at most twice the unit roundoff, due to the two separate rounding operations in `mulfp` and `addfp`.

Multiply-add is extremely common in numerical code, and modern architectures typically support a *fused* multiply-add (FMA) operation. This operation performs a multiplication followed by an addition, $x * y + z$, as though it were a single floating-point operation. The FMA operation therefore incurs a single rounding error, rather than two. The $\Lambda_{\mathbf{num}}$ implementation of a FMA operation is given in Figure 6. The index on the return type of the function is `eps`, reflecting a reduction in the roundoff error when compared to the function `MA`.

Example: Evaluating Polynomials. A standard method for evaluating a polynomial is Horner's scheme, which rewrites an nth-degree polynomial $p(x) = a_0 + a_1 x + \cdots a_n x^n$ as

$$p(x) = a_0 + x(a_1 + x(a_2 + \cdots x(a_{n-1} + a x_n) \cdots)),$$

and computes the result using only n multiplications and n additions. Using $\Lambda_{\mathbf{num}}$, we can perform an error analysis on a version of Horner's scheme that uses a FMA operation to evaluate second-order polynomials of the form $p(\vec{a}, x) = a_2 x^2 + a_1 x + a_0$ where x and all a_is are non-zero positive constants. The implementation `Horner2` in $\Lambda_{\mathbf{num}}$ is given in Figure 7 and shows that the rounding error on exact inputs is guaranteed to be bounded by `2*eps`:

$$RP((\!|\mathbf{Horner2}\ as\ x|\!)_{id}, (\!|\mathbf{Horner2}\ as\ x|\!)_{fp}) \leq 2 * \texttt{eps}. \tag{12}$$

Example: Error Propagation and Horner's Scheme. As a consequence of the metric interpretation of programs (Section 4.3), the type of `Horner2` also guarantees bounded sensitivity of the ideal semantics, which corresponds to $p(\vec{a}, x) = a_2 x^2 + a_1 x + a_0$. Thus for any $a_i, a_i', x, x' \in R^{>0}$, we can measure the sensitivity of `Horner2` to rounding errors introduced by the inputs: if x' is an

approximation to x of RP q, and each a_i is an approximation to its corresponding a_i of RP r, then

$$RP(p(\vec{a}, x), p(\vec{a}', x')) \leq \sum_{i=0}^{2} RP(a_i, a_i') + 2 \cdot RP(x, x') \leq 3r + 2q. \tag{13}$$

The term $2q$ reflects that `Horner2` is 2-sensitive in the variable x. The fact that we take the sum of the approximation distances over the a_i's follows from the metric on the function type (Section 4.3).

The interaction between the sensitivity of the function under its ideal semantics and the rounding error incurred by `Horner2` over exact inputs is made clear by the function `Horner2_with_error`, shown in Figure 7. From the type, we see that the total roundoff error of `Horner2_with_error` is `7*eps`: from eq. (13) it follows that the sensitivity of the function contributes `5*eps`, and rounding error incurred by evaluating `Horner2` over exact inputs contributes the remaining `2*eps`.

6 IMPLEMENTATION AND EVALUATION

6.1 Prototype Implementation

We have developed a prototype type-checker for $\Lambda_{\mathbf{num}}$ in OCaml, based on the sensitivity-inference algorithm due to Azevedo de Amorim et al. [4] developed for DFuzz [23], a dependently-typed extension of Fuzz. Given an environment Γ, a term e, and a type σ, the goal of type checking is to determine if a derivation $\Gamma \vdash e : \sigma$ exists. For sensitivity type systems, type checking and type inference can be achieved by solving the sensitivity inference problem. The sensitivity inference problem is defined using *context skeletons* $\Gamma^\bullet$ which are partial maps from variables to $\Lambda_{\mathbf{num}}$ types. If we denote by $\overline{\Gamma}$ the context Γ with all sensitivity assignments removed, then the sensitivity inference problem is defined [4, Definition 5] as follows.

Definition 6.1 (Sensitivity Inference). Given a skeleton $\Gamma^\bullet$ and a term e, the *sensitivity inference problem* computes an environment Γ and a type σ with a derivation $\Gamma \vdash e : \sigma$ such that $\Gamma^\bullet = \overline{\Gamma}$.

Given a term e and a skeleton environment $\Gamma^\bullet$, the algorithm produces an environment $\Gamma^\bullet$ with sensitivity information and a type σ. Calls to the algorithm are written as $\Gamma^\bullet; e \Rightarrow \Delta; \sigma$. Every step of the algorithm corresponds to a derivation in $\Lambda_{\mathbf{num}}$. The syntax of the algorithmic rules differs from the syntax of $\Lambda_{\mathbf{num}}$ (Figure 2) in two places: the argument of lambda terms require type annotations ($x : \sigma$), and the box constructor requires a sensitivity annotation ($[v\{s\}]$). The algorithmic rules for these constructs are as follows:

$$\frac{\Gamma^\bullet; v \Rightarrow \Gamma; \sigma}{\Gamma^\bullet; [v\{s\}] \Rightarrow s * \Gamma; !_s\sigma}\ (!\ \mathrm{I}) \qquad \frac{\Gamma^\bullet, x : \sigma; e \Rightarrow \Gamma, x :_s \sigma; \tau \qquad s \geq 1}{\Gamma^\bullet; \lambda(x : \sigma).e \Rightarrow \Gamma; \sigma \multimap \tau}\ (\multimap \mathrm{I})$$

Importantly, the type checking algorithm for $\Lambda_{\mathbf{num}}$ is sound:

Theorem 6.2 (Algorithmic Soundness). *If* $\Gamma^\bullet; e \Rightarrow \Gamma; \sigma$ *then there exists a derivation* $\Gamma \vdash e : \sigma$.

6.2 Evaluation

In order to serve as a practical tool, our type-checker must infer useful error bounds within a reasonable amount of time. Our empirical evaluation therefore focuses on measuring two key properties: tightness of the inferred error bounds and performance. To this end, our evaluation includes a comparison in terms of relative error and performance to two popular tools that soundly and automatically bound relative error: FPTaylor [49] and Gappa [21]. Although Daisy [17] and Rosa [19] also compute relative error bounds, they do not compute error bounds for the directed rounding modes, and our instantiation of $\Lambda_{\mathbf{num}}$ requires round towards $+\infty$ (see Section 5). For our comparison to Gappa and FPTaylor, we use benchmarks from FPBench [16], which is the standard set of benchmarks used in the domain; we also include the Horner scheme discussed in Section 5.

Table 1. Comparison of $\Lambda_{\mathbf{num}}$ to FPTaylor and Gappa. The Bound column gives upper bounds on relative error (smaller is better); the bounds for FPTaylor and Gappa assume all variables are in $[0.1, 1000]$. The Ratio column gives the ratio of $\Lambda_{\mathbf{num}}$'s relative error bound to the tightest (best) bound of the other two tools; values less than 1 indicate that $\Lambda_{\mathbf{num}}$ provides a tighter bound. The Ops column gives the number of operations in each benchmark. Benchmarks from FPBench are marked with a (*).

Benchmark	Ops	Bound			Ratio	Timing (ms)		
		$\Lambda_{\mathbf{num}}$	FPTaylor	Gappa		$\Lambda_{\mathbf{num}}$	FPTaylor	Gappa
hypot*	4	5.55e-16	**5.17e-16**	3.85e-12	1.07	2	100	20
x_by_xy*	3	**4.44e-16**	fail	2.22e-12	1.0e-04	1	-	10
one_by_sqrtxx	3	**5.55e-16**	5.09e-13	3.33e-12	1.1e-03	2	30	20
sqrt_add*	5	9.99e-16	**6.66e-16**	5.93e+01	1.5	4	30	20
test02_sum8*	8	**1.55e-15**	4.66e-14	5.97e-12	3.4e-02	1	1.4e4	40
nonlin1*	2	4.44e-16	4.49e-16	2.44e-15	1	1	30	10
test05_nonlin1*	2	4.44e-16	4.46e-16	2.02e-13	1	1	20	10
verhulst*	4	8.88e-16	**7.38e-16**	3.67e-09	0.83	1	30	20
predatorPrey*	7	**1.55e-15**	1.64e-11	7.15e-02	9.5e-05	2	70	30
test06_sums4_sum1*	4	6.66e-16	6.71e-16	2.84e-12	1	1	2e3	20
test06_sums4_sum2*	4	**6.66e-16**	1.78e-14	2.27e-12	4e-02	1	9e3	20
i4*	4	4.44e-16	4.50e-16	1.01e-12	1	1	150	20
Horner2	4	**4.44e-16**	6.49e-11	9.02e+09	6.8e-06	1	9.7e3	20
Horner2_with_error	4	**1.55e-15**	1.61e-10	9.02e+09	9.6e-06	2	1.6e4	40
Horner5	10	**1.11e-15**	5.03	9.02e+18	2.2e-16	1	1.9e4	40
Horner10	20	**2.22e-15**	1.14e+16	9.01e+33	2.5e-49	2	3.9e4	86
Horner20	40	**4.44e-15**	2.65e+49	9.01e+63	1.7e-64	3	1.0e5	458

There are limitations, summarized below, to the arithmetic operations that the instantiation of $\Lambda_{\mathbf{num}}$ used in our type-checker can handle, so we are only able to evaluate a subset of the FPBench benchmarks. Even so, larger examples with more than 50 floating-point operations are intractable for most tools [20], including FPTaylor and Gappa, and are not part of FPBench. Our evaluation therefore includes larger examples with well-known relative error bounds that we compare against. Finally, we used our type-checker to analyze the rounding error of four floating-point conditionals.

Our experiments were performed on a MacBook with a 1.4 GHz processor and 8 GB of memory. Relative error bounds are derived from the relative precision computed by $\Lambda_{\mathbf{num}}$ using Equation (8).

6.2.1 Limitations of Λ_{num}. Soundness of the error bounds inferred by our type-checker is guaranteed by Corollary 4.12 and the instantiation of $\Lambda_{\mathbf{num}}$ described in Section 5. This instantiation imposes the following limitations on the benchmarks we can consider in our evaluation. First, only the operations $+$, $*$, $/$, and sqrt are supported by our instantiation, so we can't use benchmarks with subtraction or transcendental functions. Second, all constants and variables must be strictly positive numbers, and the rounding mode must be fixed as round towards $+\infty$. These limitations follow from the fact that the RP metric (Definition 2.2) is only well-defined for non-zero values of the same sign. We leave the exploration of tradeoffs between the choice of metric and the primitive operations that can be supported by the language to future work. Given these limitations, along with the fact that $\Lambda_{\mathbf{num}}$ does not currently support programs with loops, we were able to include 13 of the 129 unique (at the time of writing) benchmarks from FPBench in our evaluation.

6.2.2 Small Benchmarks. The results for benchmarks with fewer than 50 floating-point operations are given in Table 1. Eleven of the seventeen benchmarks are taken from the FPBench benchmarks.

Table 2. The performance of $\Lambda_{\mathbf{num}}$ on benchmarks with 100 or more floating-point operations. The Std. column gives relative error bounds from the literature. Benchmarks from Satire are marked with with an (a); the Satire subcolumn gives timings for the computation of *absolute* error bounds as reported in [20].

Benchmark	Ops	Bound ($\Lambda_{\mathbf{num}}$)	Bound (Std.)	Timing (s)	
				$\Lambda_{\mathbf{num}}$	Satire
Horner50[a]	100	1.11e-14	1.11e-14	9e-03	5
MatrixMultiply4	112	1.55e-15	8.88e-16	3e-03	-
Horner75	150	1.66e-14	1.66e-14	2e-02	-
Horner100	200	2.22e-14	2.22e-14	4e-02	-
SerialSum[a]	1023	2.27e-13	2.27e-13	5	5407
Poly50[a]	1325	2.94e-13	-	2.12	3
MatrixMultiply16	7936	6.88e-15	3.55e-15	4e-02	-
MatrixMultiply64[a]	520192	2.82e-14	1.42e-14	10	65
MatrixMultiply128[a]	4177920	5.66e-14	2.84e-14	1080	763

Both FPTaylor and Gappa require user provided interval bounds on the input variables in order to compute the relative error; we used an interval of $[0.1, 1000]$ for each of the benchmarks. We used the default configuration for FPTaylor, and used Gappa without providing hints for interval subdivision. The floating-point format of each benchmark is binary64, and the rounding mode is set at round towards $+\infty$; the unit roundoff in this setting is 2^{-52} (approximately 2.22e-16). Only `Horner2_with_error` assumes error in the inputs.

6.2.3 Large Benchmarks. Table 2 shows the results for benchmarks with 100 or more floating-point operations. Five of the nine benchmarks are taken from Satire [20], an *empirically sound* static analysis tool that computes absolute error bounds. Although Satire does not statically compute relative error bounds for the benchmarks listed in Table 2, most of these benchmarks have well-known worst case relative error bounds that we can compare against. These bounds are given in the Std. column in Table 2; the relevant references are as follows: Horner's scheme [cf. 31, p. 95], summation [cf. 8, p. 260], and matrix multiply [cf. 31, p. 63]. For matrix multiplication, we report the max element-wise relative error bound produced by $\Lambda_{\mathbf{num}}$. When available, the Timing column in Table 2 lists the time reported for Satire to compute *absolute* error bounds [cf. 20, Table III].

6.2.4 Conditional Benchmarks. Table 3 shows the results for conditional benchmarks. Two of the four benchmarks are taken from FPBench and the remaining benchmarks are examples from Dahlquist and Björck [cf. 14, p. 119]. We were unable to compare the performance and computed relative error bounds shown in Table 3 against other tools. While Daisy, FPTaylor, and Gappa compute relative error bounds, they don't handle conditionals. And, while PRECiSA can handle conditionals, it doesn't compute relative error bounds. Only Rosa computes relative error bounds for floating-point conditionals, but Rosa doesn't compute bounds for the directed rounding modes.

6.2.5 Evaluation Summary. We draw three main conclusions from our evaluation.

Roundoff error analysis via type checking is fast. On small and conditional benchmarks, $\Lambda_{\mathbf{num}}$ infers an error bound in the order of milliseconds. This is at least an order of magnitude faster than either Gappa or FPTaylor. On larger benchmarks, $\Lambda_{\mathbf{num}}$'s performance surpasses that of comparable tools by computing bounds for problems with up to 520k operations in under a minute.

Table 3. The performance of $\Lambda_{\mathbf{num}}$ on conditional benchmarks. Benchmarks from FPBench are marked with with a (∗). Benchmarks from Dahlquist and Björck [cf. 14, p. 119] are marked with with a (b).

Benchmark	Bound	Timing (ms)
PythagoreanSum[b]	8.88e-16	2
HammarlingDistance[b]	1.11e-15	2
squareRoot3*	4.44e-16	2
squareRoot3Invalid*	4.44e-16	2

Roundoff error bounds derived via type checking are useful. On most small benchmarks $\Lambda_{\mathbf{num}}$ produces a tighter relative error bound than either FPTaylor or Gappa. On the few benchmarks where FPTaylor computes a tighter bound, $\Lambda_{\mathbf{num}}$'s results are still well within an order of magnitude. For benchmarks where rounding errors are composed and magnified, such as `Horner2_with_error`, and on somewhat larger benchmarks like `Horner2-Horner20`, our type-based approach performs particularly well. On larger benchmarks that are intractable for the other tools, $\Lambda_{\mathbf{num}}$ produces bounds that are nearly optimal in comparison to those from the literature. $\Lambda_{\mathbf{num}}$ is also able to provide non-trivial relative error bounds for floating-point conditionals.

Roundoff error bounds derived via type checking are strong. The relative error bounds produced by $\Lambda_{\mathbf{num}}$ hold for all positive real inputs, assuming the absence of overflow and underflow. In comparison, the relative error bounds derived by FPTaylor and Gappa only hold for the user provided interval bounds on the input variables, which we took to be $[0.1, 1000]$. Increasing this interval range allows FPTaylor and Gappa to give stronger bounds, but can also lead to slower analysis. Furthermore, given that relative error is poorly behaved for values near zero, some tools are sensitive to the choice of interval. We see this in the results for the benchmark `x_by_xy` in Table 1, where we are tasked with calculating the roundoff error produced by the expression $x/(x+y)$, where x and y are binary64 floating-point numbers in the interval $[0.1, 1000]$. For these parameters, the expression lies in the interval [5.0e-05, 1.0] and the relative error should still be well defined. However, FPTaylor (used with its default configuration) fails to provide a bound, and issues a warning due to the value of the expression being too close to zero.

Remark (User specified Input Ranges). *Allowing users to specify input ranges is a feature of many tools used for floating-point error analysis, including FPTaylor and Gappa. In some cases, a useful bound can't be computed for an unbounded range, but can be computed given a well-chosen bounded range for the inputs. Input ranges are also required for computing absolute error bounds. Extending $\Lambda_{\mathbf{num}}$ with bounded range inputs is left to future work; we note that this feature could be supported by adding a new type to the language, and by adjusting the types of primitive operations.*

7 EXTENDING THE NEIGHBORHOOD MONAD

So far, we have seen how the graded neighborhood monad can model rounding error when the rounding operation follows a standard rounding rule (round towards $+\infty$) and assuming that underflow and overflow do not occur. In this section, we propose variations of this monad to support error analysis for rounding operations with more complex behavior.

7.1 Extension: Non-Normal Numbers

In practice, rounding the result of a floating-point operation might result in a *non-normal* value: numbers that are not too small (*underflow*) or too large (*overflow*) for the size of floating-point representation. For a more realistic model of rounding, we can adjust the semantics of our language to accurately model non-normal values.

Extending the graded monad. First, we extend the neighborhood monad with exceptional values.

Definition 7.1. Let the pre-ordered monoid $\mathcal{R}$ be the extended non-negative real numbers $\mathbb{R}^{\geq 0} \cup \{\infty\}$ with the usual order and addition, and let $\diamond$ be a special element representing an exceptional value. The *exceptional neighborhood monad* is defined by the functors $\{T_r^* : \mathbf{Met} \to \mathbf{Met} \mid r \in \mathcal{R}\}$:

- $T_r^* : \mathbf{Met} \to \mathbf{Met}$ maps a metric space M to the metric space with underlying set

$$|T_r^* M| \triangleq \{(x, y) \in M \times (M \cup \{\diamond\}) \mid d_M(x, y) \leq r \text{ or } y = \diamond\}$$

  and metric

$$\begin{cases} d_{T_r^* M}((x, y), (x', y')) & \triangleq d_M(x, x') \\ d_{T_r^* M}((x, y), \diamond) & \triangleq 0 \end{cases}$$

- T_r^* takes a non-expansive function $f : A \to B$ to a function $T_r^* f : T_r^* A \to T_r^* B$ defined via:

$$(T_r^* f)((x, y)) \triangleq \begin{cases} (f(x), f(y)) & : y \in A \\ (f(x), \diamond) & : y = \diamond \end{cases}$$

T_r^* are evidently functors, and the associated maps are natural transformations.

Extending the **Met** *semantics.* We can define the exceptional metric semantics $[\![\Gamma \vdash e : \tau]\!]^* : [\![\Gamma]\!] \to [\![\tau]\!]$ as before (Definition 4.5), using the monad T_r^* instead of T_r. The only change is that rounding operations can now produce exceptional values. We assume that rounding is interpreted by a function $\rho^* : R \to (R \cup \{\diamond\})$ where $\diamond$ represents any exceptional value (e.g., underflow or overflow). We continue to assume that the numeric type is interpreted by a metric space $[\![\mathbf{num}]\!] = (R, d_R)$. For all numbers $r \in R$, we require that ρ^* satisfy $d_R(r, \rho^*(r)) \leq \epsilon$ whenever $\rho^*(r)$ is not the value $\diamond$.

Letting $f = [\![\Gamma \vdash k : \mathbf{num}]\!]^*$, we then define $[\![\Gamma \vdash \mathbf{rnd}\ k : M_\epsilon \mathbf{num}]\!]^* \triangleq f; \langle id, \rho^* \rangle$

Extending the FP semantics. To account for the floating point operational semantics possibly producing exception values, we introduce a new error value with a new typing rule:

$$v, w ::= \cdots \mid \mathbf{err} \qquad\qquad \frac{}{\Gamma \vdash \mathbf{err} : M_u \tau}\ \textsc{Err}$$

We only consider this value for the floating-point semantics—programs under the metric and real semantics cannot use **err**, and never step to **err**.

To interpret the monadic type in the floating-point semantics, we use the Maybe monad:

$$(\!|M_u \tau|\!)^*_{fp} \triangleq (\!|\tau|\!)^*_{fp} \uplus \{\diamond\}$$

The floating-point semantics remains the same as before (Definition 4.8) except for two changes. First, we interpret the rule Err by letting $(\!|\Gamma \vdash \mathbf{err} : M_u \tau|\!)^*_{fp}$ be the constant function producing $\diamond$. Second, given $f = (\!|\Gamma \vdash k : \mathbf{num}|\!)^*_{fp}$, we define $(\!|\Gamma \vdash \mathbf{rnd}\ k : M_\epsilon \mathbf{num}|\!)^*_{fp} \triangleq f; \rho^*$. Note that the function ρ^* may produce the exceptional value $\diamond$.

On the operational side, we modify the evaluation rule for round:

$$\mathbf{rnd}\ k \mapsto_{fp} \begin{cases} r & : \rho^*(k) = r \in R \\ \mathbf{err} & : \rho^*(k) = \diamond \end{cases}$$

And add a new step rule for propagating exceptional values: $\mathbf{let\text{-}bind}(\mathbf{err}, x.f) \mapsto_{fp} \mathbf{err}$.

Establishing error soundness. The following analogue of Corollary 4.12 follows from the analogue to the paired soundness theorem (Lemma 4.11).

Corollary 7.2. *Let* $\cdot \vdash e : M_r\mathbf{num}$ *be well-typed. Under the exceptional semantics, either:* $e \mapsto_{id} \mathbf{ret}\ v_{id}$ *and* $e \mapsto_{fp} \mathbf{ret}\ v_{fp}$*, and* $d_{[\![\mathbf{num}]\!]}([\![v_{id}]\!]^*, [\![v^*_{fp}]\!]) \leq r$*, or* $e \mapsto_{fp} \mathbf{err}$*.*

Thus, the error bound holds assuming floating point evaluation does not hit an exceptional value.

7.2 Further Extension

The exceptional neighborhood monad can be viewed as the composition of two monads: the neighborhood monad on **Met** models distance bounds, while the Maybe monad on **Set** models exceptional behavior. By replacing the Maybe monad with monads for other effects, we can define variants of the neighborhood monad modeling non-deterministic and probabilistic rounding.

8 RELATED WORK

Type systems for floating-point error. A type system due to Martel [40] uses dependent types to track the occurrence and propagation of *representation errors*; i.e., error due to the fact that floating-point numbers do not exactly represent real numbers. In $\Lambda_{\mathbf{num}}$, both representation error and *roundoff error*—the error due to rounding the results of operations—are accounted for by the type system. A significant difference between $\Lambda_{\mathbf{num}}$ and the type system proposed by Martel is error soundness. In Martel's system, the soundness result says that a semantic relation capturing the notion of accuracy between a floating-point expression and its ideal counterpart is preserved by a reduction relation. This is weaker than a standard type soundness guarantee. In particular, it is not shown that well-typed terms satisfy the semantic relation. In $\Lambda_{\mathbf{num}}$, the central novel property guaranteed by our type system is much stronger: well-typed programs of monadic type satisfy the error bound indicated by their type.

Program analysis for roundoff error. Many verification methods have been developed to automatically bound roundoff error. The earliest tools, like Fluctuat [27] and Gappa [21], employ abstract interpretation with interval arithmetic or polyhedra [11] to overapproximate the range of roundoff errors. This method is flexible and applies to general programs with conditionals and loops, but it can significantly overestimate roundoff error, and it is difficult to model cancellation of errors.

To provide more precise bounds, recent work relies on optimization. Conceptually, these methods bound the roundoff error by representing the error symbolically as a function of the program inputs and the error variables introduced during the computation, and then perform global optimization over all settings of the errors. Since the error expressions are typically complex, verification methods use approximations to simplify the error expression to make optimization more tractable, and mostly focus on straight-line programs. For instance, Real2Float [39] separates the error expression into a linear term and a non-linear term; the linear term is bounded using semidefinite programming, while the non-linear term is bounded using interval arithmetic. FPTaylor [49] was the first tool to use Taylor approximations of error expressions. Abbasi and Darulova [1] describe a modular method for bounding the propagation of errors using Taylor approximations, and Rosa [18, 19] uses Taylor series to approximate the propagation of errors in possibly non-linear terms.

In contrast, our type system does not rely on global optimization, can naturally accommodate both relative and absolute error, and can be instantiated to different models of floating-point arithmetic with minimal changes. Our language supports a variety of datatypes and higher-order functions. While our language does not support recursive types and general recursion, similar languages support these features [5, 15, 47] and we expect they should be possible in $\Lambda_{\mathbf{num}}$; however, the precision of the error bounds for programs using general recursion might be poor. Another limitation of our method is in typing conditionals: while $\Lambda_{\mathbf{num}}$ can only derive error bounds when the ideal and floating-point executions follow the same branch, tools that use general-purpose solvers (e.g., PRECiSA and Rosa) can produce error bounds for programs where the ideal and floating-point executions take different branches.

Verification and synthesis for numerical computations. Formal verification has a long history in the area of numerical computations, starting with the pioneering work of Harrison [28–30]. Formalized specifications of floating-point arithmetic have been developed in the Coq [9], Isabelle [52], and PVS [41] proof assistants. These specifications have been used to develop sound tools for floating-point error analysis that generate proof certificates, such as VCFloat [2, 46] and PRECiSA [51]. They have also been used to mechanize proofs of error bounds for specific numerical programs (e.g., [7, 34, 35, 43, 50]). Work by Cousot et al. [13] has applied abstract interpretation to verify the absence of floating-point faults in flight-control software, which have caused real-world accidents. Finally, recent work uses *program synthesis*: Herbie [45] automatically rewrites numerical programs to reduce numerical error, while RLibm [38] automatically generates correctly-rounded math libraries.

Type systems for sensitivity analysis. $\Lambda_{\mathbf{num}}$ belongs to a line of work on linear type systems for sensitivity analysis, starting with Fuzz [47]. We point out a few especially relevant works. Our syntax and typing rules are inspired by Dal Lago and Gavazzo [15], who propose a family of Fuzz-like languages and define various notions of operational equivalence; we are inspired by their syntax, but our case elimination rule (+ E) is different: we require s to be strictly positive when scaling the conclusion. This change is due to a subtle difference in how sums are treated. Azevedo de Amorim et al. [6] added a graded monadic type to Fuzz to handle more complex variants of differential privacy; in their application, the grade does not interact with the sensitivity language.

Other approaches to error analysis. The numerical analysis literature has explored other conceptual tools for static error analysis, such as stochastic error analysis [12]. Techniques for *dynamic* error analysis, which estimate the rounding error at runtime, have also been proposed [31].

9 CONCLUSION AND FUTURE DIRECTIONS

We have presented a type system for bounding roundoff error, combining two elements: a sensitivity analysis through a linear type system, and error tracking through a novel graded monad. Our work demonstrates that type systems can reason about quantitative roundoff error. There is a long history of research in error analysis, and we believe that we are just scratching the surface of what is possible with type-based approaches.

We briefly comment on two promising directions. First, numerical analysts have studied probabilistic models of roundoff errors, which can give better bounds on error in practice [32]. Combining our system with a probabilistic language might enable a similar analysis. Second, the error bounds we establish are *forward error* bounds, because they bound the error in the output. In practice, numerical analysts often consider *backward error* bounds, which describe how much the

input needs to be perturbed in order to realize the approximate output. Such bounds can help clarify whether the source of the error is due to the computation, or inherent in the problem instance. Tackling this kind of property is an interesting direction for future work.

ARTIFACT

The artifact for the implementation of $\Lambda_{\mathbf{num}}$ described in Section 6 is available online [36]. It includes instructions on how to reproduce the results reported in Tables 1 to 3.

ACKNOWLEDGMENTS

We thank Pedro Henrique Azevedo de Amorim, David Bindel, and the anonymous reviewers for their close reading and useful suggestions. Preliminary versions of this work were presented at Cornell's PLDG seminar and the NJ Programming Languages and Systems Seminar. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Department of Energy Computational Science Graduate Fellowship under Award Number DE-SC0021110. This work was also partially supported by the NSF (#1943130) and the ONR (#N00014-23-1-2134).

REFERENCES

[1] Rosa Abbasi and Eva Darulova. 2023. Modular Optimization-Based Roundoff Error Analysis of Floating-Point Programs. In *International Symposium on Static Analysis (SAS), Cascais, Portugal (Lecture Notes in Computer Science, Vol. 14284)*. Springer, 41–64. https://doi.org/10.1007/978-3-031-44245-2_4

[2] Andrew W. Appel and Ariel E. Kellison. 2024. VCFloat2: Floating-Point Error Analysis in Coq. In *ACM SIGPLAN Conference on Certified Proofs and Programs (CPP), London, England*. 14–29. https://doi.org/10.1145/3636501.3636953

[3] Steve Awodey. 2010. *Category Theory* (2nd ed.). Oxford University Press.

[4] Arthur Azevedo de Amorim, Marco Gaboardi, Emilio Jesús Gallego Arias, and Justin Hsu. 2014. Really Natural Linear Indexed Type-Checking. In *Symposium on Implementation and Application of Functional Programming Languages (IFL), Boston, Massachusetts*. ACM Press, 5:1–5:12. https://doi.org/10.1145/2746325.2746335 arXiv:1503.04522 [cs.LO]

[5] Arthur Azevedo de Amorim, Marco Gaboardi, Justin Hsu, Shin-ya Katsumata, and Ikram Cherigui. 2017. A semantic account of metric preservation. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Paris, France*. 545–556. https://doi.org/10.1145/3009837.3009890

[6] Arthur Azevedo de Amorim, Marco Gaboardi, Justin Hsu, and Shin-ya Katsumata. 2019. Probabilistic Relational Reasoning via Metrics. In *IEEE Symposium on Logic in Computer Science (LICS), Vancouver, British Columbia*. 1–19. https://doi.org/10.1109/LICS.2019.8785715

[7] Sylvie Boldo, François Clément, Jean-Christophe Filliâtre, Micaela Mayero, Guillaume Melquiond, and Pierre Weis. 2014. Trusting computations: A mechanized proof from partial differential equations to actual program. *Comput. Math. Appl.* 68, 3 (2014), 325–352. https://doi.org/10.1016/J.CAMWA.2014.06.004

[8] Sylvie Boldo, Claude-Pierre Jeannerod, Guillaume Melquiond, and Jean-Michel Muller. 2023. Floating-point arithmetic. *Acta Numerica* 32 (2023), 203–290. https://doi.org/10.1017/S0962492922000101

[9] Sylvie Boldo and Guillaume Melquiond. 2011. Flocq: A Unified Library for Proving Floating-Point Algorithms in Coq. In *IEEE Symposium on Computer Arithmetic (ARITH), Tübingen, Germany*. 243–252. https://doi.org/10.1109/ARITH.2011.40

[10] Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. 2014. A Core Quantitative Coeffect Calculus. In *European Symposium on Programming (ESOP), Grenoble, France (Lecture Notes in Computer Science, Vol. 8410)*. Springer-Verlag, 351–370. https://doi.org/10.1007/978-3-642-54833-8_19

[11] Liqian Chen, Antoine Miné, and Patrick Cousot. 2008. A Sound Floating-Point Polyhedra Abstract Domain. In *Asian Symposium on Programming Languages and Systems (APLAS), Bangalore, India (Lecture Notes in Computer Science, Vol. 5356)*. Springer-Verlag, 3–18. https://doi.org/10.1007/978-3-540-89330-1_2

[12] Michael P. Connolly, Nicholas J. Higham, and Theo Mary. 2021. Stochastic Rounding and Its Probabilistic Backward Error Analysis. *SIAM Journal on Scientific Computing* 43, 1 (2021), A566–A585. https://doi.org/10.1137/20M1334796 arXiv:https://doi.org/10.1137/20M1334796

[13] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2005. The ASTREÉ Analyzer. In *European Symposium on Programming (ESOP), Edinburgh, Scotland (Lecture Notes in Computer Science, Vol. 3444)*. Springer-Verlag, 21–30. https://doi.org/10.1007/978-3-540-31987-0_3

[14] Germund Dahlquist and Åke Björck. 2008. *Numerical Methods in Scientific Computing, Volume I.* Society for Industrial and Applied Mathematics. https://doi.org/10.1137/1.9780898717785

[15] Ugo Dal Lago and Francesco Gavazzo. 2022. A Relational Theory of Effects and Coeffects. 6, POPL, Article 31 (Jan. 2022). https://doi.org/10.1145/3498692

[16] Nasrine Damouche, Matthieu Martel, Pavel Panchekha, Chen Qiu, Alexander Sanchez-Stern, and Zachary Tatlock. 2017. Toward a Standard Benchmark Format and Suite for Floating-Point Analysis. In *International Workshop on Numerical Software Verification (NSV) , Toronto, Ontario*. Springer-Verlag, 63–77.

[17] Eva Darulova, Anastasiia Izycheva, Fariha Nasir, Fabian Ritter, Heiko Becker, and Robert Bastian. 2018. Daisy - Framework for Analysis and Optimization of Numerical Programs (Tool Paper). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Thessaloniki, Greece (Lecture Notes in Computer Science, Vol. 10805)*. Springer-Verlag, 270–287. https://doi.org/10.1007/978-3-319-89960-2_15

[18] Eva Darulova and Viktor Kuncak. 2014. Sound Compilation of Reals. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), San Diego, California*. 235–248. https://doi.org/10.1145/2535838.2535874

[19] Eva Darulova and Viktor Kuncak. 2017. Towards a Compiler for Reals. *ACM Trans. Program. Lang. Syst.* 39, 2, Article 8 (March 2017). https://doi.org/10.1145/3014426

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

[21] Marc Daumas and Guillaume Melquiond. 2010. Certification of bounds on expressions involving rounded operators. *ACM Trans. Math. Softw.* 37, 1 (2010), 2:1–2:20. https://doi.org/10.1145/1644001.1644003

[22] Soichiro Fujii, Shin-ya Katsumata, and Paul-André Melliès. 2016. Towards a Formal Theory of Graded Monads. In *International Conference on Foundations of Software Science and Computation Structures (FoSSaCS), Eindhoven, The Netherlands (Lecture Notes in Computer Science, Vol. 9634)*. Springer-Verlag, 513–530. https://doi.org/10.1007/978-3-662-49630-5_30

[23] Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C. Pierce. 2013. Linear Dependent Types for Differential Privacy. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Rome, Italy*. 357–370. https://doi.org/10.1145/2429069.2429113

[24] Marco Gaboardi, Shin-ya Katsumata, Dominic A. Orchard, Flavien Breuvart, and Tarmo Uustalu. 2016. Combining effects and coeffects via grading. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Nara, Japan*. 476–489. https://doi.org/10.1145/2951913.2951939

[25] Jean-Yves Girard, Andre Scedrov, and Philip J. Scott. 1992. Bounded Linear Logic: A Modular Approach to Polynomial-Time Computability. *Theor. Comput. Sci.* 97, 1 (1992), 1–66. https://doi.org/10.1016/0304-3975(92)90386-T

[26] David Goldberg. 1991. What Every Computer Scientist Should Know about Floating-Point Arithmetic. *ACM Comput. Surv.* 23, 1 (March 1991), 5–48. https://doi.org/10.1145/103162.103163

[27] Eric Goubault and Sylvie Putot. 2006. Static Analysis of Numerical Algorithms. In *International Symposium on Static Analysis (SAS), Seoul, Korea (Lecture Notes in Computer Science, Vol. 4134)*. Springer-Verlag, 18–34. https://doi.org/10.1007/11823230_3

[28] John Harrison. 1997. Floating Point Verification in HOL Light: The Exponential Function. In *International Conference on Algebraic Methodology and Software Technology (AMAST), Sydney, Australia (Lecture Notes in Computer Science, Vol. 1349)*. Springer-Verlag, 246–260. https://doi.org/10.1007/BFB0000475

[29] John Harrison. 1999. A Machine-Checked Theory of Floating Point Arithmetic. In *International Conference on Theorem Proving in Higher Order Logics (TPHOLs), Nice, France (Lecture Notes in Computer Science, Vol. 1690)*. Springer-Verlag, 113–130. https://doi.org/10.1007/3-540-48256-3_9

[30] John Harrison. 2000. Formal Verification of Floating Point Trigonometric Functions. In *International Conference on Formal Methods in Computer-Aided Design (FMCAD), Austin, Texas (Lecture Notes in Computer Science, Vol. 1954)*. Springer-Verlag, 217–233. https://doi.org/10.1007/3-540-40922-X_14

[31] Nicholas J. Higham. 2002. *Accuracy and Stability of Numerical Algorithms* (2nd ed.). Society for Industrial and Applied Mathematics. https://doi.org/10.1137/1.9780898718027 arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9780898718027

[32] Nicholas J. Higham and Theo Mary. 2019. A New Approach to Probabilistic Rounding Error Analysis. *SIAM Journal on Scientific Computing* 41, 5 (2019), A2815–A2835. https://doi.org/10.1137/18M1226312

[33] Shin-ya Katsumata. 2014. Parametric effect monads and semantics of effect systems. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), San Diego, California*. 633–646. https://doi.org/10.1145/2535838.2535846

[34] Ariel E. Kellison and Andrew W. Appel. 2022. Verified Numerical Methods for Ordinary Differential Equations. In *International Workshop on Numerical Software Verification (NSV) , Haifa, Israel (Lecture Notes in Computer Science, Vol. 13466)*. Springer-Verlag, 147–163. https://doi.org/10.1007/978-3-031-21222-2_9

[35] Ariel E. Kellison, Andrew W. Appel, Mohit Tekriwal, and David Bindel. 2023. LAProof: A Library of Formal Proofs of Accuracy and Correctness for Linear Algebra Programs. In *IEEE Symposium on Computer Arithmetic (ARITH), Portland, Oregon*. 36–43. https://doi.org/10.1109/ARITH58626.2023.00021

[36] Ariel E. Kellison and Justin Hsu. 2024. *Numerical Fuzz: A Type System for Rounding Error Analysis*. https://doi.org/10.5281/zenodo.10802849

[37] Tom Leinster. 2014. *Basic Category Theory*. Cambridge University Press. https://doi.org/10.1017/CBO9781107360068

[38] Jay P. Lim and Santosh Nagarakatte. 2022. One polynomial approximation to produce correctly rounded results of an elementary function for multiple representations and rounding modes. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 1–28. https://doi.org/10.1145/3498664

[39] Victor Magron, George A. Constantinides, and Alastair F. Donaldson. 2017. Certified Roundoff Error Bounds Using Semidefinite Programming. *ACM Trans. Math. Softw.* 43, 4 (2017), 34:1–34:31. https://doi.org/10.1145/3015465

[40] Matthieu Martel. 2018. Strongly Typed Numerical Computations. In *International Conference on Formal Methods and Software Engineering (ICFEM), Gold Coast, Australia (Lecture Notes in Computer Science, Vol. 11232)*. Springer-Verlag, 197–214. https://doi.org/10.1007/978-3-030-02450-5_12

[41] Paul S. Miner. 1995. Formal Specification of IEEE Floating-Point Arithmetic Using PVS. In *IFAC Workshop on Safety and Reliability in Emerging Control Technologies, Daytona Beach, Florida*, Vol. 28. 31–36. https://doi.org/10.1016/S1474-6670(17)44820-8

[42] Eugenio Moggi. 1991. Notions of Computation and Monads. *Inf. Comput.* 93, 1 (1991), 55–92. https://doi.org/10.1016/0890-5401(91)90052-4

[43] Mariano M. Moscato, Laura Titolo, Marco A. Feliú, and César A. Muñoz. 2019. Provably Correct Floating-Point Implementation of a Point-in-Polygon Algorithm. In *Formal Methods – The Next 30 Years (FM), Porto, Portugal*. Springer-Verlag, 21–37. https://doi.org/10.1007/978-3-030-30942-8_3

[44] F. W. J. Olver. 1978. A New Approach to Error Arithmetic. *SIAM J. Numer. Anal.* 15, 2 (1978), 368–393. https://doi.org/10.1137/0715024

[45] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. 2015. Automatically improving accuracy for floating point expressions. (2015), 1–11. https://doi.org/10.1145/2737924.2737959

[46] Tahina Ramananandro, Paul Mountcastle, Benoît Meister, and Richard Lethin. 2016. A Unified Coq Framework for Verifying C Programs with Floating-Point Computations. In *ACM SIGPLAN Conference on Certified Proofs and Programs (CPP), St. Petersburg, Florida*. 15–26. https://doi.org/10.1145/2854065.2854066

[47] Jason Reed and Benjamin C. Pierce. 2010. Distance makes the types grow stronger: a calculus for differential privacy. (2010), 157–168. https://doi.org/10.1145/1863543.1863568

[48] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2014. Stochastic optimization of floating-point programs with tunable precision. (2014), 53–64. https://doi.org/10.1145/2594291.2594302

[49] Alexey Solovyev, Marek S. Baranowski, Ian Briggs, Charles Jacobsen, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. 2019. Rigorous Estimation of Floating-Point Round-Off Errors with Symbolic Taylor Expansions. *ACM Transactions on Programming Languages and Systems* 41, 1 (2019), 2:1–2:39. https://doi.org/10.1145/3230733

[50] Mohit Tekriwal, Andrew W. Appel, Ariel E. Kellison, David Bindel, and Jean-Baptiste Jeannin. 2023. Verified Correctness, Accuracy, And Convergence Of a Stationary Iterative Linear Solver: Jacobi Method. In *International Conference on Intelligent Computer Mathematics (CICM), Cambridge, UK*. Springer-Verlag, 206–221. https://doi.org/10.1007/978-3-031-42753-4_14

[51] Laura Titolo, Marco A. Feliú, Mariano M. Moscato, and César A. Muñoz. 2018. An Abstract Interpretation Framework for the Round-Off Error Analysis of Floating-Point Programs. In *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI), Los Angeles, California (Lecture Notes in Computer Science, Vol. 10747)*. Springer-Verlag, 516–537. https://doi.org/10.1007/978-3-319-73721-8_24

[52] Lei Yu. 2013. A Formal Model of IEEE Floating Point Arithmetic. *Archive of Formal Proofs* (July 2013). https://isa-afp.org/entries/IEEE_Floating_Point.html, Formal proof development.