A Type System for Numerical Error Analysis

ARIEL 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 error. 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.

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 numerical optimization to find the maximum error between the approximation and the ideal computations over all possible realizations of the rounding error.

While these tools are effective, they share some drawbacks. First, most works focus primarily on *absolute error*, the absolute difference between ideal and approximate: $|x - \tilde{x}|$. When the ideal value x is large, the absolute error is less relevant—some kind of *relative* error is more informative. Second, analyses that rely on global optimization are not compositional, limiting scalability. Finally, existing tools largely focus on simple expressions, such as straight-line programs, and it is unclear how to extend existing methods to more full-featured programming languages.

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

Authors' addresses: Ariel Kellison, Cornell University, USA, ak2485@cornell.edu; Justin Hsu, Cornell University, USA, email@justinhsu.net.

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 a novel type system $\Lambda_{\mathbf{num}}$ 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 [42], 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 [38]. 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 [39]. 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).

Table 1. Parameters for floating-point number sets in the IEEE 754-2008 standard. For each, $emin = 1 - emax$.

Parameter	**binary32**	**binary64**	**binary128**
p	24	53	113
$emax$	+127	+1023	+16383

Table 2. Common Rounding Functions (modes).

Rounding mode	**Behavior**	**Notation**	**Unit Roundoff**
Round towards $+\infty$	$\min\{y \in \mathbb{F} \mid y \geq x\}$	$\rho_{RU}(x)$	β^{1-p}
Round towards $-\infty$	$\max\{y \in \mathbb{F} \mid y \leq x\}$	$\rho_{RD}(x)$	β^{1-p}
Round towards 0	$\rho_{RU}(x)$ if $x < 0$, otherwise $\rho_{RD}(x)$	$\rho_{RZ}(x)$	β^{1-p}
Round towards nearest[1]	$\{y \in \mathbb{F} \mid \forall z \in \mathbb{F}, \lvert x - y\rvert \leq \lvert x - z\rvert\}$	$\rho_{RN(x)}$	$\frac{1}{2}\beta^{1-p}$

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 [24], Higham [29], and Boldo et al. [7].

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. Parameter values for formats defined by the IEEE floating-point standard [2] are given in Table 1.

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 properties of these operators are given in Figure 2.

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 [29]

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

[1]For round towards nearest, there are several possible tie-breaking choices.

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 to handle underflow and overflow in $\Lambda_{\mathbf{num}}$ 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* (er_{abs}) and *relative error* (er_{rel}):

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

These measures do not apply uniformly to all values of x: the absolute error is well-behaved for small x, while the relative error is well-behaved for large x. The standard model (Equation (2)) implies that the basic floating-point operations have a relative error of at most unit roundoff.

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 [16, 17], Fluctuat [25], and SATIRE [18] 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 [42], 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 [39] to capture relative error in numerical analysis.

Definition 2.2 (Relative Precision (RP)). Let $\tilde{x}$ and x be nonzero real numbers of the same sign, then $\tilde{x}$ is said to be an approximation to x of *relative precision* (RP) δ if $x = \tilde{x}e^{\delta}$, where $\delta \leq \alpha$:

$$RP(x, \tilde{x}) = |ln(x/\tilde{x})| \leq \alpha. \tag{4}$$

Rewriting Equation (4) and the relative error as follows, we see that the RP metric is a close approximation to relative error, so long as $\delta \ll 1$:

$$er_{rel}(x, \tilde{x}) = |\delta|; \quad \tilde{x} = (1 + \delta)x, \tag{5}$$

$$RP(x, \tilde{x}) = |\delta|; \quad \tilde{x} = e^{\delta}x. \tag{6}$$

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. Spelling 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 [29]. 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

$$\mathbf{pow2'}(x) = \rho(x * x) = (x * x)(1 + \epsilon), \quad |\epsilon| \leq u, \tag{7}$$

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 (7): 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 $\mathbf{pow4}(a)$ as the following composition:

$$\begin{tikzcd}
 & & (c, \tilde{c}) \\
a \arrow[r, "\mathbf{pow2}'(a)"'] & (b, \tilde{b}) \arrow[ru, "\mathbf{pow2}'(b)"] \arrow[rd, "\mathbf{pow2}'(\tilde{b})"'] & \\
 & & (d, \tilde{d})
\end{tikzcd}$$

From left-to-right, the ideal and approximate results of $\mathbf{pow2}'(a)$ are b and $\tilde{b}$, respectively; the grade u on the monadic return type of $\mathbf{pow2}'$ ensures that these values are at distance at most u. The ideal result of $\mathbf{pow4}(a)$ is c, while the approximate result of $\mathbf{pow4}(a)$ 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 $\mathbf{pow2}'$ 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 $\mathbf{pow4}$ 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:

$$RP(g(x), f(y)) \leq RP(g(x), f(x)) + RP(f(x), f(y)). \tag{8}$$

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 $\mathbf{pow4}$ is $\mathbf{pow4} : !_4\mathbf{num} \multimap M_{3u}\mathbf{num}$, if we denote by g and f the approximate and ideal interpretations of $\mathbf{pow4}$, from Equation (8) we expect our type system to produce the following bound on the propagation of error in $\mathbf{pow4}$. For any exact number x and its approximation $\tilde{x}$ at distance at most u', we have:

$$RP(g(\tilde{x}), f(x)) \leq 3u + 4u'. \tag{9}$$

The term $4u'$ reflects that $\mathbf{pow4}$ 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 $\mathbf{pow4}$ to implement a function $\mathbf{pow4}'$ 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.

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

Fig. 1. Types, values, and terms.

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 [42], 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 [13]. 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 $\mathbf{op} \in \mathcal{O}$, 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 terms typed under Γ have sensitivity s to perturbations in the variable x; 0-sensitivity means that the term does not depend on x, while infinite sensitivity means that any perturbation in x can lead to arbitrarily large changes in e.

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

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

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

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

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

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

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

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

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

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

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

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.

$$\begin{array}{ll} \cdot + \cdot = \cdot & (\Gamma, x :_s \sigma) + \Delta = (\Gamma + \Delta), x :_s \sigma \text{ if } x \notin \Delta \\ \Gamma + (\Delta, x :_s \sigma) = (\Gamma + \Delta), x :_s \sigma \text{ if } x \notin \Gamma & (\Gamma + \Delta), x :_{s+t} \sigma = (\Gamma, x :_s \sigma) + (\Delta, x :_t \sigma) \end{array}$$

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 [13]. 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.

$$\pi_i\langle v_1, v_2\rangle \mapsto v_i \qquad \textbf{let } x \otimes y = (v, w) \textbf{ in } e \mapsto e[v/x][w/y]$$
$$\textbf{op}(v) \mapsto op(v) \qquad \textbf{let } [x] = [v] \textbf{ in } e \mapsto e[v/x]$$
$$(\lambda x.e)\ v \mapsto e[v/x] \qquad \textbf{let-bind}(\textbf{ret } v, x.e) \mapsto e[v/x]$$
$$\textbf{case } (\textbf{in}k\ v) \textbf{ of } (\textbf{inl } x.e_l \mid \textbf{inr } x.e_r) \mapsto e_k[v/x] \qquad (k \in \{l, r\})$$
$$\textbf{let-bind}(\textbf{let-bind}(v, x.f), y.g) \mapsto \textbf{let-bind}(v, x.\textbf{let-bind}(f, y.g)) \quad x \notin FV(g)$$
$$\frac{e \mapsto e'}{\textbf{let } x = e \textbf{ in } f \mapsto \textbf{let } x = e' \textbf{ in } f}$$

Fig. 3. Evaluation rules for $\Lambda_{\textbf{num}}$.

The rules (Subsumption), (Ret), (Rnd), and (M_u E) are the core rules for performing rounding error analysis in $\Lambda_{\textbf{num}}$. Intuitively, the monadic type $M_\epsilon\textbf{num}$ 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 $\textbf{let-bind}(v, x.f)$ 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.

Before introducing our dynamic semantics, we note that the static semantics of $\Lambda_{\textbf{num}}$ enjoy the properties of weakening and substitution, and define the notion of a subenvironment.

Definition 3.2 (Subenvironment). Γ is a *subenvironment* of Δ, written $\Gamma \sqsubseteq \Delta$, if whenever $\Delta(x) = (s, \sigma)$ for some sensitivity s and type σ, then there exists an $s \le s'$ such that $\Gamma(x) = (s', \sigma)$.

Lemma 3.3 (Weakening). *Let $\Gamma \vdash e : \tau$ be a well-typed term. Then for any typing environment $\Gamma \sqsubseteq \Delta$, there is a derivation of $\Delta \vdash e : \tau$.*

Proof. By induction on the typing derivation of $\Gamma \vdash e : \tau$. □

Lemma 3.4 (Substitution). *Let $\Gamma, \Delta \vdash e : \tau$ be a well-typed term, and let $\vec{v} : \Delta$ be a well-typed substitution of closed values, i.e., we have derivations $\vdash v_x : \Delta(x)$. Then there is a derivation of*

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

Proof. The base cases (Unit), (Const), and (Var) follow easily, and the remaining of the cases follow by applying the induction hypothesis to every premise of the relevant typing rule. □

3.3 Dynamic Semantics

We use a small-step operational semantics adapted from Fuzz [42], extended with rules for the monadic let-binding. The complete set of evaluation rules is given in Figure 3, where the judgment $e \mapsto e'$ indicates that the expression e takes a single step, resulting in the expression e'.

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

$$\mathcal{R}_\tau \triangleq \{e \mid e \in CT(\tau) \ \&\ \exists v \in CV(\tau).\ e \mapsto^* v \ \&\ v \in \mathcal{VR}_\tau\}$$

$$\mathcal{VR}_{unit} \triangleq \{\langle\rangle\} \qquad \mathcal{VR}_{\mathbf{num}} \triangleq R$$

$$\mathcal{VR}_{\sigma\times\tau} \triangleq \{\langle v, w\rangle \mid v \in \mathcal{R}_\sigma \ \&\ w \in \mathcal{R}_\tau\} \qquad \mathcal{VR}_{\sigma\otimes\tau} \triangleq \{(v, w) \mid v \in \mathcal{R}_\sigma \ \&\ w \in \mathcal{R}_\tau\}$$

$$\mathcal{VR}_{\sigma+\tau} \triangleq \{v \mid \mathbf{inl}\ v \ \&\ v \in \mathcal{R}_\sigma \text{ or } \mathbf{inr}\ v \ \&\ v \in \mathcal{R}_\tau\} \qquad \mathcal{VR}_{\sigma\multimap\tau} \triangleq \{v \mid \forall w \in \mathcal{VR}_\sigma.\ vw \in \mathcal{R}_\tau\}$$

$$\mathcal{VR}_{!_s\tau} \triangleq \{[v] \mid v \in \mathcal{R}_\tau\} \qquad \mathcal{VR}_{M_u\tau} \triangleq \bigcup_n \mathcal{VR}^n_{M_u\tau}$$

$$\mathcal{VR}^0_{M_u\tau} \triangleq \{v \mid v \equiv \mathbf{ret}\ w \ \&\ w \in \mathcal{R}_\tau \text{ or } v \equiv \mathbf{rnd}\ \ k \ \&\ k \in \mathcal{R}_{\mathbf{num}}\}$$

$$\mathcal{VR}^{n+1}_{M_u\tau} \triangleq \mathcal{R}^n_{M_u\tau} \cup \left\{\mathbf{let\text{-}bind}(v, x.f) \mid \exists\ \sigma, u_1, u_2, j.\ u \geq u_1 + u_2 \ \&\ n > j \ \&\ v \in \mathcal{VR}^j_{M_{u_1}\sigma} \right.$$
$$\left. \&\ \left(\forall w \in \mathcal{VR}_\sigma, f[w/x] \in \mathcal{VR}^{n-j}_{M_{u_2}\tau}\right)\right\}$$

Fig. 4. Reducibility Predicate

but do not reduce its size. Even so, it is possible to show that well-typed programs are terminating. If we denote the set of closed values of type τ by $CV(\tau)$ and the set of closed terms of type τ by $CT(\tau)$ so that $CV_\tau \subseteq CT(\tau)$, and define $\mapsto^*$ as the reflexive, transitive closure of the single step judgment $\mapsto$, then we can state our termination theorem as follows.

Theorem 3.5 (Termination). *If $\cdot \vdash e : \tau$ then there exists $v \in CV(\tau)$ such that $e \mapsto^* v$.*

The proof of Theorem 3.5 follows by a standard logical relations argument. Our logical relations will also show something a bit stronger: roughly speaking, programs of monadic type $M_u\tau$ can be viewed as finite-depth trees.

Definition 3.6. We define the reducibility predicate $\mathcal{R}_\tau$ inductively on types in Figure 4.

The proof of Theorem 3.5 requires two lemmas. The first is quite standard, and the definition of the reducibility predicate ensures that the proof follows without complication by induction on the derivation $\cdot \vdash e : \tau$.

Theorem 3.7. *The predicate $\mathcal{VR}$ is preserved by backward and forward reductions: if $\cdot \vdash e : \tau$ and $e \mapsto e'$ then $e \in \mathcal{R}_\tau \iff e' \in \mathcal{R}_\tau$.*

Observe that the following lemma follows without complication by induction on m, the depth of the predicate $\mathcal{VR}$.

Lemma 3.8 (Subsumption). *If $e \in \mathcal{VR}^m_{M_u\tau}$ then $e \in \mathcal{VR}^m_{M_{u'}\tau}$ supposing $u' \geq u$.*

Using Theorem 3.7 and Lemma 3.8 we can prove the following.

Lemma 3.9. *If $\cdot \vdash e : \tau$ then $e \in \mathcal{R}_\tau$.*

Proof. We first prove the following stronger statement. Let $\Gamma \triangleq x_1 :_{s_1} \sigma_1, \cdots, x_i :_{s_i} \sigma_i$ be a typing environment and let $\vec{w}$ denote the values $\vec{w} \triangleq w_1, \cdots, w_i$. If $\Gamma \vdash e : \tau$ and $w_i \in \mathcal{VR}_{\Gamma(x_i)}$ for every $x_i \in dom(\Gamma)$ then $e[\vec{w}/dom(\Gamma)] \in \mathcal{R}_\tau$. The proof follows by induction on the derivation $\Gamma \vdash e : \tau$. Let us consider the monadic cases, as the non-monadic cases are standard. The base cases (Const), (Ret), and (Rnd) follow by definition, and Subsum follows by theorem 3.8. The case for (M_u E) requires some detail. The rule is

$(M_u\ \mathrm{E})$

$$\frac{\Gamma \vdash v : M_u \sigma \qquad \Theta, x :_s \sigma \vdash f : M_{u'} \tau}{s * \Gamma + \Theta \vdash \textbf{let-bind}(v, x.f) : M_{s*u+u'} \tau}$$

and so we are required to show $\textbf{let-bind}(v, x.f) \in \mathcal{R}_{M_{s*u+u'}\tau}$. Let us denote the typing environment $s * \Gamma + \Theta$ by Δ. We apply weaking (theorem 3.3) and the induction hypothesis to each premise of the rule to obtain $f[\vec{w}/dom(\Delta)][w'/x] \in \mathcal{R}_{M_{u'}\tau}$ for any $w' \in R_\sigma$ and $v[\vec{w}/dom(\Delta)] \in \mathcal{R}_u\sigma$. The proof follows by cases on v. We will make use of the following lemma.

For any typing environment Γ, let $\Gamma e : \tau$ be a well-typed term,and let $\vec{v}_1, \vec{v}_2 : \Gamma$ be a well-typed substitutions of closed values. If $e[\vec{v}_1/dom(\Gamma)] \in \mathcal{VR}^m_{M_u\tau}$ for some $m \in \mathbb{N}$ and $e[\vec{v}_2/dom(\Gamma)] \in \mathcal{R}_{M_u\tau}$, then $e[\vec{v}_2/dom(\Gamma)] \in \mathcal{VR}^m_{M_u\tau}$.

Case: $v \equiv \textbf{rnd}\ \ k$ with $k \in \mathbb{R}$. From lemmas 3.7 and 3.4 we have that $f[\vec{w}/dom(\Delta)][r/x] \in \mathcal{VR}^m_{M_{u'}\tau}$ for some $r \in \mathbb{R}$ and $m \in \mathbb{N}$, and itfollows that $\textbf{let-bind}(v, x.f) \in \mathcal{VR}^{m+1}_{M_{s*u+u'}\tau}$.

Case: $v \equiv \textbf{ret}\ v'$ with $v' \in \mathcal{VR}_\sigma$. From the reductionrules we have that

$$\textbf{let-bind}(v, x.f)[\vec{w}/dom(\Delta)] \mapsto f[v'/x][\vec{w}/dom(\Delta)].$$

Instantiating the induction hypothesis with v' yields $f[v'/x][\vec{w}/dom(\Delta)] \in \mathcal{R}_{M_{u'}\tau}$ and the conclusion follows from lemmas 3.8, 3.7, and 3.4.

Case: $v \equiv \textbf{let-bind}(\textbf{rnd}\ \ k, y.g)$. From the reduction rules we have that

$$\textbf{let-bind}(v, x.f)[\vec{w}/dom(\Delta)] \mapsto \textbf{let-bind}(\textbf{rnd}\ \ k, y.\textbf{let-bind}(g, x.f))[\vec{w}/dom(\Delta)],$$

with $y \notin FV(f)$. From lemmas 3.7 and 3.4 we have that $f[\vec{w}/dom(\Delta)][r/x] \in \mathcal{VR}^m_{M_{u'}\tau}$ for some $r \in \mathbb{R}$ and $m \in \mathbb{N}$. From the induction hypothesis on v we can concludethat, for any $r' \in \mathbb{R}$, $g[\vec{w}/dom(\Delta)][r'/y] \in \mathcal{VR}^n_{u^*}\sigma$ for some $u^* \leq u$ and $n \in \mathbb{N}$, and it therefore follows from lemma 3.8 that $\textbf{let-bind}(\textbf{rnd}\ \ k, y.\textbf{let-bind}(g, x.f))[\vec{w}/dom(\Delta)] \in \mathcal{VR}^{(m+1)+(n+1)}_{M_{s*u+u'}\tau}$. The conclusion follows from lemmas 3.7 and 3.4. □

The following lemma clearly follows by definition of the reducibility predicate.

Lemma 3.10. *If $e \in \mathcal{R}_\tau$ then there exists a $v \in CV(\tau)$ such that $e \mapsto^* v$.*

The proof of termination (Theorem 3.5) then follows from Lemma 3.10 and Lemma 3.9.

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 languagebased 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 assume familiarity with basic concepts from category theory (e.g., categories, functors, natural transformations), and we will introduce more specialized concepts as we go along. 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 elsewhere [22].

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* [23], 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* [9, 22]. 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 also define the following associated natural transformations:

- The functor
- For $s, t \in \mathcal{S}$ and $s \leq t$, the map $(s \leq t)_A : D_t A \to D_s A$ is the identity; note the direction.
- The map $m_{s,I} : I \to D_s I$ is the identity map on the singleton metric space.

- The map $m_{s,A,B} : D_s A \otimes D_s B \to D_s(A \otimes B)$ is the identity map on the underlying set.
- The map $w_A : D_0 A \to I$ maps all elements to the singleton.
- The map $c_{s,t,A} : D_{s+t} A \to D_s A \otimes D_t A$ is the diagonal map taking a to (a, a).
- The map $\epsilon_A : D_1 A \to A$ is the identity.
- The map $\delta_{s,t,A} : D_{s \cdot t} A \to D_s(D_t A)$ is the identity.

These maps are all non-expansive and it can be shown that they satisfy the diagrams [22] defining a $\mathcal{S}$-graded exponential comonad, but we will not need these abstract equalities for our purposes.

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. [22], 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_ϵ**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 [20, 31].

Lemma 4.4. *Let $q, r \in \mathcal{R}$. For any metric space A, the maps $(q \leq r)_A$, η_A, and $\mu_{q,r,A}$ are non-expansive maps and natural in A.*

Proof. Non-expansiveness and naturality for the subeffecting maps $(q \leq r)_A : T_q A \to T_r A$ and the unit maps $\eta_A : A \to T_0 A$ are straightforward. We describe the checks for the multiplication map $\mu_{q,r,A}$.

First, we check that the multiplication map has the claimed domain and codomain. Note that $d_A(x,x') = d_{T_r A}((x,y),(x',y')) \le q$ because of the definition of T_q, and $d_A(x',y') \le r$ because of the definition of T_r, so via the triangle inequality we have $d_A(x,y') \le r+q$ as claimed.

Second, we check non-expansiveness. Let $((x,y),(x',y')$ and $((w,z),(w',z'))$ be two elements of $T_q(T_r A)$. Then:

$$\begin{aligned}
d_{T_{r+q}A}(\mu((x,y),(x',y')),\mu((w,z),(w',z'))) &= d_{T_{r+q}A}((x,y'),(w,z')) && \text{(def. } \mu) \\
&= d_A(x,w) && \text{(def. } d_{T_{r+q}A}) \\
&= d_{T_r A}((x,y),(w,z)) && \text{(def. } d_{T_r}A) \\
&= d_{T_q(T_r A)}(((x,y),(x',y')),((w,z),(w',z'))) \\
& && \text{(def. } d_{T_q(T_r A)})
\end{aligned}$$

Finally, we can check naturality. Let $f : A \to B$ be any non-expansive map. By unfolding definitions, it is straightforward to see that $\mu_{q,r,B} \circ T_q(T_r f) = T_{r+q} f \circ \mu_{q,r,A}$. □

Lemma 4.5. *The functors T_r and associated maps form a $\mathcal{R}$-graded monad on* **Met**.

Proof. Establishing this fact requires checking the following two diagrams:

$$\begin{tikzcd}
T_r A \arrow[r, "\eta_{T_r A}"] \arrow[d, "T_r \eta_A"'] \arrow[dr, equal] & T_0(T_r A) \arrow[d, "\mu_{0,r,A}"] \\
T_r(T_0 A) \arrow[r, "\mu_{r,0,A}"'] & T_r A
\end{tikzcd}
\qquad
\begin{tikzcd}
T_p(T_q(T_r A)) \arrow[r, "T_p \mu_{q,r,A}"] \arrow[d, "\mu_{p,q,T_r A}"'] & T_p(T_{q+r} A) \arrow[d, "\mu_{p,q+r,A}"] \\
T_{p+q}(T_r A) \arrow[r, "\mu_{p+q,r,A}"'] & T_{p+q+r} A
\end{tikzcd}$$

Both diagrams follow by unfolding definitions. □

Graded monad strengths. As proposed by Moggi [38], monads that model computational effects should be *strong monads*; roughly, they should behave well with respect to products.

Definition 4.6. Let $r \in \mathcal{R}$. We define non-expansive maps $st_{r,A,B} : A \otimes T_r B \to T_r(A \otimes B)$ via

$$st_{r,A,B}(a,(b,b')) \triangleq ((a,b),(a,b'))$$

Moreover, these maps are natural in A and B.

We can check non-expansiveness. For $(a,(b,b'))$ and $(c,(d,d'))$ in $A \otimes T_r B$, we have:

$$\begin{aligned}
d_{T_r(A\otimes B)}(st(a,(b,b')),st(c,(d,d'))) &= d_{T_r(A\otimes B)}(((a,b),(a,b')),((c,d),(c,d'))) && \text{(def. } st) \\
&= d_{A\otimes B}((a,b),(c,d)) && \text{(def. } d_{T_r(A\otimes B)}) \\
&= d_A(a,c) + d_B(b,d) && \text{(def. } d_{A\otimes B}) \\
&= d_A(a,c) + d_{T_r B}((b,b'),(d,d')) && \text{(def. } d_{T_r B}) \\
&= d_{A\otimes T_r B}((a,(b,b')),(c,(d,d'))) && \text{(def. } d_{A\otimes T_r B})
\end{aligned}$$

Furthermore, it is possible to show that these maps satisfy the commutative diagrams needed to make T_r a graded strong monad [20, 31], though we will not need this fact for our development. Finally, we have shown that the neighborhood monad is strong with respect to the tensor product $\otimes$; in fact, the neighborhood monad is also strong with respect to the Cartesian product $\times$.

Graded distributive law. Gaboardi et al. [22] showed that languages supporting graded coeffects and graded effects can be modeled with a graded comonad, a graded monad, and a graded distributive law. In our model, we have the following family of maps.

Lemma 4.7. *Let $s \in \mathcal{S}$ and $r \in \mathcal{R}$ be grades, and let A be a metric space. Then identity map on the carrier set $|A| \times |A|$ is a non-expansive map*

$$\lambda_{s,r,A} : D_s(T_r A) \to T_{s \cdot r}(D_s A)$$

Moreover, these maps are natural in A.

Proof. We first check the domain and codomain. Let $x, y \in A$ be such that (x, y) is in the domain $D_s(T_r A)$ of the map. Thus (x, y) must also be in $T_r A$, and satisfy $d_A(x, y) \le r$ by definition of T_r. To show that this element is also in the range, we need to show that $d_{D_s A}(x, y) \le s \cdot r$, but this holds by definition of D_s. We can also check that this map is non-expansive:

$$\begin{aligned} d_{T_{s\cdot r}(D_s A)}((x,y),(x',y')) &\triangleq d_{D_s A}(x,x') && \text{(def. } T_{s\cdot r}) \\ &\triangleq s \cdot d_A(x,x') && \text{(def. } D_s) \\ &\triangleq s \cdot d_{T_r A}((x,y),(x',y')) && \text{(def. } T_r) \\ &\triangleq d_{D_s(T_r A)}((x,y),(x',y')) && \text{(def. } D_s) \end{aligned}$$

Since $\lambda_{s,r,A}$ is the identity map on the underlying set $|A| \times |A|$, it is evidently natural in A. □

It is similarly straightforward to show that the maps $\lambda_{s,r,A}$ form a graded distributive law in the sense of Gaboardi et al. [22]: for $s \le s'$ and $r \le r'$ the identity map $T_{s\cdot r}(D_s A) \to T_{s'\cdot r'}(D_{s'} A)$ is also natural in A, and the four diagrams required for a graded distributive law all commute [22, Fig. 8], but since we do not rely on these properties we will omit these details.

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 4.8. 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 $[\![\Gamma]\!]$:

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

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 $\gamma(x) \in [\![\tau]\!]$. Formally, projections are defined via the weakening maps $(0 \le s)_A ; w_A : D_s A \to I$ and the unitors.

We begin with two simple lemmas about context addition $(\Gamma + \Delta)$ and context scaling $s \cdot \Gamma$.

Lemma 4.9. *Let Γ and Δ such that $\Gamma + \Delta$ is defined. Then there is a non-expansive map $c_{\Gamma,\Delta} : [\![\Gamma + \Delta]\!] \to [\![\Gamma]\!] \otimes [\![\Delta]\!]$ given by:*

$$c_{\Gamma,\Delta}(\gamma) \triangleq (\gamma_\Gamma, \gamma_\Delta)$$

where γ_Γ and γ_Δ project out the positions in $dom(\Gamma)$ and $dom(\Delta)$, respectively.

Lemma 4.10. *Let Γ be a context and $s \in \mathcal{S}$ be a sensitivity. Then the identity function is a non-expansive map from $[\![s \cdot \Gamma]\!] \to D_s[\![\Gamma]\!]$.*

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

Definition 4.11. Fix $\rho : R \to R$ to be a (set) function such that for every $r \in R$ we have $d_R(r, \rho(r)) \le \epsilon$, and for every operation $\{\textbf{op} : \sigma \multimap \tau\} \in \Sigma$ in the signature fix an interpretation $[\![\textbf{op}]\!] : [\![\sigma]\!] \to [\![\tau]\!]$ such that for every closed value $\cdot \vdash v : \sigma$, we have $[\![\textbf{op}]\!]([\![v]\!]) = [\![op(v)]\!]$.
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.

We detail the cases here. We write our maps in diagrammatic order. To reduce notation, we sometimes omit indices on maps and we elide the bookkeeping morphisms from the SMCC structure (the unitors $\lambda_A : I \otimes A \to A$ and $\rho_A : A \otimes I \to I$; the associators $\alpha_{A,B,C} : (A \otimes B) \otimes C \to A \otimes (B \otimes C)$, and the symmetries $\sigma_{A,B} : A \otimes B \to B \otimes A$).

Const. Define $[\![\Gamma \vdash k : \textbf{num}]\!] : [\![\Gamma]\!] \to [\![\textbf{num}]\!]$ to be the constant function returning $k \in R$.
Ret. Let $f = [\![\Gamma \vdash v : \tau]\!]$. Define $[\![\Gamma \vdash \textbf{ret}\, v : M_0 \tau]\!]$ to be $f; \eta_{[\![\tau]\!]}$.
Subsumption. Let $f = [\![\Gamma \vdash e : M_r \tau]\!]$. Define $[\![\Gamma \vdash e : M_{r'} \tau]\!]$ to be $f; (r \le r')_{[\![\tau]\!]}$.
Round. Letting $f = [\![\Gamma \vdash k : \textbf{num}]\!]$, we can define

$$[\![\Gamma \vdash \textbf{rnd}\, k : M_\epsilon \textbf{num}]\!] \triangleq f; \langle id, \rho \rangle$$

Explicitly, the second map takes $r \in R$ to the pair $(r, \rho(r))$. The output is in $[\![M_\epsilon \textbf{num}]\!]$ by our assumption of the rounding function ρ, and the function is non-expansive by the definition of the metric on $[\![M_\epsilon \textbf{num}]\!]$.

Let-Bind. Let $f = [\![\Gamma \vdash e : M_r \sigma]\!]$ and $g = [\![\Delta, x :_s \sigma \vdash e' : M_q \tau]\!]$. We need to combine these ingredients to define a morphism from $[\![s \cdot \Gamma + \Delta]\!]$ to $[\![M_{s \cdot r+q} \tau]\!]$. First, we can apply the comonad to f and then compose with the distributive law to get:

$$D_s f; \lambda_{s,r,[\![\sigma]\!]} : D_s[\![\Gamma]\!] \to T_{s \cdot r} D_s [\![\sigma]\!].$$

Repeatedly pre-composing with the map $m_{s,A,B} : D_s A \otimes D_s B \to D_s(A \otimes B)$, we get a map from $[\![s \cdot \Gamma]\!] \to T_{s \cdot r} D_s [\![\sigma]\!]$. Composing in parallel with $id_{[\![\Theta]\!]}$ and post-composing with the strength, we have:

$$((m; D_s f; \lambda_{s,r,\sigma}) \otimes id_{[\![\Theta]\!]}); \otimes st_{s \cdot r, [\![\Theta]\!], [\![\sigma]\!]} : [\![\Theta]\!] \otimes [\![s \cdot \Gamma]\!] \to T_{s \cdot r}([\![\Theta]\!] \otimes D_s [\![\sigma]\!])$$

Next, applying the functor $T_{s \cdot r}$ to g and then post-composing with the multiplication $\mu_{s \cdot r, q, [\![\tau]\!]}$, we have:

$$T_{s \cdot r} g; \mu_{s \cdot r, q, [\![\sigma]\!]} : T_{s \cdot r}([\![\Theta]\!] \otimes D_s [\![\sigma]\!]) \to T_{s \cdot r+q} [\![\tau]\!],$$

which can be composed with the previous map to get a map from $[\![\Theta]\!] \otimes [\![s \cdot \Gamma]\!]$ to $T_{s \cdot r+q}[\![\tau]\!] = [\![M_{s \cdot r+q} \tau]\!]$. Pre-composing with $c_{s \cdot [\![\Gamma]\!], [\![\Theta]\!]}$ and a swap gives the desired map from $[\![s \cdot \Gamma + \Theta]\!]$ to $[\![M_{s \cdot r+q} \tau]\!]$.

Op. By assumption, we have an interpretation $[\![op]\!]$ for every operation in the signature Σ. We interpret $[\![\Gamma \vdash \textbf{op}(v) : \tau]\!]$ as the composition $[\![\Gamma \vdash v : \sigma]\!]; [\![\textbf{op}]\!]$.

Var. We define $[\![\Gamma \vdash x : \tau]\!]$ to be the map that maps $[\![\Gamma]\!]$ to the x-th component $[\![\tau]\!]$. All other components are mapped to I and then removed with the unitor.

Abstraction. Let $f = [\![\Gamma, x :_1 \sigma \vdash e : \tau]\!] : [\![\Gamma]\!] \otimes D_1[\![\sigma]\!] \to [\![\tau]\!]$. Now, $D_1[\![\sigma]\!] = [\![\sigma]\!]$ in our model. Since **Met** is an SMCC, we have a map $\lambda(f) : [\![\Gamma]\!] \to ([\![\sigma]\!] \multimap [\![\tau]\!])$.

Application. Since **Met** is an SMCC, there is a morphism $ev : (A \multimap B) \otimes A \to B$. Let $f = [\![\Gamma \vdash v : \sigma \multimap \tau]\!]$ and $g = [\![\Theta \vdash w : \sigma]\!]$. Then we can define:

$$c_{[\![\Gamma]\!],[\![\Theta]\!]}; (f \otimes g); ev : [\![\Gamma + \Theta]\!] \to [\![\tau]\!]$$

Unit. We let $[\![\Gamma \vdash \langle\rangle : \mathbf{unit}]\!]$ be the map that sends all points in Γ to $\star \in [\![\mathbf{unit}]\!]$.

$\times$ **Intro.** Letting f and g be the denotations of the premises, take $\langle f, g\rangle : [\![\Gamma]\!] \to [\![\sigma \times \tau]\!]$.

$\times$ **Elim.** Let f be the denotation of the premise, and post-compose by the projection π_i to get a map $[\![\Gamma]\!] \to [\![\tau_i]\!]$.

$\otimes$ **Intro.** Let f and g be the denotations of the premises, and define:

$$c_{[\![\Gamma]\!],[\![\Theta]\!]}; (f \otimes g) : [\![\Gamma + \Theta]\!] \to [\![\sigma \otimes \tau]\!]$$

$\otimes$ **Elim.** Let $f = [\![\Gamma \vdash v : \sigma \otimes \tau]\!] : [\![\Gamma]\!] \to [\![\sigma]\!] \otimes [\![\tau]\!]$ and $g = [\![\Theta, x :_s \sigma, y :_s \tau \vdash e : \rho]\!] : [\![\Theta]\!] \otimes D_s[\![\sigma]\!] \otimes D_s[\![\tau]\!] \to [\![\rho]\!]$. Applying the functor D_s to f and pre-composing with m, we get

$$m; D_s f : [\![s \cdot \Gamma]\!] \to D_s([\![\sigma]\!] \otimes [\![\tau]\!])$$

Since the map m is the identity in our model, we can post-compose by its inverse to get:

$$m; D_s f; m^{-1} : [\![s \cdot \Gamma]\!] \to D_s([\![\sigma]\!]) \otimes D_s([\![\tau]\!])$$

Composing in parallel with $id_{[\![\Theta]\!]}$, we get:

$$id_{[\![\Theta]\!]} \otimes (m; D_s f; m^{-1}) : [\![\Theta]\!] \otimes [\![s \cdot \Gamma]\!] \to [\![\Theta]\!] \otimes D_s[\![\sigma]\!] \otimes D_s[\![\tau]\!]$$

Post-composing with g and pre-composing with $c_{[\![s \cdot \Gamma]\!],[\![\Theta]\!]}$ completes the definition.

$+$ **Intro L.** Let f be the denotation of the premise, and take $f; \iota_1$ where ι_2 is the first injection into the coproduct.

$+$ **Intro R.** Let f be the denotation of the premise, and take $f; \iota_2$ where ι_2 is the second injection into the coproduct.

$+$ **Elim.** We need a few facts about our model. First, there is an isomorphism $dist_{D,s} : D_s(A + B) \cong D_s A + D_s B$ when s is *strictly* greater than zero. Second, there is a map $dist_\times : A \times (B + C) \to A \times B + A \times C$, which pushes the first component into the disjoint union. This map is non-expansive, and in fact **Met** is a distributive category.
Let $f = [\![\Gamma \vdash v : \sigma + \tau]\!]$ and $g_i = [\![\Theta, x_i :_s \sigma \vdash e_i : \rho]\!]$ for $i = 1, 2$. Since s is strictly greater than zero, $dist_{D,s}$ is an isomorphism. By using the functor D_s on f, composing in parallel with $id_{[\![\Theta]\!]}$ and distributing, we have:

$$(id_{[\![\Theta]\!]} \otimes (m; D_s f; dist_{D,s})); dist_\times : [\![\Theta]\!] \otimes [\![(s]\!] \to [\![\Theta]\!] \otimes D_s[\![\sigma]\!] + [\![\Theta]\!] \otimes D_s[\![\tau]\!]$$

By post-composing with the pairing map $[g_1, g_2]$ from the coproduct, and pre-composing with $c_{[\![s \cdot \Gamma]\!],[\![\Theta]\!]}$ and a swap, we get a map $[\![s \cdot \Gamma + \Theta]\!] \to [\![\rho]\!]$ as desired.

$!_s$ **Intro.** Let f be the denotation of the premise, and take $m; D_s f$.

$!_s$ **Elim.** Let $f = [\![\Gamma \vdash v : !_s \sigma]\!]$ and $g = [\![\Theta, x :_{m \cdot n} \sigma \vdash e : \tau]\!]$. We have:

$$m; D_m f; \delta^{-1}_{m,n,[\![\sigma]\!]} : [\![m \cdot \Gamma]\!] \to D_{m \cdot n}[\![\sigma]\!]$$

Here we use that $\delta_{m,n,[\![\sigma]\!]}$ is an isomorphism in our model. By composing in parallel with $id_{[\![\Theta]\!]}$, we can then post-compose by g. Pre-composing with $c_{[\![s \cdot \Gamma]\!],[\![\Theta]\!]}$ gives a map $[\![s \cdot \Gamma + \Theta]\!] \to [\![\tau]\!]$, as desired.

Let. This case is essentially the same as the other elimination cases.

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.

We first need a weakening lemma.

Lemma 4.12 (Weakening). *Let $\Gamma, \Gamma' \vdash e : \tau$ be a well-typed term. Then for any context, there is a derivation of $\Gamma, \Delta, \Gamma' \vdash e : \tau$ with semantics $[\![\Gamma, \Gamma' \vdash e : \tau]\!] \circ \pi$, where $\pi : [\![\Gamma, \Delta, \Gamma']\!] \to [\![\Gamma, \Gamma']\!]$ projects the components in Γ and Γ'.*

Proof. By induction on the typing derivation of $\Gamma, \Gamma' \vdash e : \tau$. □

Next, we show that the subsumption rule is admissible.

Lemma 4.13 (Subsumption). *Let $\Gamma \vdash e : M_r \tau$ be a well-typed program of monadic type, where the typing derivation concludes with the subsumption rule. Then either e is of the form $\textbf{ret}\, v$ or $\textbf{rnd}\, k$, or there is a derivation of $\Gamma \vdash e : M_r \tau$ with the same semantics that does not conclude with the subsumption rule.*

Proof. By straightforward induction on the typing derivation, using the fact that subsumption is transitive, and the semantics of the subsumption rule leaves the semantics of the premise unchanged since the subsumption map $(r \le s)_A$ is the identity function. □

Lemma 4.14 (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 $[\![\Gamma, \Gamma' \vdash e[\vec{v}/dom(\Delta)] : \tau]\!] = (id_{[\![\Gamma]\!]} \otimes [\![\cdot \vdash \vec{v} : \Delta]\!] \otimes id_{[\![\Gamma']\!]}); [\![\Gamma, \Delta, \Gamma' \vdash e : \tau]\!]$.

Proof. By induction on the typing derivation of $\Gamma, \Delta, \Gamma' \vdash e : \tau$. The base cases Unit and Const are obvious. The other base case Var follows by unfolding the definition of the semantics. Most of the rest of the cases follow from the substitution lemma for Fuzz [5, Lemma 3.3]. We show the cases for Round, Return, and Let-Bind, which differ from Fuzz. We omit the bookkeeping morphisms.

Case Round. Given a derivation $f = [\![\Gamma, \Delta, \Gamma' \vdash w : \textbf{num}]\!]$, by induction, there is a derivation $[\![\Gamma, \Gamma' \vdash w[\vec{v}/\Delta] : M_\epsilon \textbf{num}]\!] = (id_{[\![\Gamma]\!]} \otimes [\![\cdot \vdash \vec{v} : \Delta]\!] \otimes id_{[\![\Gamma']\!]}); f$. By applying rule Round and by definition of the semantics of this rule, we have a derivation

$$[\![\Gamma, \Gamma' \vdash (\textbf{rnd}\ w)[\vec{v}/\Delta] : M_\epsilon \textbf{num}]\!] = (id_{[\![\Gamma]\!]} \otimes [\![\cdot \vdash \vec{v} : \Delta]\!] \otimes id_{[\![\Gamma']\!]}); f; \langle id, \rho \rangle.$$

We are done since $[\![\Gamma, \Delta, \Gamma' \vdash \textbf{rnd}\ \ w : M_\epsilon \textbf{num}]\!] = f; \langle id, \rho \rangle$.

Case Return. Same as previous, using the unit of the monad $\eta_{[\![\tau]\!]}$ in place of $\langle id, \rho \rangle$.

Case Let-Bind. Suppose that $\Gamma = \Gamma_1, \Delta_1, \Gamma_2$ and $\Theta = \Theta_1, \Delta_2, \Theta_2$ such that $\Delta = s \cdot \Delta_1 + \Delta_2$. By combining Lemma 4.9 and Lemma 4.10, there is a natural transformation $\sigma : [\![s \cdot \Gamma + \Theta]\!] \to [\![\Theta]\!] \otimes D_s [\![\Gamma]\!]$.

Let $g_1 = [\![\Gamma_1, \Delta_1, \Gamma_2 \vdash w : M_r \sigma]\!]$ and $g_2 = [\![\Theta_1 1, \Delta_2, \Theta_2, x :_s \sigma \vdash f : M_{r'} \tau]\!]$. By induction, we have:

$$\tilde{g_1} = [\![\Gamma_1, \Gamma_2 \vdash w[\vec{v}/\Delta] : M_r \sigma]\!] = (id_{[\![\Gamma_1]\!]} \otimes [\![\cdot \vdash \vec{v} : \Delta]\!] \otimes id_{[\![\Gamma_2]\!]}); g_1$$
$$\tilde{g_2} = [\![\Theta_1, \Theta_2, x :_s \sigma \vdash f[\vec{v}/\Delta] : M_{r'} \tau]\!] = (id_{[\![\Theta_1]\!]} \otimes [\![\cdot \vdash \vec{v} : \Delta]\!] \otimes id_{[\![\Theta_2, x :_s \sigma]\!]}); g_2$$

Thus we have a derivation of the judgment $s \cdot (\Gamma_1, \Gamma_2) + (\Theta_1, \Theta_2) \vdash \textbf{let-bind}(w, x.f)[\vec{v}/\Delta] : M_{s \cdot r + r'} \tau$, and by the definition of the semantics of Let-Bind, its semantics is:

$$split; (id_{[\![\Theta_1, \Theta_2]\!]} \otimes (D_s \tilde{g_1}; \lambda_{s,r,[\![\sigma]\!]})); st_{[\![\Theta_1, \Theta_2]\!], [\![\sigma]\!]}; T_{s \cdot r} \tilde{g_2}; \mu_{s \cdot r, r', [\![\tau]\!]}$$

From here, we can conclude by showing that the first morphisms in $\tilde{g}_1$ and $\tilde{g}_2$ can be pulled out to the front. For instance,

$$D_s\tilde{g}_1 = (id_{[\![s\cdot\Gamma_1]\!]} \otimes D_s[\![\cdot \vdash \vec{v} : \Delta]\!] \otimes id_{[\![s\cdot\Gamma_2]\!]}); D_s g_1$$

by functoriality. By naturality of *split*, the first morphism can be pulled out in front of *split*. Similarly, for $\tilde{g}_2$, we have:

$$\begin{aligned} &st_{[\![\Theta_1,\Theta_2]\!],[\![\sigma]\!]}; T_{s\cdot r}(id_{[\![\Theta_1]\!]} \otimes [\![\cdot \vdash \vec{v} : \Delta]\!] \otimes id_{[\![\Theta_2, x :_s \sigma]\!]}) \\ &\qquad = (id_{[\![\Theta_1]\!]} \otimes [\![\cdot \vdash \vec{v} : \Delta]\!] \otimes id_{[\![\Theta_2]\!]} \otimes id_{T_{s\cdot r}D_s[\![\sigma]\!]}); st_{[\![\Theta_1,\Delta_2,\Theta_2]\!],[\![\sigma]\!]} \end{aligned}$$

by naturality of strength. By naturality, we can pull the first morphism out in front of *split*.

□

Lemma 4.15 (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: $[\![\vdash e : \tau]\!] = [\![\vdash e' : \tau]\!]$.*

Proof. By case analysis on the step rule, using the fact that e is well-typed. For the beta-reduction steps for programs of non-monadic type, preservation follows by the soundness theorem; these cases are exactly the same as in Fuzz [5].

The two step rules for programs of monadic type are new. It is possible to show soundness by appealing to properties of the graded monad T_r, but we can also show soundness more concretely by unfolding definitions and considering the underlying maps.

Let-Bind q. Suppose that $e = \textbf{let-bind}(\textbf{ret}\ v, x.f)$ is a well-typed program with type $M_{s\cdot r+q}\tau$. Since subsumption is admissible (Lemma 4.13), we may assume that the last rule is Let-Bind and we have derivations $\cdot \vdash \textbf{ret}\ v : M_r\sigma$ and $x :_s \sigma \vdash f : M_q\tau$. By definition, the semantics of $\cdot \vdash e : M_{s\cdot r+q}\tau$ is given by the composition:

$$I \longrightarrow D_sI \xrightarrow{D_s(v;\eta_\sigma;(0\leq r)_\sigma)} D_sT_r\sigma \xrightarrow{\lambda_{s,r,\sigma}} T_{s\cdot r}D_s\sigma \xrightarrow{T_{s\cdot r}f} T_{s\cdot r}T_q\tau \xrightarrow{\mu_{s\cdot r,q,\tau}} T_{s\cdot r+q}\tau$$

By substitution (Lemma 4.14), we have a derivation of $\cdot \vdash f[v/x] : M_q\tau$. By applying the subsumption rule, we have a derivation of $\cdot \vdash f[v/x] : M_{s\cdot r+q}\tau$ with semantics:

$$I \longrightarrow D_sI \xrightarrow{D_sv} D_s\sigma \xrightarrow{f} T_q\tau \xrightarrow{\lambda_{q,s\cdot r+q,\tau}} T_{s\cdot r+q}\tau$$

Noting that the underlying maps of s and μ are the identity function, both compositions have the same underlying maps, and hence are equal morphisms.

Let-Bind Assoc. Suppose that $e = \textbf{let-bind}(\textbf{let-bind}(\textbf{rnd}\ k, x.f), y.g)$ is a well-typed program with type $M_{s\cdot r+q}\tau$. Since subsumption is admissible (Lemma 4.13), we have derivations:

$$\vdash \textbf{rnd}\ k : M_{r_1}\textbf{num} \qquad x :_t \textbf{num} \vdash f : M_{r_2}\sigma \qquad y :_s \sigma \vdash g : M_q\tau$$

such that $t \cdot r_1 + r_2 = r$. By applying Let-Bind on the latter two derivations, we have:

$$x :_{s\cdot t} \textbf{num} \vdash \textbf{let-bind}(f, y.g) : M_{s\cdot r_2+q}\tau$$

And by applying Let-Bind again, we have:

$$\vdash \textbf{let-bind}(\textbf{rnd}\ k, x.\textbf{let-bind}(f, y.g)) : M_{s\cdot t\cdot r_1+s\cdot r_2+q}\tau$$

This type is precisely $M_{s\cdot r+q}\tau$. The semantics of e and e' have the same underlying maps, and hence are equal morphisms. □

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

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

Note that $\textbf{let-bind}(\textbf{rnd}\ k, x.f)$ is no longer a value under these semantics, since $\textbf{rnd}\ k$ 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 4.17. 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:

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

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

Following the same approach as in Lemma 4.15, it is straightforward to prove that these denotational semantics are sound for their respective operational semantics.

Lemma 4.18 (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.*

Proof. By case analysis on the step relation. We detail the cases where $e = \textbf{rnd}\ k$.

Case: $\textbf{rnd}\ k \mapsto_{id} \textbf{ret}\ k$. Suppose that $\cdot \vdash \textbf{rnd}\ k : M_u\textbf{num}$, where $\epsilon \le u$. Then there is a derivation of $\cdot \vdash \textbf{ret}\ k : M_u\textbf{num}$ by Return and Subsumption, and

$$(\!|\cdot \vdash \textbf{rnd}\ k : M_u\textbf{num}|\!)_{id} = (\!|\cdot \vdash k : \textbf{num}|\!)_{id} = (\!|\cdot \vdash \textbf{ret}\ k : M_u\textbf{num}|\!)_{id}.$$

Case: $\mathbf{rnd}\, k \mapsto_{fp} \mathbf{ret}\, \rho(k)$. Suppose that $\cdot \vdash \mathbf{rnd}\, k : M_u \mathbf{num}$, where $\epsilon \leq u$. Then there is a derivation of $\cdot \vdash \mathbf{ret}\, k : M_u \mathbf{num}$ by Return and Subsumption, and

$$\llparenthesis \cdot \vdash \mathbf{rnd}\, k : M_u \mathbf{num} \rrparenthesis_{fp} = \llparenthesis \cdot \vdash \rho(k) : \mathbf{num} \rrparenthesis_{fp} = \llparenthesis \cdot \vdash \mathbf{ret}\, \rho(k) : M_u \mathbf{num} \rrparenthesis_{fp}.$$

□

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 4.19 (Pairing). *Let $\cdot \vdash e : M_r \mathbf{num}$. Then we have: $U\llbracket e \rrbracket = \langle \llparenthesis e \rrparenthesis_{id}, \llparenthesis e \rrparenthesis_{fp} \rangle$ in* **Set**: *the first projection of $U\llbracket e \rrbracket$ is $\llparenthesis e \rrparenthesis_{id}$, and the second projection is $\llparenthesis e \rrparenthesis_{fp}$.*

Proof. By the logical relation for termination, the judgment $\cdot \vdash e : M_r \mathbf{num}$ implies that e is in $\mathcal{R}^n_{M_r \mathbf{num}}$ for some $n \in \mathbb{N}$. We proceed by induction on n.

For the base case $n = 0$, we know that e reduces to either $\mathbf{ret}\, v$ or $\mathbf{rnd}\, v$. We can conclude since by inversion v must be a real constant, and $U\llbracket v \rrbracket = \llparenthesis v \rrparenthesis_{id} = \llparenthesis v \rrparenthesis_{fp}$.

For the inductive case $n = m + 1$, we know that e reduces to $\mathbf{let\text{-}bind}(\mathbf{rnd}\, k, x.f)$. By the logical relation, we know that for all $\cdot \vdash v : \mathbf{num}$, we have $f[v/x] \in R^m_{M_r \mathbf{num}}$ and so by induction:

$$\llparenthesis f[k/x] \rrparenthesis_{id} \triangleq \llparenthesis \mathbf{let\text{-}bind}(\mathbf{rnd}\, k, x.f) \rrparenthesis_{id} = U\llbracket f[k/x] \rrbracket ; \pi_1$$

$$\llparenthesis f[\rho(k)/x] \rrparenthesis_{fp} \triangleq \llparenthesis \mathbf{let\text{-}bind}(\mathbf{rnd}\, k, x.f) \rrparenthesis_{fp} = U\llbracket f[\rho(k)/x] \rrbracket ; \pi_2$$

Thus we just need to show:

$$U\llbracket \mathbf{let\text{-}bind}(\mathbf{rnd}\, k, x.f) \rrbracket = \langle U\llbracket f[k/x] \rrbracket ; \pi_1, U\llbracket f[\rho(k)/x] \rrbracket ; \pi_2 \rangle$$

where we have judgments $\cdot \vdash \mathbf{rnd}\, k : M_r \mathbf{num}$ and $x :_s \mathbf{num} \vdash f : M_q \mathbf{num}$. Unfolding the definition, $\llbracket \mathbf{let\text{-}bind}(\mathbf{rnd}\, k, x.f) \rrbracket$ is equal to the composition:

$$I \xrightarrow{m_{s,I}} D_s I \xrightarrow{D_s \llbracket \mathbf{rnd}\, k \rrbracket} D_s T_r \llbracket \mathbf{num} \rrbracket \xrightarrow{\lambda_{s,r,\llbracket \mathbf{num} \rrbracket}} T_{r \cdot s} D_s \llbracket \mathbf{num} \rrbracket \xrightarrow{T_{r \cdot s} \llbracket f \rrbracket} T_{r \cdot s} T_q \llbracket \mathbf{num} \rrbracket \xrightarrow{\mu_{r \cdot s, q, \llbracket \mathbf{num} \rrbracket}} T_{r \cdot s + q} \llbracket \mathbf{num} \rrbracket$$

We conclude by applying the substitution lemma and considering the underlying maps. □

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

Corollary 4.20 (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 $d_{\llbracket \mathbf{num} \rrbracket}(\llbracket v_{id} \rrbracket, \llbracket v_{fp} \rrbracket) \leq r$.*

Proof. Under the ideal and floating point semantics, the only values of monadic type are of the form $\mathbf{ret}\, v$. Since these operational semantics are type-preserving and normalizing, we must have $e \mapsto^*_{id} \mathbf{ret}\, v_{id}$ and $e \mapsto^*_{fp} \mathbf{ret}\, v_{fp}$. By soundness (Lemma 4.18), $\llparenthesis e \rrparenthesis_{id} = \llparenthesis \mathbf{ret}\, v_{id} \rrparenthesis_{id}$ and $\llparenthesis e \rrparenthesis_{fp} = \llparenthesis \mathbf{ret}\, v_{fp} \rrparenthesis_{fp}$. By pairing (Lemma 4.19), we have $U\llbracket e \rrbracket = \langle \llparenthesis \mathbf{ret}\, v \rrparenthesis_{id}, \llparenthesis \mathbf{ret}\, v \rrparenthesis_{fp} \rangle$. Since the forgetful functor is the identity on morphisms, we have $\llbracket e \rrbracket = \langle \llparenthesis \mathbf{ret}\, v_{id} \rrparenthesis_{id}, \llparenthesis \mathbf{ret}\, v_{fp} \rrparenthesis_{fp} \rangle : I \to T_r \llbracket \mathbf{num} \rrbracket$ in **Met**. Now by definition $\llparenthesis \mathbf{ret}\, v_{id} \rrparenthesis_{id} = \llparenthesis v_{id} \rrparenthesis_{id}$ and $\llparenthesis \mathbf{ret}\, v_{fp} \rrparenthesis_{fp} = \llparenthesis v_{fp} \rrparenthesis_{fp}$, hence we can conclude by the definition of the monad T_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 $\llbracket \mathbf{num} \rrbracket$ 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 $\mathbf{num}$ as the set of strictly positive real numbers $\mathbb{R}^{>0}$ with the relative precision (RP) metric (Equation (4)), 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. 5. 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. 6. Type signatures of (defined) operations that perform rounding in $\Lambda_{\mathbf{num}}$.

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

```
function addfp (xy: <num, num>)
  : M[eps]num {
  rnd(add(|(pi1 xy),(pi2 xy)|))
}
```

Fig. 7. Example defined operations that perform rounding in $\Lambda_{\mathbf{num}}$. We denote the unit roundoff by `eps`.

error analysis as described by Olver [39]. Using this metric, we extend the language with the four primitive arithmetic operations shown in Figure 5.

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 5 are non-expansive functions [39, Corollary 1 & Property V]; recall that the two product types have different metrics (Section 4.3).

Using the primitive operations in Figure 5 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. We show the type signature of these functions in Figure 6, and provide $\Lambda_{\mathbf{num}}$ implementations of a multiplication and addition that perform rounding in Figure 7.

We present the examples in this section in the actual syntax used by the implementation of $\Lambda_{\mathbf{num}}$, which we introduce in Section 6, and which 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. (6) we have that $RP(x, \rho(x)) \leq$ `eps` where `eps` is the unit roundoff. Error soundness (Corollary 4.20) 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.

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

MA : num⊸ num⊸ num⊸ M[2*eps]num
```

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

FMA : num⊸ num⊸ num⊸ M[eps]num
```

Fig. 8. Multiply add and fused-multiply add in $\Lambda_{\mathbf{num}}$.

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 given in Figure 8. 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 8. 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 is rewrites a nth-degree polynomial $p(x) = a_0 + a_1 x + \cdots a_n x^n$ as

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

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 degree 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 9 and shows that the rounding error on exact inputs is guaranteed to be bounded by `2*eps`:

$$RP(\llparenthesis \mathbf{Horner2}\ as\ x \rrparenthesis_{id}, \llparenthesis \mathbf{Horner2}\ as\ x \rrparenthesis_{fp}) \leq 2 * \mathtt{eps}. \tag{10}$$

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

$$RP(p(\vec{a}, x), p(\vec{a}', x')) \leq \sum_{i=0}^{2} RP(a_i, a_i') + 2 \cdot RP(x, x') \leq 3r + 2q. \tag{11}$$

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 9. From the type, we see that the total roundoff error of `Horner2_with_error` is

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

Fig. 9. Horner's scheme for evaluating a second order polynomial in $\Lambda_{\mathbf{num}}$, with (`Horner2_with_error`) and without (`Horner2`) input error.

`7*eps`: from eq. (11) 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`.

Remark. *$\Lambda_{\mathbf{num}}$ is a higher-order language, and it is possible to implement Horner's scheme for polynomials of fixed degree n by first writing a n-ary monadic fold function—a higher-order function—and then applying it to a product of n coefficients a_i. The type system of $\Lambda_{\mathbf{num}}$ is capable of expressing the fold function along with its roundoff error.*

5.1 Floating-Point Conditionals

In the presence of rounding error, conditional branches present a particular challenge: while the ideal execution may follow one branch, the floating-point execution may follow another. In $\Lambda_{\mathbf{num}}$, we can perform rounding error analysis on programs with conditional expressions (case analysis) when executions *take the same branch*, for instance, when the data in the conditional is a boolean expression that does not have floating-point error because it is some kind of parameter to the system, or some exactly-represented value that is computed only from other exactly-represented values. This is a restriction of the Fuzz-style type system of $\Lambda_{\mathbf{num}}$, which is not able to compare the difference between two different branches since the main metatheoretic guarantee only serves as a sensitivity analysis describing how a single program behaves on two different inputs. In $\Lambda_{\mathbf{num}}$, the rounding error of a program with a case analysis is then a measure of the maximum rounding error that occurs in any single branch.

As an example of performing rounding error analysis in $\Lambda_{\mathbf{num}}$ on functions with conditionals, we first add the primitive operation **is_pos** $:!_\infty\mathbb{R} \multimap \mathbb{B}$, which tests if a real number is greater than zero. The sensitivity on the argument to **is_pos** is necessarily infinity, since an arbitrarily small change in the argument to could lead to an infinitely large change in the boolean output. Using **is_pos** we define the function **case1**, which computes the square of a negative number, or returns the value 0 (lifted to monadic type):

$$\mathbf{case1} : !_\infty\mathbb{R} \multimap M_u\mathbb{R}$$

$$\begin{aligned}\mathbf{case1}\ x = \mathbf{let}\ \ [c]\ =\ \ &\mathbf{is\_pos}\ x\ \mathbf{in}\\ &\mathbf{if}\ c\ \mathbf{then}\ \mathbf{mul}_{fp}(x,x)\ \mathbf{else}\ \mathbf{ret}\ 0.\end{aligned}$$

From the signature of **case1**, we see that the relative distance (RD) is unit roundoff, due to the single rounding in $\mathbf{mul}_{fp}(x,x)$.

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 [21], 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 $\Gamma^\bullet = \overline{\Gamma}$.

$$\dfrac{}{\Gamma^\bullet, x : \sigma; x \Rightarrow \Gamma^0, x :_1 \sigma; \sigma}\ (\text{Var}) \qquad \dfrac{\begin{array}{c}\Gamma^\bullet; v \Rightarrow \Delta; \sigma \otimes \tau \\ \Gamma^\bullet, x : \sigma, y : \tau; e \Rightarrow \Gamma, x :_{s_1} \sigma, y :_{s_2} \tau; \rho\end{array}}{\Gamma^\bullet; \textbf{let}\ (x, y) = v\ \textbf{in}\ e \Rightarrow \Gamma + \textbf{max}(s_1, s_2) * \Delta; \rho}\ (\otimes\ \text{E})$$

$$\dfrac{\Gamma^\bullet; v \Rightarrow \Gamma; \tau_1 \times \tau_2}{\Gamma^\bullet; \pi_i\ v \Rightarrow \Gamma; \tau_i}\ (\times\ \text{E}) \quad \dfrac{\begin{array}{c}\Gamma^\bullet, x : \sigma; e \Rightarrow \Gamma, x :_s \sigma; \tau \\ s \geq 1\end{array}}{\Gamma^\bullet; \lambda(x : \sigma).e \Rightarrow \Gamma; \sigma \multimap \tau}\ (\multimap \text{I}) \quad \dfrac{\begin{array}{c}\Gamma^\bullet; v \Rightarrow \Gamma; \sigma \multimap \tau \\ \Gamma^\bullet; w \Rightarrow \Delta; \sigma' \\ \sigma' \sqsubseteq \sigma\end{array}}{\Gamma^\bullet; vw \Rightarrow \Gamma + \Delta; \tau}\ (\multimap \text{E})$$

$$\dfrac{\begin{array}{c}\Gamma^\bullet; e \Rightarrow \Gamma; \tau \\ \Gamma^\bullet, x; f \Rightarrow \Theta, x :_s \tau; \sigma \qquad s > 0\end{array}}{\Gamma^\bullet; \textbf{let}\ x = e\ \textbf{in}\ f \Rightarrow s * \Gamma + \Theta; \sigma}\ (\text{Let}) \qquad \dfrac{\Gamma^\bullet; v \Rightarrow \Gamma_1; \sigma \qquad \Gamma^\bullet; w \Rightarrow \Gamma_2; \tau}{\Gamma^\bullet; \langle v, w \rangle \Rightarrow \textbf{max}(\Gamma_1, \Gamma_2); \sigma \times \tau}\ (\times\ \text{I})$$

$$\dfrac{\Gamma^\bullet; v \Rightarrow \Gamma; \sigma}{\Gamma^\bullet; \textbf{inl}\ \ v \Rightarrow \Gamma; \sigma + \tau}\ (+\ \text{I}) \quad \dfrac{\begin{array}{c}\Gamma^\bullet; v \Rightarrow \Gamma; !_s\sigma \\ \Gamma^\bullet; x \Rightarrow \Theta, x :_{t*s} \sigma; e : \tau\end{array}}{\Gamma^\bullet; \textbf{let}\ [x] = v\ \textbf{in}\ e \Rightarrow t * \Gamma + \Theta; \tau}\ (!\ \text{E}) \quad \dfrac{\begin{array}{c}\Gamma^\bullet; v \Rightarrow \Gamma; \sigma \\ \{\textbf{op} : \sigma \multimap \textbf{num}\} \in \Sigma\end{array}}{\Gamma^\bullet; \textbf{op}(v) \Rightarrow \Gamma; \textbf{num}}\ (\text{Op})$$

$$\dfrac{\begin{array}{c}\Gamma^\bullet, x; e \Rightarrow \Theta, x :_s \sigma; \rho \qquad \Gamma^\bullet; v \Rightarrow \Gamma; \sigma + \tau \\ \Gamma^\bullet, y; f \Rightarrow \Theta, y :_s \tau; \rho \qquad s > 0\end{array}}{\Gamma^\bullet; \textbf{case}\ v\ \textbf{of}\ (\textbf{inl}\ x.e \mid \textbf{inr}\ y.f) \Rightarrow s * \Gamma + \Theta; \rho}\ (+\ \text{E}) \qquad \dfrac{\Gamma^\bullet; v \Rightarrow \Gamma; \sigma}{\Gamma^\bullet; [v\{s\}] \Rightarrow s * \Gamma; !_s\sigma}\ (!\ \text{I})$$

$$\dfrac{\begin{array}{c}\Gamma^\bullet; v \Rightarrow \Gamma_1; \sigma \\ \Gamma^\bullet; w \Rightarrow \Gamma_2; \tau\end{array}}{\Gamma; (v, w) \Rightarrow \Gamma_1 + \Gamma_2; \sigma \otimes \tau}\ (\otimes\ \text{I}) \quad \dfrac{\Gamma^\bullet; v \Rightarrow \Gamma; \tau}{\Gamma^\bullet; \textbf{ret}\ v \Rightarrow \Gamma; M_0\tau}\ (\text{Ret}) \quad \dfrac{\Gamma^\bullet; v \Rightarrow \Gamma; \textbf{num}}{\Gamma^\bullet; \textbf{rnd}\ v \Rightarrow \Gamma; M_q\textbf{num}}\ (\text{Rnd})$$

$$\dfrac{\begin{array}{c}\Gamma^\bullet; v \Rightarrow \Gamma; M_r\sigma \\ \Gamma^\bullet, x; f \Rightarrow \Theta, x :_s \sigma; M_q\tau\end{array}}{\Gamma^\bullet; \textbf{let-bind}(v, x.f) \Rightarrow s * \Gamma + \Theta; M_{s*r+q}\tau}\ (M_u\ \text{E}) \qquad \dfrac{}{\Gamma^\bullet; k \in R \Rightarrow \Gamma^0; \textbf{num}}\ (\text{Const})$$

Fig. 10. Algorithmic rules for $\Lambda_{\mathbf{num}}$. Γ^0 denotes an environment where all variables have sensitivity zero.

$$\frac{\sigma \sqsubseteq \sigma' \qquad \tau \sqsubseteq \tau'}{\sigma \times \tau \sqsubseteq \sigma' \times \tau'} \sqsubseteq .\times \qquad \frac{\sigma \sqsubseteq \sigma' \qquad \tau \sqsubseteq \tau'}{\sigma \otimes \tau \sqsubseteq \sigma' \otimes \tau'} \sqsubseteq .\otimes \qquad \frac{\sigma \sqsubseteq \sigma' \qquad \tau \sqsubseteq \tau'}{\sigma + \tau \sqsubseteq \sigma' + \tau'} \sqsubseteq .+$$

$$\frac{\sigma' \sqsubseteq \sigma \qquad \tau \sqsubseteq \tau'}{\sigma \multimap \tau \sqsubseteq \sigma' \multimap \tau'} \sqsubseteq .\multimap \qquad \frac{\sigma \sqsubseteq \sigma' \qquad u \leq u'}{M_u \sigma \sqsubseteq M_{u'} \sigma'} \sqsubseteq .M \qquad \frac{\sigma \sqsubseteq \sigma' \qquad s \leq s'}{!_s \sigma \sqsubseteq !_{s'} \sigma'} \sqsubseteq .!$$

Fig. 11. Subtyping relation for $\Lambda_{\mathbf{num}}$, with $s, s', u, u' \in \mathbb{R}^{\geq 0} \cup \{\infty\}$.

We solve the sensitivity inference problem using the algorithm given in Figure 10. 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 $\Gamma^\bullet; e \Rightarrow \Delta; \sigma$.

Following [4] the algorithm uses a *bottom-up* rather than a *top-down* approach. In the *top-down* approach, given a term e, type σ, and environment Γ, the environment is split and used recursively to type the subterms of the expression e. The *bottom-up* approach avoids splitting the environment Γ by calculating the minimal sensitivities and roundoff errors required to type each subexpression. The sensitivities and errors of each subexpression are then combined and compared to Γ and σ using subtyping. While the DFuzz type sytem supports a notion of subtyping capturing the fact that a k-sensitive function is also k'-sensitive for $k \leq k'$, in $\Lambda_{\mathbf{num}}$, subtyping is admissable:

Theorem 6.2. *Given a derivation $\Gamma \vdash e : \tau$ and a type τ' such that $\tau \sqsubseteq \tau'$ where $\sqsubseteq$ is the subtyping relation defined according to rules in Figure 11, then the derivation $\Gamma \vdash e : \tau'$ is derivable.*

Type checking in $\Lambda_{\mathbf{num}}$ enjoys several nice properties: the algorithm is sound, complete, and decidable. Unlike DFuzz, where subtyping is undecidable due to polynomial constraints over natural numbers, in $\Lambda_{\mathbf{num}}$ type checking requires only checking inequalities on real constants.

Theorem 6.3 (Algorithmic Soundness). *If $\Gamma^\bullet; e \Rightarrow \Gamma; \sigma$ then the derivation $\Gamma \vdash e : \sigma$ exists.*

Proof. By induction on the algorithmic derivations, we see that every step of the algorithm corresponds to a derivation in $\Lambda_{\mathbf{num}}$. Many cases are immediate, but those that utilize explicit subtyping or subenvironments are not; we detail those here.

Case ($\multimap$ E). By induction, we have $\Gamma \vdash v : \sigma \multimap \tau$ and $\Delta \vdash w : \sigma'$. We also know that $\sigma' \sqsubseteq \sigma$; the derivation $\Gamma \vdash w : \sigma$ follows from subtyping on the right.

Case ($\times$ I). Let us denote $\mathbf{max}(\Gamma_1, \Gamma_2)$ by the environment Δ. By induction and Definition 3.2, we have $\Delta \vdash v : \sigma$ and $\Delta \vdash w : \tau$.

Case ($\otimes$ E). Let us denote by Θ the environment $\Gamma + \mathbf{max}(s_1, s_2) * \Delta$ and by s_{max} the sensitivity $\mathbf{max}(s_1, s_2)$. By definition Definition 3.2, we have that $\Gamma, x :_{s_{max}} : \sigma, y :_{s_{max}} : \tau \sqsubseteq \Gamma, x :_{s_1} : \sigma, y :_{s_2} : \tau$ and by induction we have that $\Gamma, x :_{s_{max}} : \sigma, y :_{s_{max}} : \tau \vdash e : \rho$ and $\Delta \vdash v : \sigma \times \tau$.

Case (Let). By induction, we have that $\Gamma \vdash e : \tau$, and we also know that $\tau \sqsubseteq \tau'$. It suffices to show that $\Gamma \vdash e : \tau'$, which follows from definition Definition 3.2.

□

Theorem 6.4 (Algorithmic Completeness). *If $\Gamma \vdash e : \sigma$ is valid derivation in $\Lambda_{\mathbf{num}}$, then there exists a type σ' and an environment Γ' such that $\overline{\Gamma}; e \Rightarrow \Gamma'; \sigma'$ and $\Gamma \sqsubseteq \Gamma'$ and $\sigma' \sqsubseteq \sigma$.*

Proof. Derivations in $\Lambda_{\mathbf{num}}$ correspond to steps of the algorithm; completeness follows by induction on $\Lambda_{\mathbf{num}}$ derivations, and by unfolding the definitions of subenvironments (Definition 3.2) and subtyping (Figure 11). An interesting case to consider is (! I).

Case (! I).

$$\frac{\Gamma^\bullet; v \Rightarrow \Gamma; \sigma}{\Gamma^\bullet; [v\{s\}] \Rightarrow s * \Gamma; !_s \sigma} \text{ (! I)}$$

By induction, we have $\overline{\Gamma}; v \Rightarrow \Gamma'; \sigma'$ where $\Gamma \sqsubseteq \Gamma'$ and $\sigma' \sqsubseteq \sigma$. It follows that $\overline{\Gamma}; [vs] \Rightarrow s * \Gamma'; !_s\sigma'$. We are required to show that $s * \Gamma \sqsubseteq s * \Gamma'$ and $!_s\sigma' \sqsubseteq !_s\sigma$, which follow by definition.

□

Annotations for sensitivity inference. The sensitivity-inference algorithm relies on type or sensitivity annotations in two cases: a type must be provided for the argument of lambda terms ($\multimap$ I) and a numeric sensitivity must be provided in the introduction rule for graded exponential types (! I). The required annotations are highlighted in blue in Figure 10. With these annotations alone, we are able to perform sensitivity and type inference using the algorithmic rules defined in Figure 10.

6.2 Evaluation

We compare the performance of the type-checker for $\Lambda_{\mathbf{num}}$ to two popular static analysis tools that *soundly* and automatically bound numerical errors: FPTaylor [43] and Gappa [19]. The soundness of the error bounds inferred by our type-checker is guaranteed by Corollary 4.20 and the instantiation of $\Lambda_{\mathbf{num}}$ described in Section 5. In particular, we only consider benchmarks over the positive real numbers and we use the round towards $+\infty$ rounding mode. We use benchmarks from FPBench [14] and also include the Horner scheme for second and fifth order polynomials in our comparison.

The results of our evaluation are given in Table 3. It is important to note that both FPTaylor and Gappa require user provided interval bounds on the input variables in order to compute the relative error. For each of the benchmarks we evaluate, we use an interval of $[0.1, 1000]$. We used the default configuration for FPTaylor, and used Gappa without providing hints for interval subdivision. The precision of each benchmark is set at binary64; the unit roundoff in this setting is 2^{52} (approximately 2.22e-16). Only the `Horner2_with_error` benchmark assumes error in the inputs.

We draw three main conclusions from our evaluation.

Roundoff error analysis via type checking is fast. On all of the benchmarks, $\Lambda_{\mathbf{num}}$ was able to solve the sensitivity checking problem on the order of milliseconds, at least an order of magnitude faster than both Gappa and FPTaylor.

Roundoff error bounds derived via type checking are useful. While $\Lambda_{\mathbf{num}}$ does not always produce the tightest error bound on all benchmarks, this is to be expected: the RP metric is a small overapproximation (on the order of 2^{-104} for binary64 and round towards $+\infty$) to the relative error (see Section 2). However, *this small overapproximation is negligible in most cases.* On benchmarks where rounding errors are composed and magnified, such as in `Horner5`, `Horner2_with_error`, and `test02_sum8`, we see that our type-based approach performs particularly well.

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. This is exemplified by the benchmark `x_by_xy` in Table 3, 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 $[5.0\text{e-}05, 1.0]$ 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.

Table 3. 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	1	100	10
x_by_xy*	3	**4.44e-16**	fail	2.22e-12	2.0e-04	1	-	10
one_by_sqrtxx	3	**5.55e-16**	5.09e-13	3.33e-12	1.09e-03	1	30	10
sqrt_add*	5	8.88e-16	**6.66e-16**	5.93e+01	1.5	2	20	20
test02_sum8*	8	**1.56e-15**	4.66e-14	5.97e-12	3.35e-02	2	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	30	10
Horner2	4	**4.44e-16**	6.49e-11	9.02e+09	6.84e-06	2	9.7e3	10
Horner2_with_error	4	**1.55e-15**	1.61e-10	9.02e+09	9.64e-06	2	1.6e4	20
Horner5	10	**1.33e-15**	1.11e-01	9.02e+18	1.20e-14	2	1.9e4	20

7 EXTENDING THE NEIGHBORHOOD MONAD

So far, we have seen how the graded neighborhood monad can model error under different metrics from rounding operations. 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 our main example from Section 5, we interpret the numeric type **num** as the strictly positive real numbers $\mathbb{R}^{>0}$ with Olver's RP metric. The rounding operation ρ is required to satisfy $RP(r, \rho(r))$ for every positive real number r. In practice, rounding the result of a floating-point operation might result in a *non-normal* floating-point number: 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:
$$(T^*_r f)((x, y)) \triangleq \begin{cases} (f(x), f(y)) & : y \in A \\ (f(x), \diamond) & : y = \diamond \end{cases}$$

The associated natural transformations are defined as follows:

- 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 $\mu_{q,r,A} : T^*_q(T^*_r A) \to T^*_{q+r} A$ is defined via:

$$\begin{cases} \mu_{q,r,A}((x,y),(x',y')) & \triangleq (x, y') \\ \mu_{q,r,A}((x,y),\diamond) & \triangleq (x, \diamond) \end{cases}$$

T^*_r are evidently functors, and the associated maps are natural transformations. T^*_r is also a strong monad, and comes with a distributive law.

Definition 7.2. Let $r \in \mathcal{R}$. We define a family of non-expansive maps:

$$st^*_{r,A,B} : A \otimes T^*_r B \to T^*_r(A \otimes B)$$

$$st^*_{r,A,B}(a,(b,b')) \triangleq \begin{cases} ((a,b),(a,b')) & : b' \neq \diamond \\ ((a,b),\diamond) & : b' = \diamond \end{cases}$$

making T^*_r a strong monad. Furthermore, for any $s \in \mathcal{S}$ the identity map is a non-expansive map with the following type:

$$\lambda^*_{s,r,A} : D_s(T^*_r A) \to T^*_{s \cdot r}(D_s A)$$

Extending the **Met** *semantics.* We can define the exceptional metric semantics $[\![\Gamma \vdash e : \tau]\!]^* : [\![\Gamma]\!] \to [\![\tau]\!]$ just as before (Definition 4.11), 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); it would be straightforward to distinguish between different kinds of errors, but we stick to a single error for simplicity. 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 the rounding function to satisfy $d_R(r, \rho^*(r)) \leq \epsilon$ whenever $\rho^*(r)$ is not the exceptional 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:

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

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 4.16) 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: $\textbf{let-bind}(\textbf{err}, x.f) \mapsto_{fp} \textbf{err}$

Establishing error soundness. It is straightforward to show that these step rules are sound for the floating point semantics; the proof is the analogue of Lemma 4.18.

Lemma 7.3 (Preservation). *Let $\cdot \vdash e : \tau$ be a well-typed closed term, and suppose $e \mapsto_{fp} e'$. Then there is a derivation of $\cdot \vdash e' : \tau$ such that $(\!|\vdash e : \tau|\!)^*_{fp} = (\!|\vdash e' : \tau|\!)^*_{fp}$.*

Finally, we have the same paired soundness theorem as before:

Lemma 7.4 (Pairing). *Let $\cdot \vdash e : M_r\textbf{num}$. 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 $(\!|e|\!)^*_{fp}$.*

The proof is essentially identical to Lemma 4.19. Finally, we have the following analogue of Corollary 4.20.

Corollary 7.5. *Let $\cdot \vdash e : M_r\textbf{num}$ be well-typed. Under the exceptional semantics, either: $e \mapsto_{id}$ $\textbf{ret } v_{id}$ and $e \mapsto_{fp} \textbf{ret } v_{fp}$, and $d_{[\![\textbf{num}]\!]}([\![v_{id}]\!]^*, [\![v^*_{fp}]\!]) \leq r$, or $e \mapsto_{fp} \textbf{err}$.*

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 other variants of the neighborhood monad modeling non-deterministic, state-dependent, and probabilistic rounding behaviors. We discuss these extensions in the supplementary materials.

Non-deterministic rounding. In effectful programming languages, non-deterministic choice can be modeled by the *powerset* monad P, which maps any set X to the powerset $2^X = \{A \mid A \subseteq X\}$. This monad comes with a family of maps $\tilde{\mu}_X : P(PX) \to PX$, which simply take the union, and a family of maps $\tilde{\eta}_X : X \to PX$ which returns the singleton set.

We can define variants of the neighborhood monad based on the powerset monad. For any $r \in \mathcal{R}$, we can define functors $TP_r : \textbf{Met} \to \textbf{Met}$ with carrier sets

$$TP^+_r(X, d_X) \triangleq \{(x, A) \in X \times PX \mid \text{for all } a \in A,\ d_X(x, a) \leq r\}$$
$$TP^-_r(X, d_X) \triangleq \{(x, A) \in X \times PX \mid \text{exists } a \in A,\ d_X(x, a) \leq r\}$$

and both metrics based on the first component: $d((x, A), (y, B)) \triangleq d_X(x, y)$. The associated maps for this monad are inherited from the associated maps of P. For both TP^+_r and TP^-_r, we define:

$$\eta_X(x) = (x, \tilde{\eta}_X(x))$$
$$\mu_X((x, A), \{(y_i, B_i) \mid i \in I\}) = (x, \tilde{\mu}_X(\{B_i \mid i \in I\}))$$

Theorem 7.6. *The maps η_X and μ_X make TP^+_r and TP^-_r into a $\mathcal{R}$-graded monad.*

Proof. It is clear that $\eta_X : X \to TP_0X$ on the carrier sets; this map is non-expansive by the definition of the metric on TP_r. We can also check that $\mu_X : TP_q(TP_rX) \to TP_{q+r}X$ for both variants; we consider TP^+_r, but the other variant is exactly the same. Suppose that

$$((x, A), \{(y_i, B_i) \mid i \in I\}) \in TP^+_q(TP^+_rX).$$

Then by definition of TP_r^+, the distance d_X between y_i and every element in B_i is at most r, and by definition of TP_q^+, the distance $d_{TP_r^+X}$ between (x, A) and every element (y_i, B_i) is at most r, so the d_X distance between x and y_i is at most r for every i by definition of the metric. Thus by the triangle inequality, the distance d_X between X and every element of B_i is at most $q + r$ as desired. Non-expansiveness of this map is straightforward, the two diagrams required for a graded monad follow from the monad laws for P. □

This monad can model numerical computations with non-deterministic aspects. For instance, rounding could behave non-deterministically in the event of ties, or when the standard allows implementation-specific behavior.

The two variants of the neighborhood monad provide different guarantees: using TP_r^+ ensures that *all* resolutions of the non-determinism differ from the ideal value by at most r, while using TP_r^- ensures that *some* resolution of the non-determinism differs from the ideal value by at most r; this situation seems analogous to *may* versus *must* non-determinism [33].

State-dependent rounding. Another typical monad for computational effects is the *global state* monad S. Given a fixed set Σ of possible global states, the state monad maps any set X to the set of functions $\Sigma \to \Sigma \times X$. The state monad comes with a family of maps $\tilde{\mu}_X : S(SX) \to SX$, which flattens a nested stateful computation by sequencing, and a family of maps $\tilde{\eta}_X : X \to SX$, which maps $x \in X$ to the stateful computation that preserves its state and always produces x.

For any $r \in \mathcal{R}$, we can define functors TS_r : **Met** $\to$ **Met** with carrier sets

$$TS_r(X, d_X) \triangleq \{(x, f) \in X \times SX \mid \text{for all } s \in S,\ f(s) = (s', a) \text{ and } d_X(x, a) \leq r\}$$

and metrics based on the first component: $d((x, f), (y, g)) \triangleq d_X(x, y)$. The associated maps for this monad are inherited from the associated maps of S:

$$\eta_X(x) = (x, \tilde{\eta}_X(x))$$
$$\mu_X((x, f), g \in S(TS_rX)) = (x, S(\pi_2)(g); \tilde{\mu}_X)$$

Theorem 7.7. *The maps η_X and μ_X make TS_r into a $\mathcal{R}$-graded monad.*

Proof. It is clear that $\eta_X : X \to TS_0X$ on the carrier sets; this map is non-expansive by the definition of the metric on TS_r. We can also check that $\mu_X : TS_q(TS_rX) \to TS_{q+r}X$. Suppose that

$$((x, f \in SX), g \in S(TS_rX)) \in TS_q(TS_rX).$$

Then for every state $\sigma \in \Sigma$, $g(\sigma) = (\sigma', (y, h)) \in \Sigma \times TS_rX$ so for every state $\sigma' \in \Sigma$, the distance d_X between y and the output z in $(-, z) = h(\sigma')$ is at most r. By the definition of TS_q, the distance between (x, f) and $g(\sigma) = (y, h)$ is at most q, so the distance d_X between x and y is at most q. Thus by the triangle inequality, the distance between x and z is at most $q+r$ as desired. Non-expansiveness of this map is straightforward, the two diagrams required for a graded monad follow from the monad laws for S. □

This monad can be used to model rounding behavior that depends on the machine-specific state, such as floating point registers. The definition of TS_r ensures that stateful computations always produce values that are at most r from the ideal value, regardless of the initial state of the system.

Randomized rounding. Finally, we can consider layering the neighborhood monad with monads for randomized choice. For instance, the *(finite) distribution* monad D maps any set X to the set of probability distributions over X with finite support. This monad comes with a family of maps $\tilde{\mu}_X : D(DX) \to DX$, which flattens a distribution over distributions, and a family of maps $\tilde{\eta}_X : X \to DX$, which returns a point (deterministic) distribution. Much like our previous monads, we can define functors $TD_r : \mathbf{Met} \to \mathbf{Met}$ where the carrier set of $TD_r X$ is a subset of $X \times DX$. There are at least three choices for which subset to take, however:

(1) $(x, p : DX) \in TD_r X$ iff $d_X(x, a) \leq r$ for *all* a in the support of p.
(2) $(x, p : DX) \in TD_r X$ iff $d_X(x, a) \leq r$ for *some* a in the support of p.
(3) $(x, p : DX) \in TD_r X$ iff the *expected value* of $d_X(x, a)$ when a is drawn from p is at most r.

The unit η_X and multiplication μ_X maps are inherited from $\tilde{\eta}_X$ and $\tilde{\mu}_X$, just as before.

Theorem 7.8. *The maps η_X and μ_X make all variants of TD_r into a $\mathcal{R}$-graded monad.*

Proof. The first two variants follow from the essentially the same argument for TP_r^+ and TP_r^-, respectively. We focus on the third variant. As with the other cases, the most intricate point to check is that $\mu_X : TD_q(TD_r X) \to TD_{q+r} X$ on carrier sets. The argument follows from the triangle inequality and the law of total expectations. We write $[x_i : \alpha_i \mid i \in I]$ where $x_i \in X$ and $\alpha_i \in [0,1]$ sum up to 1 for a distribution in DX. Suppose that:

$$((x, -), p = [(y_i, p_i) : \alpha_i \mid i \in I]) \in TD_q(TD_r X) \qquad \text{and} \qquad p_i = [z_i : \beta_{ij} \mid j \in J] \in DX$$

Then by definition of $TD_r X$, we have $\mathbb{E}_{z \sim p_i}[d_X(y_i, z)] \leq r$ for every i, and by definition of $TD_q X$ and the metric on TD_q, we have $\mathbb{E}_{y \sim p}[d_X(x, y)] \leq q$. By definition, μ_X maps the input $((x, -), p)$ to (x, m), where $m = D(\pi_2)(p); \tilde{\mu}_X$. Now we can compute:

$$\begin{aligned}
\mathbb{E}_{z \sim m}[d_X(x, z)] &= \mathbb{E}_{y_i \sim p}[\mathbb{E}_{z \sim p_i}[d_X(x, z)]] && \text{(def. } \tilde{\mu}_X\text{)} \\
&\leq \mathbb{E}_{y_i \sim p}[\mathbb{E}_{z \sim p_i}[d_X(x, y) + d_X(y, z)]] && \text{(triangle ineq.)} \\
&= \mathbb{E}_{y_i \sim p}[d_X(x, y) + \mathbb{E}_{z \sim p_i}[d_X(y, z)]] && \text{(constant)} \\
&\leq \mathbb{E}_{y_i \sim p}[d_X(x, y) + r] \leq q + r && \text{(def. } TD_r \text{ and } TD_q\text{)}
\end{aligned}$$

□

The first two choices provide worst-case and best-case bounds on numerical error when rounding may be probabilistic. The last choice is particularly interesting: it provides an *average-case* bound on the numerical error between the floating-point value and the ideal value.

Towards a more uniform picture. We have outlined several examples of composing the graded neighborhood monad with monads for computational effects, but we do not understand the picture in full generality. We see several interesting questions, which we leave for future work:

- How can we define the carrier sets for the graded monad? For some of our computational monads we seem to be able to define multiple versions of the graded monad, but currently it is not clear which effectful monads support which graded versions.
- What properties of the effect monad carry over to the neighborhood monad? For instance, we conjecture that as long as the effect monad is strong, then the graded monad is also strong. It is currently not clear what other properties of the effect monad are preserved.
- How generally does this construction work? The main ingredients in our construction seem to be the neighborhood monad on **Met** and an effect monad over **Set**. A natural question is whether these categories can be generalized further.

8 RELATED WORK

Type systems for floating-point error. A type system due to Martel [36] uses dependent types to track the occurrence and propagation of *representation errors* in floating-point numbers; i.e., the loss in precision due to the fact that floating-point numbers can 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; there are now benchmarks like FPBench [14] to compare different approaches. The earliest methods, like Fluctuat [25] and Gappa [19] employ abstract interpretation with interval arithmetic or polyhedra [10] to overapproximate the range of absolute roundoff errors in computations. This method is flexible and applies to general programs with conditionals and loops, but it can substantially overestimate the roundoff error in computations with non-linear arithmetic, and it is difficult to accurately model cancellation of errors.

To provide more precise bounds, more recent work relies on optimization. Conceptually, these methods represent the roundoff error symbolically as a function of the program inputs and the error terms introduced during the computation, and then perform global optimization over all settings of the errors in order to bound the roundoff error. Since the error expressions are typically quite complex, verification methods use a approximations to simplify the error expression to make optimization more tractable, and mostly focus on straight-line programs. For instance, Real2Float [35] 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 [43] was the first tool to use Taylor approximations to polynomial approximations to the error bound. Daisy [15] also uses methods based on Taylor approximations, but the method is not as optimized as FPTaylor's. Abbasi and Darulova [3] describe a modular method for bounding the propagation of input and roundoff errors using FPTaylor's method, and the tool Rosa [16, 17] uses Taylor approximations to approximate the propagation of errors in possbily 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, 13, 42] and we expect they should be possible in $\Lambda_{\mathbf{num}}$; however, the precision of the error bounds for programs using general recursion would likely be poor. One other limitation of our method is in typing conditionals: while $\Lambda_{\mathbf{num}}$ can derive error bounds when the ideal and floating-point executions follow the same branches, 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 [26–28]. Formalized specifications of floating-point arithmetic have been developed in the Coq [8], Isabelle [47], and PVS [37] proof assistants. These specifications have been used to develop sound tools for floating-point error analysis that generate proof certificates, such as VCFloat [41] and PRECiSA [45]. They have also been used to mechanize proofs of error bounds for specific numerical programs (e.g., [1, 32, 44]). Work by Cousot et al. [12] 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 [40] automatically rewrites numerical programs to reduce numerical error, while RLibm [34] 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 [42], a type system for verifying differential privacy, many researchers have investigated this idea. We point out a few especially relevant works. Our syntax and typing rules are inspired by Dal Lago and Gavazzo [13], 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. In $\Lambda_{\mathbf{num}}$, as in Fuzz, the distance between left and right injections is ∞, whereas in the system by Dal Lago and Gavazzo [13], left and right injections are not related at any distance. Our approach allows non-trivial operations returning booleans to be typed as infinite sensitivity functions, but the case rule must be adjusted: to preserve soundness, the conclusion must retain a dependence on the guard expression, even if the guard is not used in the branches.

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. Finally, recent work by wunder et al. [46] proposes a variant of Fuzz with "bunched" (tree-shaped) contexts, with two methods of combining contexts. It could be interesting to developed a bunched version of $\Lambda_{\mathbf{num}}$—the metrics for $\otimes$ and $\times$ could be naturally accommodated in the contexts, possibly leading to more precise error analysis.

Other approaches to error analysis. The numerical analysis literature has explored other conceptual tools for static error analysis, such as stochastic error analysis [11], and backwards error analysis. Techniques for *dynamic* error analysis have also been proposed [29]. These methods aim to estimate and compensate for the rounding error of specific computations at runtime. It would be interesting to consider these techniques from a formal methods perspective, whether by connecting dynamic error analysis with ideas like runtime verification, or developing methods to verify the correctness of dynamic error analysis.

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 [30]. 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 *backwards 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.

REFERENCES

[1] 2014. Trusting computations: A mechanized proof from partial differential equations to actual program. *Computers & Mathematics with Applications* 68, 3 (2014), 325–352.

[2] 2019. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), 1–84. https://doi.org/10.1109/IEEESTD.2019.8766229

[3] Rosa Abbasi and Eva Darulova. 2023. Modular Optimization-Based Roundoff Error Analysis of Floating-Point Programs. In *Static Analysis*, Manuel V. Hermenegildo and José F. Morales (Eds.). Springer Nature Switzerland, Cham, 41–64.

[4] Arthur Azevedo de Amorim, Marco Gaboardi, Emilio Jesús Gallego Arias, and Justin Hsu. 2014. Really Natural Linear Indexed Type Checking. In *Proceedings of the 26nd 2014 International Symposium on Implementation and Application of Functional Languages* (Boston, MA, USA) *(IFL '14)*. Association for Computing Machinery, New York, NY, USA, Article 5, 12 pages. https://doi.org/10.1145/2746325.2746335

[5] Arthur Azevedo de Amorim, Marco Gaboardi, Justin Hsu, Shin-ya Katsumata, and Ikram Cherigui. 2017. A semantic account of metric preservation. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 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 *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–19. https://doi.org/10.1109/LICS.2019.8785715

[7] 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

[8] Sylvie Boldo and Guillaume Melquiond. 2011. Flocq: A Unified Library for Proving Floating-Point Algorithms in Coq. In *2011 IEEE 20th Symposium on Computer Arithmetic*. 243–252. https://doi.org/10.1109/ARITH.2011.40

[9] Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. 2014. A Core Quantitative Coeffect Calculus. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings (Lecture Notes in Computer Science, Vol. 8410)*, Zhong Shao (Ed.). Springer, 351–370. https://doi.org/10.1007/978-3-642-54833-8_19

[10] Liqian Chen, Antoine Miné, and Patrick Cousot. 2008. A Sound Floating-Point Polyhedra Abstract Domain. In *Programming Languages and Systems, 6th Asian Symposium, APLAS 2008, Bangalore, India, December 9-11, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 5356)*, G. Ramalingam (Ed.). Springer, 3–18. https://doi.org/10.1007/978-3-540-89330-1_2

[11] 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

[12] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2005. The ASTREÉ Analyzer. In *Programming Languages and Systems, 14th European Symposium on Programming, ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3444)*, Shmuel Sagiv (Ed.). Springer, 21–30. https://doi.org/10.1007/978-3-540-31987-0_3

[13] Ugo Dal Lago and Francesco Gavazzo. 2022. A Relational Theory of Effects and Coeffects. 6, POPL, Article 31 (jan 2022), 28 pages. https://doi.org/10.1145/3498692

[14] 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 *Numerical Software Verification*, Sergiy Bogomolov, Matthieu Martel, and Pavithra Prabhakar (Eds.). Springer International Publishing, Cham, 63–77.

[15] Eva Darulova, Anastasiia Izycheva, Fariha Nasir, Fabian Ritter, Heiko Becker, and Robert Bastian. 2018. Daisy - Framework for Analysis and Optimization of Numerical Programs (Tool Paper). In *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10805)*, Dirk Beyer and Marieke Huisman (Eds.). Springer, 270–287. https://doi.org/10.1007/978-3-319-89960-2_15

[16] Eva Darulova and Viktor Kuncak. 2014. Sound Compilation of Reals *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 235–248. https://doi.org/10.1145/2535838.2535874

[17] Eva Darulova and Viktor Kuncak. 2017. Towards a Compiler for Reals. *ACM Trans. Program. Lang. Syst.* 39, 2, Article 8 (mar 2017), 28 pages. https://doi.org/10.1145/3014426

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

[19] 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

[20] Soichiro Fujii, Shin-ya Katsumata, and Paul-André Melliès. 2016. Towards a Formal Theory of Graded Monads. In *Foundations of Software Science and Computation Structures - 19th International Conference, FOSSACS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9634)*, Bart Jacobs and Christof Löding (Eds.). Springer, 513–530. https://doi.org/10.1007/978-3-662-49630-5_30

[21] Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C. Pierce. 2013. Linear Dependent Types for Differential Privacy. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. Association for Computing Machinery, New York, NY, USA, 357–370. https://doi.org/10.1145/2429069.2429113

[22] Marco Gaboardi, Shin-ya Katsumata, Dominic A. Orchard, Flavien Breuvart, and Tarmo Uustalu. 2016. Combining effects and coeffects via grading. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM, 476–489. https://doi.org/10.1145/2951913.2951939

[23] 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

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

[25] Eric Goubault and Sylvie Putot. 2006. Static Analysis of Numerical Algorithms. In *Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4134)*, Kwangkeun Yi (Ed.). Springer, 18–34. https://doi.org/10.1007/11823230_3

[26] John Harrison. 1997. Floating point verification in HOL light: The exponential function. In *Algebraic Methodology and Software Technology*, Michael Johnson (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 246–260.

[27] John Harrison. 1999. A Machine-Checked Theory of Floating Point Arithmetic. In *Theorem Proving in Higher Order Logics*, Yves Bertot, Gilles Dowek, Laurent Théry, André Hirschowitz, and Christine Paulin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 113–130.

[28] John Harrison. 2000. Formal Verification of Floating Point Trigonometric Functions. In *Formal Methods in Computer-Aided Design*, Warren A. Hunt and Steven D. Johnson (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 254–270.

[29] Nicholas J. Higham. 2002. *Accuracy and Stability of Numerical Algorithms* (second 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

[30] 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

[31] Shin-ya Katsumata. 2014. Parametric effect monads and semantics of effect systems. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 633–646. https://doi.org/10.1145/2535838.2535846

[32] Ariel E. Kellison and Andrew W. Appel. 2022. Verified Numerical Methods for Ordinary Differential Equations. In *Software Verification and Formal Methods for ML-Enabled Autonomous Systems - 5th International Workshop, FoMLAS 2022, and 15th International Workshop, NSV 2022, Haifa, Israel, July 31 - August 1, and August 11, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13466)*, Omri Isac, Radoslav Ivanov, Guy Katz, Nina Narodytska, and Laura Nenzi (Eds.). Springer, 147–163. https://doi.org/10.1007/978-3-031-21222-2_9

[33] Søren Bøgh Lassen. 1998. *Relational Reasoning about Functions and Nondeterminism.* Ph. D. Dissertation. University of Aarhus. BRICS DS-98-2.

[34] 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. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–28. https://doi.org/10.1145/3498664

[35] 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

[36] Matthieu Martel. 2018. Strongly Typed Numerical Computations. In *Formal Methods and Software Engineering - 20th International Conference on Formal Engineering Methods, ICFEM 2018, Gold Coast, QLD, Australia, November 12-16, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11232)*, Jing Sun and Meng Sun (Eds.). Springer, 197–214. https://doi.org/10.1007/978-3-030-02450-5_12

[37] Paul S. Miner. 1995. Formal Specification of IEEE Floating-Point Arithmetic Using PVS. *IFAC Proceedings Volumes* 28, 25 (1995), 31–36. https://doi.org/10.1016/S1474-6670(17)44820-8 2nd IFAC Workshop on Safety and Reliability in Emerging Control Technologies, Daytona Beach, USA, 1-3 November.

[38] 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

[39] 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

[40] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. 2015. Automatically Improving Accuracy for Floating Point Expressions. *SIGPLAN Not.* 50, 6 (jun 2015), 1–11. https://doi.org/10.1145/2813885.2737959

[41] Tahina Ramananandro, Paul Mountcastle, Benoît Meister, and Richard Lethin. 2016. A Unified Coq Framework for Verifying C Programs with Floating-Point Computations. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs* (St. Petersburg, FL, USA) *(CPP 2016)*. Association for Computing Machinery, New York,

NY, USA, 15–26. https://doi.org/10.1145/2854065.2854066

[42] Jason Reed and Benjamin C. Pierce. 2010. Distance Makes the Types Grow Stronger: A Calculus for Differential Privacy. *SIGPLAN Not.* 45, 9 (sep 2010), 157–168. https://doi.org/10.1145/1932681.1863568

[43] Alexey Solovyev, Marek S. Baranowski, Ian Briggs, Charles Jacobsen, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. 2018. Rigorous Estimation of Floating-Point Round-Off Errors with Symbolic Taylor Expansions. 41, 1, Article 2 (dec 2018), 39 pages. https://doi.org/10.1145/3230733

[44] 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. Springer-Verlag, Berlin, Heidelberg, 206–221. https://doi.org/10.1007/978-3-031-42753-4_14

[45] 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 *Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10747)*, Isil Dillig and Jens Palsberg (Eds.). Springer, 516–537. https://doi.org/10.1007/978-3-319-73721-8_24

[46] june wunder, Arthur Azevedo de Amorim, Patrick Baillot, and Marco Gaboardi. 2023. Bunched Fuzz: Sensitivity for Vector Metrics. In *Programming Languages and Systems*, Thomas Wies (Ed.). Springer Nature Switzerland, Cham, 451–478.

[47] 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.