

# Negative Fuzz: A Type-Based Approach to Numerical Analysis With Negative Numbers

ANONYMOUS AUTHOR(S)

Bounding floating-point error is important for many mathematical and scientific programs. However, many automatic approaches to verifying floating-point error have scalability problems. Type-based approaches, such as Numerical Fuzz, offer a way to scalably and precisely bound round-off error. However, Numerical Fuzz does not support negative numbers or subtraction and is overly conservative with its treatment of addition and subtraction. We define a *paired model* for Numerical Fuzz that can soundly represent negative numbers and subtraction. We also improve the precision of addition and subtraction via a new primitive. To compute error bounds, we develop a modular interval-style bounds analysis in the type system. We further develop *bound polymorphism* so that functions can be polymorphic over the intervals their inputs belong to. Our bounds analysis requires little work to use; we automatically infer all interval-style bounds and instances of bound polymorphism. We find that our approach is faster than competing approaches, sometimes by orders of magnitude, while giving useful and competitive error bounds.

CCS Concepts: • **Software and its engineering** → **Functional languages**; • **Theory of computation** → **Type theory**; **Logic and verification**; • **Mathematics of computing** → Interval arithmetic.

Additional Key Words and Phrases: linear type systems, verification, round-off error, floating point

## ACM Reference Format:

Anonymous Author(s). 2026. Negative Fuzz: A Type-Based Approach to Numerical Analysis With Negative Numbers. 1, 1 (February 2026), 38 pages. <https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

## 1 INTRODUCTION

It is natural for users such as mathematicians and scientists to desire programs that operate over the reals. As language designers, we wish to show that the floating-point semantics we have provided is “close enough”. Towards this end, we develop automated numerical analyses to bound the error introduced by our floating-point approximations. We wish for our analyses to be efficient, as automatic as possible, and precise.

A major challenge in the automated numerical analysis literature is to perform a scalable analysis with precise error bounds. Many existing approaches rely on global optimization [Solovyev et al. 2018] [Tirpankar et al. 2025], interval analysis [Daumas and Melquiond 2010], or SMT-based methods [Darulova and Kuncak 2017]. However these approaches frequently time-out or fail to produce usable error bounds on large programs. Type-based techniques offer an alternative approach that is inherently compositional and scalable: all of the information necessary to perform an error analysis on a function is contained within the type.

In this paper, we extend Numerical Fuzz [Kellison and Hsu 2024], an affine call-by-value graded type system for bounding round-off error. The core idea of Numerical Fuzz is to track two key properties of each function: how a function magnifies error in its inputs (function sensitivity), and the maximum error introduced by the function. This allows Numerical Fuzz to compositionally

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM XXXX-XXXX/2026/2-ART

<https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

bound the maximum accumulated error in a program via a graded monad. There are two limitations of Numerical Fuzz that our work addresses:

- (1) **Lack of support for negative numbers or subtraction:** Subtraction over the reals is infinitely-sensitive, which leads Numerical Fuzz to conservatively conclude that any use of subtraction might lead to infinite round-off error. Since many programs use subtraction or have negative numbers, this limitation severely restricts the practical utility of Numerical Fuzz in modeling real-world programs.
- (2) **Conservative treatment of addition and subtraction:** Floating-point error of addition and subtraction is known to grow in the *height* of a computation tree. In Numerical Fuzz, the bounded error of a sequence of additions grows in the *number of nodes* in the corresponding computation tree. For perfect binary computation trees, this leads to a log-factor difference in the obtained error bound. As many programs use addition or subtraction, this limits the precision of the Numerical Fuzz analysis.

### Our approach: Negative Fuzz

Supporting subtraction and improving the precision of addition are orthogonal contributions, so we describe them separately:

*Supporting subtraction and negative numbers.* To support subtraction and negative numbers, we first introduce a *paired model* that simulates the floating-point error of the standard, unpaired program model. Our paired model associates each possibly negative real  $r$  in our semantics with a pair of non-negative  $(a, b)$  such that  $r = a - b$ . Addition and subtraction over the reals can be modeled as always-growing and finitely-sensitive additions over the paired components. To reason about the unpaired and paired models side-by-side, we use a triple  $(r, a, b)$  where  $r \in \mathbb{R}$  and  $a, b \in \mathbb{R}^+$  and  $r = a - b$ . For our paired approach to be useful in bounding the round-off error on the unpaired  $r$  representing real-world floating-point programs, we need to be able to (1) “simulate” floating-point round-off error on  $r$  a special error-preserving paired rounding function, (2) measure error on the paired model, and (3) use our interval-style bounds analysis to “translate” error bounds on the paired model to be back on the unpaired  $r$  representing the real-world program. We defer detailed discussion of 1-3 to the technical sections of this paper.

Our bounds analysis occurs purely in the type system. To aid usability, we support analyzing functions via *bound polymorphism*, which allows functions to be polymorphic in the bounds assigned to the inputs and to be type-checked independently. To reduce the user type annotation burden, we also provide a type inference algorithm that can automatically generalize and instantiate bound-polymorphic functions. We prove the soundness of our type inference algorithm and evaluate its utility and performance in Section 6.

*More precise analysis of addition and subtraction.* Floating-point addition and subtraction are not associative. In fact, different associations of the same computation can result in drastically different error terms. Unfortunately, Numerical Fuzz is not capable of distinguishing between different associations. We address this limitation by introducing a primitive that allows error on a cartesian pair to be computed as the maximum error on the components, which more accurately captures the categorial semantics of our graded monad introduced in Numerical Fuzz [Kellison and Hsu 2024]. We prove that our primitive is type-sound and show how to use the primitive in Section 5.

*Contributions summary.* Concretely, our contributions are as follows:

- We extend the Numerical Fuzz family of languages by incorporating an bounds analysis into the type system using *bound polymorphism*. To make programs easier and more concise

to write, we further extend Numerical Fuzz to enable expressions in more places. We prove the soundness of our extensions to Numerical Fuzz in (Section 3).

- We develop a *paired model* for Numerical Fuzz that allows for modeling subtraction and negative numbers in a finitely-sensitive manner (Section 4). We provide a polymorphic bounds analysis in the type system to reason about our paired model. We provide error soundness theorems to allow the error grade and bounds analysis from program type to be translated into a floating-point error bound on the standard, unpaired program model.
- We improve the precision of Numerical Fuzz by enabling the analysis to more efficiently reason about addition and subtraction. We accomplish this by introducing a new primitive that can more expressively capture different techniques for sequencing addition and subtraction operations (Section 5). In every benchmark that we rewrite to use our primitive, we observe significant improvements on the precision of the inferred error bounds.
- We develop a type inference algorithm for our type system that can automatically infer function sensitivities, round-off error bounds, and bound polymorphism in many useful programs; the user merely supplies a program without bound polymorphism, sensitivity annotations, or round-off grades. We prove the soundness of our type inference algorithm.
- We implement our type-based approach for error analysis. When compared against competing approaches (FPTaylor, Gappa and Satire) on a suite of benchmarks translated into our core language, we obtain useful and competitive error bounds with faster and more scalable performance and low annotation overhead (Section 7).

## 2 TECHNICAL OVERVIEW

In this paper, we are primarily motivated to extend Numerical Fuzz to support more floating-point operations and richer, more precise ways to reason about addition and subtraction. We call our extension Negative Fuzz because it supports negative numbers and subtraction. We first provide some background and our motivating running example. Then, we overview our extensions.

### 2.1 Background: floating-point and error metrics

The binary floating-point number system is a family of formats with the following setup:

$$x = (-1)^s \cdot m \cdot 2^{e-p+1} \quad (1)$$

where  $s$  controls the sign,  $m$  is the mantissa,  $e$  is the exponent, and  $p$  is the precision. A specific floating-point number system fixes a particular precision  $p$  and represents a subset of the reals, for example `BINARY32` and `BINARY64` which fix  $p = 32$  and  $p = 64$  respectively. Arithmetic operations with round-off error can be represented in a floating-point system as the exact operation followed by rounding to a float by a rounding function (written  $\rho$ ). The error introduced by the rounding function  $\rho$  is known as *round-off error*. The choice of rounding function will vary depending on the rounding mode (e.g. round-to-nearest, round-towards-infinity, etc.) and the precision used.

For this paper, we assume the absence of overflow and underflow. Then, the standard floating-point rounding error model assumes  $\rho$  will observe round-off error bounded in the following way:

$$\rho(x) = x(1 + \delta) \quad |\delta| \leq u \quad (2)$$

where  $u$  is the *unit round-off* value representing the maximum error possible in the last bit for the floating point format (e.g.  $2^{-24}$  and  $2^{-53}$  for 32 and 64-bit floating-point numbers respectively).

We are interested in bounding round-off error. To bound error, one must first have a suitable definition for measuring error. Following Numerical Fuzz, we rely on Olver's [Olver 1978] *relative precision* metric for measuring round-off error.

$$d_{\mathbb{R}}(r, \tilde{r}) = |\ln(\frac{r}{\tilde{r}})| \quad (3)$$

## 2.2 Running example: pairwise summation

We consider the well-known pairwise summation (sometimes referred to as cascade summation) algorithm. As a concrete running example, we consider adding up the floating-point numbers  $w, x, y, z \in [-1, 1]$  via pairwise summation:

$$\rho(\rho((w + x)) + \rho((y + z))) \quad (4)$$

## 2.3 Supporting Subtraction and Negative Numbers

Numerical Fuzz can handle inputs in the range  $[0, 1]$  but cannot handle our example which includes negative numbers. This is because subtraction in Numerical Fuzz and addition over the unrestricted reals is infinitely sensitive. The core problem is that relatively small perturbations in the input of subtraction can lead to hard-to-bound relatively large perturbations in the output. As an illustrative toy example, suppose we have two large numbers like 1,000,001 and 1,000,000 that we wish to subtract. If we perturb the input by a small amount, we can observe quite large *relative* changes in the output. In other words, subtracting two large nearby numbers with small round-off error in its inputs could obtain a small number with arbitrarily high relative round-off error. This feature of subtraction is sometimes referred to as *catastrophic cancellation* and forms one of the primary technical obstacles to verifying our running example.

Note that supporting addition with negative numbers is effectively the same problem as supporting subtraction. To understand the problem posed by an infinitely-sensitive operation, we try to write our running example using **rnd** to represent our floating point rounding function  $\rho$ , **add** to represent exact addition (+) and **let-bind** to sequence floating-point computation:

$$\begin{aligned} \text{let-bind } q &= \text{rnd } (\text{add } \langle w, x \rangle) \text{ in} \\ \text{let-bind } p &= \text{rnd } (\text{add } \langle y, z \rangle) \text{ in} \\ &\text{rnd } (\text{add } \langle q, p \rangle) \end{aligned} \quad (5)$$

under the restriction that  $w, x, y, z \in [-1, 1]$ . We will show how the program cannot be typed in Numerical Fuzz. We will first assign a type in Negative Fuzz to a simplified toy example. Then, we will demonstrate how to use the type to obtain a sound error bound.

*Standard representation.* Numerical Fuzz's type system relies on bounding function sensitivity. We first attempt to assign a type to a hypothetical addition primitive in Numerical Fuzz over the unrestricted reals: **add** :  $!_{\infty}(\text{num} \times \text{num}) \multimap \text{num}$ . Then, the type of our running example is  $M_{\infty} \text{ num}$  in the context  $w : \text{num}, x : \text{num}, y : \text{num}, z : \text{num}$ .<sup>1</sup> Our type comes from two facts. Firstly, due to **rnd**, we know that there can be up to  $u$  error introduced in the inputs  $p$  and  $q$  to the last **add** call. Secondly, since the **add** function may amplify error in its input by up to  $\infty$ , the round-off error of the whole program is  $\infty \cdot u = \infty$  where  $u$  is our unit-roundoff constant. Therefore, we know that our example program is both infinitely-sensitive in its free variables and has infinitely-bounded error. This is not a useful error bound.

<sup>1</sup>For presentational purposes, we elide the sensitivity co-effect annotation in the typing context. The program is infinitely-sensitive in variables  $w, x, y, z$ .

*Paired representation.* As addition and subtraction over the unrestricted reals is infinitely-sensitive, we first deal with the problem of infinite sensitivity.

We develop a paired representation where unrestricted addition and subtraction have finitely-bounded sensitivity. The trick is to associate for each real  $r \in \mathbb{R}$  a triple  $(r, a, b) \in \mathbb{P} = \mathbb{R} \times \mathbb{R}^+ \times \mathbb{R}^+ = \text{num}$  such that  $r = a - b$ . The paired components  $a$  and  $b$  are only used for error analysis and serve a function similar to ghost variables in other program analyses. Our error function  $d_{\mathbb{P}}((r, a, b), (\tilde{r}, \tilde{a}, \tilde{b}))$  over  $\mathbb{P}$  is defined below:

*Definition 2.1 (Distance over pairs).*  $d_{\mathbb{P}}((r, a, b), (\tilde{r}, \tilde{a}, \tilde{b})) = \max(d_{\mathbb{R}}(a, \tilde{a}), d_{\mathbb{R}}(b, \tilde{b}))$ .

Importantly, we can encode our operations, including addition and subtraction, such that both components of the pair are always-growing. This avoids the problem of catastrophic cancellation in the paired representation. When our semantics adds  $r_0 = (a_0, b_0)$  and  $r_1 = (a_1, b_1)$ , we produce  $r_0 + r_1 = (a_0 + a_1, b_0 + b_1)$ . When we subtract two numbers  $r_0 - r_1 = (a_0 + b_1, b_0 + a_1)$ , our paired type representation only needs to add components; subtraction is represented implicitly. By ensuring that each operation in the paired representation is always-growing, we can encode addition and subtraction in a finitely-sensitive manner.

Under the paired representation, we can implement and faithfully type addition, subtraction, and multiplication in terms of finitely-sensitive operations over the paired components. Addition is defined below:

$$\begin{aligned} \text{add}((r, a, b), (r', a', b')) &\mapsto (r + r', a + a', b + b') \\ \text{sub}((r, a, b), (r', a', b')) &\mapsto (r - r', a + b', a' + b) \end{aligned} \quad (6)$$

In this manner, we can bound the sensitivity of addition and subtraction and soundly assign a type. **add** and **mul** have the same type as in Numerical Fuzz. Subtraction has type as addition: **sub** : **num**  $\times$  **num**  $\multimap$  **num**. Returning to our running example, we can assign the type **num**  $\times$  **num**  $\multimap M_u \text{num}$  to the program  $\lambda x. \text{let } z = \text{sub } x \text{ in rnd } z$  where  $u$  is the unit round-off constant.

*Using the type to assign an error bound.* Now we wish to translate our type over the paired model into a error bound over the unpaired model. Our type system will assign the type  $M_{3 \cdot u} \text{num}((-4,4),(0,4),(0,4))$  to the program. In order to obtain a useful error bound, we need two pieces of information derived from the program's type: (1) the result of an interval-style analysis on the exact, unrounded versions of  $r, a, b$ , and (2) the relative precision bound between the exact and approximate components between  $d_{\mathbb{R}}(a, \tilde{a})$  and  $d_{\mathbb{R}}(b, \tilde{b})$ , which comes from our  $3u$  monad grade.

To obtain (1), we run an interval-style analysis on the exact versions of  $r, a, b$ . We observe that in our running example, we have that the paired representation of  $w, x, y, z \in [-1, 1] \times [0, 1] \times [0, 1]$ . Our bounds analysis will take this input and determine that the ideal computation has type  $\text{num}((-4,4),(0,4),(0,4))$  ensures that the program result will be within  $[-4, 4] \times [0, 4] \times [0, 4]$  with the invariant for every triple  $(r, a, b)$  the equation  $r = a - b$  holds. To obtain (2), we can deduce from the  $3u$  grade of our monad to conclude that for all inputs  $w, x, y, z$  to our input program, for an output value  $(r, a, b), (\tilde{r}, \tilde{a}, \tilde{b})$ , that  $\max(d(a, \tilde{a}), d(b, \tilde{b})) \leq 3 \cdot u$ . Combining (1) and (2), we can apply our main absolute error theorem, Theorem 4.10, for 64-bit floats and obtain the absolute error bound:  $r - \tilde{r} = 2.6645352591003757 \times 10^{-15}$ .

*Bound polymorphism.* There is some difficulty in assigning a single, most precise type to **add**. The difficulty arises because **add** is called multiple times in different contexts. If we look at the first two add calls, we would try to assign the type:

$$\text{add} : \text{num}((-1,1),(0,1),(0,1)) \times \text{num}((-1,1),(0,1),(0,1)) \multimap \text{num}((-2,2),(0,2),(0,2))$$

However, this would make it impossible to type the second add call. On the other hand, if we look at the last add call, we would try to assign the type:

$$\mathbf{add} : \mathbf{num}_{((-2,2),(0,2),(0,2))} \times \mathbf{num}_{((-2,2),(0,2),(0,2))} \multimap \mathbf{num}_{((-4,4),(0,4),(0,4))}$$

However, this would make our analysis imprecise in our first two calls to **add**.

To ensure that our bounds analysis is scalable and that functions types can be reused, we support *bound polymorphism*, which allows functions to be specialized with different concrete bounds  $b$  at each call site. We write types  $\tau$  polymorphic in bound variable  $\epsilon$  as  $\forall \epsilon. \tau$ . By adding *bound polymorphism*, we allow functions to be typed like so:  $id : \forall \epsilon. \mathbf{num}_\epsilon \multimap \mathbf{num}_\epsilon$ . We can introduce and eliminate polymorphism via  $\forall$  introduction and elimination rules. We also introduce *bound operations*, which are meta-level operations that let us type:

$$\mathbf{add} : \forall \epsilon_1, \epsilon_2, \mathbf{num}_{\epsilon_1} \times \mathbf{num}_{\epsilon_2} \multimap \mathbf{num}_{\epsilon_1 + \epsilon_2}$$

To reduce annotation overhead, we provide a type inference algorithm that can automatically infer bound polymorphism abstraction and instantiation sites. We prove our type inference algorithm sound in Section 6 and provide an evaluation in Section 7.

## 2.4 Language Feature: More Precise Treatment of Addition and Subtraction

Returning to our running example, we would expect Numerical Fuzz to infer that the error on our paired summation example has  $2u$  error. However, Numerical Fuzz will infer that the error on our paired summation example has  $3u$  error. The issue is that the monadic computation sequencing rule forces the error analysis to always compute the worst-case round-off error associated with the most error-prone addition summation  $\rho(\rho(\rho(w + x) + y) + z)$ , which yields an undesirable, overly-conservative error analysis.

More generally, we would expect Numerical Fuzz to infer that a binary tree of height  $h$  where each leaf contains a number that the sum of numbers over the tree computed by recursively adding the children of each parent has floating-point error growing in  $O(h)$ . For example, consider the pairwise summation algorithm which will recursively a list of numbers in half and compute the sum of the halves. This produces a computation tree of height  $\lceil \log_2(n) \rceil$ . We would expect Numerical Fuzz's inferred error to grow in  $O(\log_2(n))$ . However, this is not the case; Numerical Fuzz instead infers error growing in size  $O(n)$ .

Our extension enables the user to obtain more precise error bounds that grow linearly in size of the *height* of the computation tree, allowing us to obtain a the expected  $O(\log_2(n))$  bound for pairwise summation. We accomplish this by developing a special primitive **factor** in our term language. Rewriting our running example with our special primitive below, we can see that using **factor** results in a significantly smaller error grade for a program representing the same association and order of operations. A smaller monadic error grade results in a smaller floating-point error bound, reducing the error of our running example by approximately one-third from  $2.6645352591003757 \times 10^{-15}$  to  $1.7763568394002505 \times 10^{-15}$ .

**let-bind**  $q = \mathbf{rnd}(\mathbf{add}\langle w, x \rangle)$  in  
**let-bind**  $p = \mathbf{rnd}(\mathbf{add}\langle y, z \rangle)$  in  
 $\mathbf{rnd}(\mathbf{add}\langle q, p \rangle)$

**let-bind**  $a =$   
**factor**  $\langle \mathbf{rnd}(\mathbf{add}\langle w, x \rangle), \mathbf{rnd}(\mathbf{add}\langle y, z \rangle) \rangle$  in  
 $\mathbf{rnd}(\mathbf{add}\ a)$

(a) Without factor: the type assigned is  $M_{3u} \mathbf{num}$ .

(b) With factor: the type assigned is  $M_{2u} \mathbf{num}$ .

Fig. 1. Side-by-side comparison of addition with and without the **factor** primitive. Note that we omit the bounds subscripts for presentational purposes.

More generally, for perfect binary trees of addition, we have a  $\log_2(n)$  factor improvement on our error bounds when compared to Numerical Fuzz. We defer a detailed explanation of **factor** and how it allows for structuring and sequencing arithmetic towards a more precise error analysis in Section 5. We also showcase the improved precision offered by programs annotated with the **factor** primitive in Section 7.

### 3 LANGUAGE

Our language is an extension of the Numerical Fuzz [Kellison and Hsu 2024] family of call-by-value affine lambda calculi. In this section, we detail the construction of our modular family of languages. We first describe the terms (Section 3.1), types and static semantics (Section ??), operational semantics (Section 3.3), and a modular interface for soundly instantiating the language. Finally, we define a logical relation and prove soundness. In Section 4, we will instantiate our family of languages and demonstrate how to read a type as a floating-point error bound.

#### 3.1 Terms

Like many lambda calculi for linear type systems, Numerical Fuzz supports variables, function abstraction and application,  $\pi_i$  for projection,  $in_i$  for injection, and two different types of products. Our term syntax has explicit terms to represent our effectful computation (rounding to the nearest float: **rnd**  $e$ ) and coefficients (scaling sensitivity:  $[e]$ ). We have several terms for sequencing and structuring computation. We have **let-bind** for sequencing monadic computation, **let-cobind** for sequencing comonadic computation, **let-pair** for unpacking tensors, **case ... of ...** for unpacking sum types, and **let** for sequencing variable assignment. Note that our term language is more expressive than Numerical Fuzz as we do not restrict the bound term in our let expressions to be values. This makes it easier to write more concise programs with less variables needed for storing the results of intermediate computations.

We also introduce a language for bounds (shown in Figure 2) as well as explicit terms to represent polymorphic abstraction ( $\Lambda e.e$ ) and instantiation ( $e\{b\}$ ). This is useful for incorporating an bound analysis into the type inference algorithm, which we detail in Section 6.

#### 3.2 Types and static semantics

Following Numerical Fuzz, our type system has graded monad types  $M_q\tau$  for bounding the round-off error in  $\tau$  by a real, non-negative grade  $q$ , scaled metric types. We also have graded comonadic types  $!_s\tau$  for bounding the sensitivity of a type  $\tau$  by a real, non-negative grade  $s$ . Our language also supports linear function types  $\tau_0 \multimap \tau$  corresponding to 1-sensitive functions.

Our typing judgements incorporate a graded effect and co-effect type system to simultaneously track round-off error and function sensitivity. For example, the following term with typing context and judgement indicates that the variable  $x$  is  $s$ -sensitive in the expression  $e$  and has worst-case round-off error upper-bounded by  $q$ .

$$x :_s \tau_0 \vdash e : M_q\tau_1 \quad (7)$$

Typing contexts have the following grammar:  $\Gamma, \Delta ::= . \mid \Gamma, x :_s \tau$ . Semantically, we can treat contexts  $\Gamma$  as a partial map from variables  $x$  to pairs of sensitivities and types  $(s, \tau)$ . Similarly to Numerical Fuzz, we can sum and scale typing contexts. We represent context scaling as  $\Gamma + \Delta$  and scaling by a real  $s$  as  $s \cdot \Gamma$ . We define pair context and scaling and then context summing and scaling below.

*Definition 3.1 (Pair summing).* We can sum a pair of sensitivities and types  $(s, \tau) + (s', \tau) = (s + s', \tau)$ . Note that this is only well-defined if the types are syntactically equivalent.

344	Bounds $b, b_0, b_1$	$::= \epsilon \mid c \in \mathbb{B} \mid \mathbf{bop}_c(b_0, \dots, b_c)$
345	Types $\tau, \tau_0, \tau_1$	$::= \mathbf{unit} \mid \mathbf{num}_b \mid \tau_0 \times \tau_1 \mid \tau_0 \otimes \tau_1 \mid \tau_0 + \tau_1 \mid \tau_0 \multimap \tau_1 \mid !_s \tau \mid M_u \tau \mid \mathbf{bnd} \mid \forall \epsilon. \tau$
346	Types $\tau, \tau_0, \tau_1$	
347	Values $v, w$	$::= \langle \rangle \mid k \in \mathbf{num} \mid \langle x, y \rangle \mid (x, y) \mid \mathbf{in}_i x \mid \lambda x. e \mid \Lambda \epsilon. e$
348	Values $v, w$	
349	Terms $e, f, g$	$::= v \mid x \mid \mathbf{op} e \mid e f \mid e \{i\} \mid \pi_i e \mid \langle e, f \rangle \mid (e, f)$
350	Terms $e, f, g$	
351		$\mid \mathbf{let-pair} (x, y) = e \mathbf{in} f \mid \mathbf{let} x = e \mathbf{in} f$
352		$\mid \mathbf{in}_i e \mid \mathbf{case} e \mathbf{of} (\mathbf{in}_1 x. f_1 \mid \mathbf{in}_2 x. f_2)$
353		$\mid [e] \mid \mathbf{rnd} e \mid \mathbf{ret} e \mid \mathbf{factor} e$
354		$\mid \mathbf{let-bind} x = e \mathbf{in} f \mid \mathbf{let-cobind} x = e \mathbf{in} f$
355		
356	Evaluation contexts $C ::=$	$[.] \mid \mathbf{op}(C) \mid C e \mid v C \mid C \{b\} \mid \pi_i C$
357	Evaluation contexts $C ::=$	$\mid \mathbf{in}_i C \mid \langle C, e \rangle \mid \langle v, C \rangle \mid (C, e) \mid (v, C) \mid [C]$
358	Evaluation contexts $C ::=$	$\mid \mathbf{rnd} C \mid \mathbf{ret} C \mid \mathbf{factor} C$
359	Evaluation contexts $C ::=$	$\mid \mathbf{let} x = C \mathbf{in} f \mid \mathbf{let-pair} x = C \mathbf{in} f$
360	Evaluation contexts $C ::=$	$\mid \mathbf{let-bind} x = C \mathbf{in} f \mid \mathbf{let-cobind} x = C \mathbf{in} f$
361	Evaluation contexts $C ::=$	
362	Evaluation contexts $C ::=$	
363	Evaluation contexts $C ::=$	
364	Evaluation contexts $C ::=$	

Fig. 2. Types, values, and terms.  $\mathbf{op} \in \mathcal{O}$ .  $i \in \{1, 2\}$ .  $\mathbf{op}$  and  $\mathbf{bop}$  are syntactic metavariables representing functions of arbitrary arity.  $\mathbf{num}$  is a parameter in the programming language representing the set of numbers the language computes over.

*Definition 3.2 (Context summing).* A context  $\Theta = \Gamma + \Gamma'$  if and only if  $\forall x, \theta(x) = \Gamma(x) + \Gamma'(x)$  and is well defined.

*Definition 3.3 (Pair scaling).* We can scale a pair of sensitivities and types  $s' \cdot (s, \tau) = (s' \cdot s, \tau)$ .

*Definition 3.4 (Context scaling).* A context  $\Gamma' = s \cdot \Gamma$  if and only if  $\forall x, \Gamma'(x) = s \cdot \Gamma(x)$

The introduction and elimination rules for  $\otimes$ ,  $\times$ , and  $\multimap$  mirror Numerical Fuzz and correspond to the standard introduction and elimination rules in linear logic. We are also able to scale typing contexts through the  $(! I)$  rule. This is useful for instantiating functions with finitely scaled sensitivity.

The monadic grade upper-bounds the maximum possible round-off error possible. Most of the rules for controlling the grade, (Ret), (Rnd),  $(M_u E)$ , (Subsumption) are taken from Numerical Fuzz and are our core reasoning principles for reasoning about round-off error. Intuitively, (Ret) introduces no round-off error and (Rnd) introduces round-off error  $u$  corresponding to the unit round-off constant. Note that  $M_u E$  is the characteristic rule for **let-bind** that incorporates sensitivity and round-off information. Our language also introduces a new (Factor) rule for reasoning about monadic computations with round-off error, which allows for tighter error analyses. We detail the (Factor) rule and walk through an example in Section 5.

In this paper, we use the color *blue* to denote bound polymorphism. We have a *inj* function for mapping constants  $k$  into the corresponding interval that contains a single point. In the (Widen), (Bvar), (Bop),  $(\forall I)$ , and  $(\forall E)$  rules, we further extend Numerical Fuzz's type system with an abstract interpretation-style analysis by annotating  $\mathbf{num}_i$  types with a subscript bound with the grammar  $b$  shown in Figure 2. Our type system is parametric in  $\mathbb{B}$ , which is a set of bounds equipped with a partial order  $\leq$ . In Section ??, we define the  $\llbracket - \rrbracket$  relation interpreting bound expressions  $b$  to a subset of  $\mathbb{B}$ . We additionally extend Numerical Fuzz by adding expressions in more places (e.g. the

application rule) and a **factor** primitive to enable tighter error bounds. To support operations that have polymorphic bounds, we additionally modify our (Op) typing rule from Numerical Fuzz.

### 3.3 Operational Semantics

In Figure 4, we define the operational semantics rewrite relation  $\mapsto$  to map from closed terms in  $\Lambda_{\text{num}}^-$  to closed terms in  $\Lambda_{\text{num}}^-$ . Note that we only define our language over closed terms. Rather than stapling together and reasoning about two separate semantics, an ideal and approximate semantics, we combine our reasoning under one rewrite relation. This setup simplifies the structure of our logical relation and main soundness theorem easier to follow.

Accordingly, the interesting stepping rules that differ from Numerical Fuzz are the **rnd** rule, which steps a value to an ideal and rounded value, the **ret** rule, which steps a value to the diagonal map, and the **let-bind** stepping rule, which composes rounded computation. For our new **factor** primitive, we can view its operational semantics as a reassociating of the underlying ideal and approximate terms.

We also extend the Numerical Fuzz language to allow expressions in more places. We define evaluation contexts to separate the structural plumbing of the operational semantics from the more interesting portions of the operational semantics. Evaluation contexts and the characteristic *context rule* allow us to simplify reasoning about the rewrite relation, which we factor out into a separate lemma in our soundness proof.

### 3.4 Modular Interface

Numerical Fuzz is a modular family of programming languages parameterized by the appropriate  $\rho$ , constant parameter  $u$ , and the appropriate set of numeric computations  $\Sigma$ . It is the proof obligation for any language designer instantiating the language to demonstrate that these properties hold in order for our parameterized soundness theorems to follow.

*Definition 3.5 (Interface for instantiating Negative Fuzz.).* The interface for Negative Fuzz consists of  $\text{num}$ ,  $d_{\text{num}}$ ,  $\rho$ ,  $u$ ,  $\Sigma$ ,  $(\mathbb{B}, \leq)$ , and  $\Sigma_{\text{bnd}}$  such that the following properties hold:

- a) **The  $\text{num}$  and distance function  $d_{\text{num}}$  form a metric space.**
- b) **Property of  $u$  and rounding function  $\rho$ .** We assume that the  $\forall k \in \text{num}, d_{\text{num}}(\rho(k), k) \leq u$  where  $u$  is the grade in the **rnd** typing rule.
- c) **Property of  $\text{op} \in \Sigma$ : metric preservation.**<sup>2</sup>  
We can view  $\text{op}$  as a (possibly constant) metalevel function. We also assume that for every operation has the following shape with no free type variables  $\text{op} : \forall \epsilon_0, \epsilon_1. \tau_0 \multimap \tau_1 \in \Sigma$  where for the corresponding  $\text{op} \{b_0\} \{b_1\} v$  for all  $b_0, b_1 \in \mathbb{B}$  and for all  $v$  mapping bounds and syntactic values in  $(\tau_0 \rightarrow \tau_1)[b_0/\epsilon_0, b_1/\epsilon_1]$  is 1-sensitive as measured by  $\mathcal{SD}$ , defined below.
- d) **Partial order of bounds.** The set of bounds and binary relation  $(\mathbb{B}, \leq)$  form a partial order.
- e) **Property of  $\text{bop} \in \Sigma_{\text{bnd}}$ : closure over bounds ( $\mathbb{B}$ ).** We further need to assume that for every operation  $\text{bop}(b_0, \dots, b_n) : \Sigma_{\text{num}}$ , we have that if  $c_0, \dots, c_n \in \mathbb{B}$  then the corresponding  $\text{bop}(c_0, \dots, c_n) = c' : \text{bnd}$  holds.

The remainder of this subsection is dedicated to defining distances between terms, which is necessary to understand the metric preservation property our interface demands on all operations. We first define evaluation ( $\llbracket - \rrbracket$ ) of our bound expressions to  $\mathbb{B}$  which we call our *concrete bounds*. Each bound operation **bop** of arity  $n$  has a corresponding mathematical function  $\text{bop} : \mathbb{B}^n \rightarrow \mathbb{B}$ .

<sup>2</sup>Note that we could make this portion of the interface more general by defining it in terms of the logical relation in Section 3.10. We have opted to keep the interface minimal for presentational purposes.

$$\begin{array}{c}
\frac{}{\Delta \mid \Gamma \vdash \langle \rangle : \mathbf{unit}} \text{ (Unit)} \quad \frac{s \geq 1}{\Delta \mid \Gamma, x :_s \tau, \Theta \vdash x : \tau} \text{ (Var)} \quad \frac{\Delta \mid \Gamma, x :_1 \tau_0 \vdash e : \tau}{\Delta \mid \Gamma \vdash \lambda x. e : \tau_0 \multimap \tau} (\multimap \text{I}) \\
\frac{\Delta \mid \Gamma \vdash e : \tau_0 \multimap \tau \quad \Delta \mid \Theta \vdash f : \tau_0}{\Delta \mid \Gamma + \Theta \vdash ef : \tau} (\multimap \text{E}) \quad \frac{\Delta \mid \Gamma \vdash e : \tau_0 \quad \Delta \mid \Gamma \vdash f : \tau_1}{\Delta \mid \Gamma \vdash \langle e, f \rangle : \tau_0 \times \tau_1} (\times \text{I}) \\
\frac{\Delta \mid \Gamma \vdash e : \tau_1 \times \tau_2}{\Delta \mid \Gamma \vdash \pi_i e : \tau_i} (\times \text{E}) \quad \frac{\Delta \mid \Gamma \vdash e : \tau_0 \quad \Delta \mid \Theta \vdash f : \tau_1}{\Delta \mid \Gamma + \Theta \vdash (e, f) : \tau_0 \otimes \tau_1} (\otimes \text{I}) \\
\frac{\Delta \mid \Gamma \vdash e : \tau_0 \otimes \tau_1 \quad \Delta \mid \Theta, x :_s \tau_0, y :_s \tau_1 \vdash f : \tau}{\Delta \mid s * \Gamma + \Theta \vdash \mathbf{let-pair} (x, y) = e \mathbf{in} f : \tau} (\otimes \text{E}) \quad \frac{\Delta \mid \Gamma \vdash e : \tau_0}{\Delta \mid \Gamma \vdash \mathbf{in}_i e : \tau_0 + \tau_1} (+ \text{I}_i) \\
\frac{\Delta \mid \Gamma \vdash e : \tau_0 + \tau_1 \quad \Delta \mid \Theta, x :_s \tau_0 \vdash f_1 : \tau \quad \Delta \mid \Theta, x :_s \tau_1 \vdash f_2 : \tau \quad s > 0}{\Delta \mid s * \Gamma + \Theta \vdash \mathbf{case} e \mathbf{of} (\mathbf{in}_1 x. f_1 \mid \mathbf{in}_2 x. f_2) : \tau} (+ \text{E}) \\
\frac{\Delta \mid \Gamma \vdash e : !_s \tau_0 \quad \Delta \mid \Theta, x :_{t*s} \tau_0 \vdash f : \tau}{\Delta \mid t * \Gamma + \Theta \vdash \mathbf{let-cobind} x = e \mathbf{in} f : \tau} (! \text{E}) \quad \frac{\{\mathbf{op} : \forall \epsilon_0, \epsilon_1, \tau_0 \multimap \tau_1\} \in \Sigma}{\Delta \mid \Gamma \vdash \mathbf{op} : \forall \epsilon_0, \epsilon_1, \tau_0 \multimap \tau_1} (\text{Op}) \\
\frac{\Delta \mid \Gamma \vdash e : \tau_0 \quad \Delta \mid \Theta, x :_s \tau \vdash f : \tau}{\Delta \mid s * \Gamma + \Theta \vdash \mathbf{let} x = e \mathbf{in} f : \tau} (\text{Let}) \quad \frac{k \in \mathbf{num}}{\Delta \mid \Gamma \vdash k : \mathbf{num}_{inj(k)}} (\text{Const}) \\
\frac{\llbracket b_0 \rrbracket \leq \llbracket b_1 \rrbracket \quad \Delta \mid \Gamma \vdash e : \mathbf{num}_{b_0}}{\Delta \mid \Gamma \vdash e : \mathbf{num}_{b_1}} (\text{Widen}) \quad \frac{\Delta \mid \Gamma \vdash e : \tau}{\Delta \mid s * \Gamma \vdash [e] : !_s \tau} (! \text{I}) \\
\frac{\Delta \mid \Gamma \vdash e : M_q \tau \quad r \geq q}{\Delta \mid \Gamma \vdash e : M_r \tau} (\text{Subsumption}) \quad \frac{\Delta \mid \Gamma \vdash e : (M_q \tau_0) \times (M_r \tau_1)}{\Delta \mid \Gamma \vdash \mathbf{factor} e : M_{\max(q,r)}(\tau_0 \times \tau_1)} (\text{Factor}) \\
\frac{\Delta \mid \Gamma \vdash e : \tau}{\Delta \mid \Gamma \vdash \mathbf{ret} e : M_0 \tau} (\text{Ret}) \quad \frac{\Delta \mid \Gamma \vdash e : \mathbf{num}_i}{\Delta \mid \Gamma \vdash \mathbf{rnd} e : M_u \mathbf{num}_i} (\text{Rnd}) \\
\frac{\Delta \mid \Gamma \vdash e : M_r \tau_0 \quad \Delta \mid \Theta, x :_s \tau_0 \vdash f : M_q \tau}{\Delta \mid s * \Gamma + \Theta \vdash \mathbf{let-bind} x = e \mathbf{in} f : M_{s*r+q} \tau} (M_u \text{E}) \\
\frac{}{\Delta, \epsilon : \mathbf{bnd} \vdash \epsilon : \mathbf{bnd}} (\text{Bvar}) \quad \frac{\Delta \vdash b_0 : \mathbf{bnd} \quad \dots \quad \Delta \vdash b_n : \mathbf{bnd} \quad \mathbf{bop} \in \Sigma_{\mathbf{bnd}}}{\Delta \vdash \mathbf{bop}(b_0, \dots, b_n) : \mathbf{bnd}} (\text{Bop}) \\
\frac{\Delta, \epsilon : \mathbf{bnd} \mid \Gamma \vdash e : \tau \quad \epsilon \notin \text{FTV}(\Gamma)}{\Delta \mid \Gamma \vdash \Lambda \epsilon. e : \forall \epsilon. \tau} (\forall \text{-I}) \quad \frac{\Delta \mid \Gamma \vdash e : \forall \epsilon. \tau \quad \Delta \vdash : \mathbf{bnd}}{\Delta \mid \Gamma \vdash e \{i\} : \tau[i/\epsilon]} (\forall \text{-E})
\end{array}$$

Fig. 3. Typing rules for  $\Lambda_{\mathbf{num}}^-$ , with  $s, t, q, r, u \in \mathbb{R}^{\geq 0} \cup \{\infty\}$  and for  $i \in \{1, 2\}$  where  $u$  is a fixed constant parameter (see Definition 3.5 for details on picking an adequate constant).  $\text{FTV}(\Gamma)$  refers to all the free type variables (e.g.  $\epsilon_0, \epsilon_1$ ) in  $\Gamma$ .

$$\begin{array}{ll}
\text{op } \{b_0\} \{b_1\} v \mapsto \text{op } \{b_0\} \{b_1\} v & \text{let } x = v \text{ in } e \mapsto e[v/x] \\
\pi_i \langle v_1, v_2 \rangle \mapsto v_i & \text{let-pair } (x, y) = (v, w) \text{ in } e \mapsto e[v/x][w/y] \\
(\lambda x. e) v \mapsto e[v/x] & \text{let-cobind } x = [v] \text{ in } e \mapsto e[v/x] \\
\text{rnd } k \mapsto (k, \rho(k)) & \text{ret } v \mapsto (v, v) \\
\text{case } (\text{in}_i v) \text{ of } (\text{in}_1 x. e_1 \mid \text{in}_2 x. e_2) \mapsto e_i[v/x] & \\
\text{factor } ((v_1, v_2), (v_3, v_4)) \mapsto ((v_1, v_3), (v_2, v_4)) & \\
\text{bop}(c_0, \dots, c_n) \mapsto \text{bop}(c_0, \dots, c_n) & \\
\Lambda \epsilon. e \ c \mapsto e[c/\epsilon] & \\
\\ 
\frac{f[v_1/x] \mapsto^* (v_3, v_4) \quad f[v_2/x] \mapsto^* (v_5, v_6)}{\text{let-bind } x = (v_1, v_2) \text{ in } f \mapsto (v_3, v_6)} & \\
\frac{e \mapsto e'}{C[e] \mapsto C[e']} \text{ (Context Rule)} & 
\end{array}$$

Fig. 4. Substitution-style evaluation rules for  $\Lambda_{\text{num}}^-$ . Parameterized for  $i \in \{1, 2\}$ .  $\text{op}$  is a higher-order metavar. When bolded  $\text{op}$  refers to the syntax and when italicized  $\text{op}$  refers to the corresponding function on syntactic values (it may be a constant function).

We can now define closed *bound expressions*, which we will use to define the evaluation function used in the definition of our logical relations.

*Definition 3.6 (Closed bound expressions).* A bound expression  $b$  is a closed bound expression iff either:

- $b = c \in \mathbb{B}$ . In other words,  $b$  is a constant.
- $b = \text{bop}(b_0, \dots, b_1)$  where  $b_0, \dots, b_1$  are all closed bound expressions.  $b$  is made up of constants.

*Definition 3.7 (Evaluation of closed bound expressions).* For an bound expression  $b$ ,

$$\begin{aligned}
\llbracket c \rrbracket &\triangleq c \\
\llbracket \text{bop}(b_0, \dots, b_n) \rrbracket &\triangleq \text{bop}(b_0, \dots, b_n)
\end{aligned} \tag{8}$$

We proceed to define distances between bounds and expressions. Our syntactic definition for distance,  $d_\tau$  and the distance between syntactic values  $\mathcal{SD}_\tau$  (for Syntactic Distance) and  $\mathcal{SDV}_\tau$  (for Syntactic Distance for Values) are closely related. Some care is needed to ensure that the relation is well-founded. We define our distance over syntactic values as follows:

*Definition 3.8 (Distance between closed syntactic terms).*

$$\begin{aligned}
\mathcal{SD}_\tau(e_0, e_1) &\triangleq \mathcal{SDV}_\tau(v_0, v_1) && \text{if } e_0 \mapsto^* v_0 \text{ and } e_1 \mapsto^* v_1 \\
\mathcal{SD}_\tau(e_0, e_1) &\triangleq \infty && \text{otherwise} \\
\mathcal{SDV}_{\text{unit}}(v, w) &\triangleq 0 && \text{for } v, w = \langle \rangle \\
\mathcal{SDV}_{\text{num}_b}(c_0, c_1) &\triangleq d_{\text{num}} && \text{for } c_0, c_1 \in \llbracket b \rrbracket \\
\mathcal{SDV}_{\tau_0 \times \tau_1}((v_0, v_1), (w_0, w_1)) &\triangleq \max(\mathcal{SDV}_{\tau_0}(v_0, w_0), \mathcal{SDV}_{\tau_0}(v_1, w_1)) \\
\mathcal{SDV}_{\tau_0 \otimes \tau_1}((v_0, v_1), (w_0, w_1)) &\triangleq \mathcal{SDV}_{\tau_0}(v_0, w_0) + \mathcal{SDV}_{\tau_0}(v_1, w_1) \\
\mathcal{SDV}_{\tau_0 + \tau_1}(\text{in}_i v, \text{in}_i w) &\triangleq \mathcal{SDV}_{\tau_i}(v, w) \\
\mathcal{SDV}_{\tau_0 \multimap \tau_1}(v_0, v_1) &\triangleq \sup_{w \in \mathcal{V}_{\mathcal{R}_{\tau_0}}} \mathcal{SD}_{\tau_1}(v_0 \ w, v_1 \ w) \\
\mathcal{SDV}_{!s\tau}([v], [w]) &\triangleq s \cdot \mathcal{SDV}_\tau(v, w) \\
\mathcal{SDV}_{M_q \tau}((v_0, v_1), (w_0, w_1)) &\triangleq \mathcal{SDV}_\tau(v_0, w_0) \\
\mathcal{SDV}_{\text{bnd}}(b_0, b_1) &\triangleq 0 && \text{for } \llbracket b_0 \rrbracket = \llbracket b_1 \rrbracket \\
\mathcal{SDV}_{\text{bnd}}(b_0, b_1) &\triangleq \infty && \text{otherwise} \\
\mathcal{SDV}_{\forall \epsilon, \tau}(v_0, v_1) &\triangleq \sup_{c \in \mathbb{B}} \mathcal{SD}_{\tau[w/\epsilon]}(v_0 \ \{c\}, v_1 \ \{c\}) \\
\mathcal{SDV}_\tau(v, w) &\triangleq \infty && \text{otherwise}
\end{aligned}$$

(9)

*Definition 3.9 (Distance between expressions).* For any two closed expressions  $e_0, e_1$  falling in same type relation  $e_0, e_1 \in R_\tau$  (where  $\tau$  is a closed type), we can write  $e_0 \sim_r e_1 : \tau$  where  $\mathcal{SD}_\tau(e_0, e_1) \leq r$ .

### 3.5 Type Soundness

In our language, we only reason about closed types and terms. Let  $CV(\tau)$  be the closed values of type  $\tau$  and  $CE(\tau)$  be the closed expressions of type  $\tau$ . Then we can define a unary logical relation over types which capture the core information needed to prove our error soundness theorem (Metric Preservation, Theorem 10).

Definition 3.10 (Logical relation).

$$\begin{aligned}
\mathcal{R}_\tau &\triangleq \{e \mid e \in CE(\tau) \text{ and } \exists v \in CV(\tau) \text{ s.t. } e \mapsto^* v \text{ and } v \in \mathcal{VR}_\tau\} \\
\mathcal{VR}_{\text{unit}} &\triangleq \{\langle \rangle\} \\
\mathcal{VR}_{\text{num}_b} &\triangleq [\![b]\!] \\
\mathcal{VR}_{\tau_0 \times \tau_1} &\triangleq \{\langle v, w \rangle \mid v \in \mathcal{R}_{\tau_0} \text{ and } w \in \mathcal{R}_{\tau_1}\} \\
\mathcal{VR}_{\tau_0 \otimes \tau_1} &\triangleq \{\langle v, w \rangle \mid v \in \mathcal{R}_{\tau_0} \text{ and } w \in \mathcal{R}_{\tau_1}\} \\
\mathcal{VR}_{\tau_0 + \tau_1} &\triangleq \{\text{inl } v \mid v \in \mathcal{R}_{\tau_0}\} \cup \{\text{inr } v \mid v \in \mathcal{R}_{\tau_1}\} \\
\mathcal{VR}_{\tau_0 \multimap \tau_1} &\triangleq \{\lambda x.e \mid \forall w_0, w_1 \in \mathcal{VR}_{\tau_0}, \\
&\quad (\lambda x.e) w_0, (\lambda x.e) w_1 \in \mathcal{R}_{\tau_1} \text{ and } \mathcal{SD}_{\tau_1}((\lambda x.e) w_0, (\lambda x.e) w_1) \leq \mathcal{SD}_{\tau_0}(w_0, w_1)\} \\
\mathcal{VR}_{!s, \tau} &\triangleq \{[v] \mid v \in \mathcal{R}_\tau\} \\
\mathcal{VR}_{M_q \tau} &\triangleq \{\langle v, w \rangle \mid v, w \in \mathcal{R}_\tau \text{ and } \mathcal{SDV}_\tau(v, w) \leq q\} \\
\mathcal{VR}_{\forall \epsilon, \tau} &\triangleq \{\Lambda \epsilon.v \mid \forall c \in \mathbb{B}, v\{c\} \in \mathcal{R}_{\tau[c/\epsilon]}\}
\end{aligned} \tag{10}$$

LEMMA 3.11 ( $\mathcal{SD}$  IS A METRIC).  $\mathcal{SD}$  forms a metric over our syntactic terms; in particular it satisfies:

- (1) Distance from any point to itself is zero:  $\forall x, \mathcal{SD}(x, x) = 0$ .
- (2) Positivity:  $\forall x, y, \mathcal{SD}(x, y) \geq 0$ .
- (3) Symmetry:  $\forall x, y, \mathcal{SD}(x, y) = \mathcal{SD}(y, x)$ .
- (4) Triangle inequality:  $\forall x, y, z, \mathcal{SD}(x, z) \leq \mathcal{SD}(x, y) + \mathcal{SD}(y, z)$ .

PROOF. The properties holds for the base cases of  $\mathcal{SDV}$  and follow for the remaining cases by our inductive hypothesis. Since our operational semantics is deterministic, the properties follow for  $\mathcal{SD}$ .  $\square$

LEMMA 3.12 ( $\mathcal{SD}$  IS PRESERVED UNDER STEPPING). If  $e_0 \mapsto e'_0$ , then for any  $e_1$ ,  $\mathcal{SD}(e_0, e_1) = \mathcal{SD}(e'_0, e_1)$ .

PROOF. Holds by inspection of the definition of  $\mathcal{SD}$ .  $\square$

Note that by metric symmetry,  $\mathcal{SD}$  is preserved under stepping on both sides.

Following Fuzz, we prove the following syntactic properties relating our type system and operational semantics. These syntactic properties will be useful for reasoning about type inference in Section 6.

LEMMA 3.13 (ADMISSIBILITY OF WEAKENING). If  $\Delta \mid \Gamma \vdash e : \tau$ , then  $\Delta + \Delta' \mid \Gamma + \Sigma \vdash e : \tau$ .

PROOF. We induct over our typing derivation. For each case, we can ignore arbitrary variables in our context (the type system is affine). In particular, for the  $!E$ ,  $\otimes E$ ,  $+ E$ ,  $M_u E$ , and  $\otimes I$  rules, the enviroment must be split into a  $\Gamma$  (which can in some cases be scaled) and  $\Delta$  (which does not ever get scaled); for each of these cases we choose to push unused variables into  $\Delta$ .  $\square$

LEMMA 3.14 (ADMISSIBILITY OF WEAKENING (ON BOUND CONTEXTS)). If  $\Delta \vdash : \text{bnd}$  and  $\Delta \subseteq \Delta'$ , then  $\Delta' \vdash : \text{bnd}$ .

PROOF.  $\Delta$  tracks the free type variables in  $i$ . We induct over our typing derivation. The base cases hold. For the inductive cases, we induct over the size of  $\Delta' - \Delta$ . Our proof is complete.  $\square$

We now define vectors of expressions, and substitutions which will also be useful later for stating and proving our main type soundness proof (Metric Preservation, Theorem ??). We proceed to incrementally build up to the definition of distance between substitutions below.

*Definition 3.15 (Distance between expression vectors.).* We write vectors of expressions  $\sigma, \sigma'$  are  $\gamma$  apart for a given typing context  $\Delta \mid \Gamma$  like so  $\sigma \sim_\gamma \sigma' : \Delta \mid \Gamma$  if and only if  $\gamma = r_0, r_1, \dots$  where  $\sigma = \sigma_\Delta, e_0, \dots, e_n$  and  $\sigma' = \sigma_\Delta, e'_0, \dots, e'_n$  such that:

$$e_0 \sim_{r_0} e'_0 : \tau_0 [\sigma_\Delta], \dots, e_n \sim_{r_n} e'_n : \tau_n [\sigma_\Delta]$$

where  $\sigma_\Delta$  is a bound vector with no free variables. Note that the distance between expression vectors are undefined when the bound vector  $\sigma_\Delta$  differs between substitutions.

*Definition 3.16 (Environment compatibility.).* We also say that a substitution vector

$$\sigma_\Delta, [e_0/x_0, \dots, e_n/x_n]$$

is *compatible* with a typing context  $x_0 : \tau_0, \dots, x_n : \tau_n$  if each term  $e_0 \in \mathcal{R}_{\tau_0 [\sigma_\Delta]}, \dots, e_n \in \mathcal{R}_{\tau_n [\sigma_\Delta]}$  where all types are closed by the bound vector  $\sigma_\Delta$ .

*Definition 3.17 (Dot product of distance vectors.).* Our definition for the dot product of a distance vector  $\gamma = r_0, \dots, r_n$  with respect to a context  $\Gamma$  with sensitivities  $s_0, \dots, s_n$  is the same as  $\text{Fuzz} [?]$ :  $\gamma \cdot \Gamma = \sum_{i=1}^n r_i s_i$ .

In the remainder of the paper, we'll treat and represent our distance vector  $\gamma$  as a lookup function and assume that there is an implicit fixed ordering on the variables.

*Definition 3.18 (Distance vector lookup.).* We write, for a distance vector  $\gamma$  and variable  $x$ ,  $\gamma(x)$  for the lookup of the distance of variable  $x$  in  $\gamma$ . If the variable  $x$  is not in the domain,  $\gamma(x) = 0$  by default.

We have now finished defining all the components necessary to state our logical relation. Now that we can talk about substitutions, we follow  $\text{Fuzz}$  and prove a  $r$ -sensitive substitution lemma.

**LEMMA 3.19 ( $r$ -SENSITIVE SUBSTITUTION).** *Let  $\Delta_0 \mid \Gamma \vdash e : \tau$  and  $\Delta_1 \mid \Theta, x : \tau \vdash e' : \tau'$  and  $r \neq 0$  and  $\Delta_0 \cap \Delta_1 = \emptyset$ , then  $\Delta_0 + \Delta_1 \mid r \cdot \Gamma + \Theta \vdash e' [e/x] : \tau'$ .*

**PROOF.** We first induct over the height of the final typing derivation with substitution. Note that the *Bvar* and *Bop* rules are rules for a different typing judgement from our main judgement  $\vdash$ . The Unit, Const, and Op base cases are immediate. We detail the remaining cases here:

**Var.** In the case that  $s = 1$ , holds immediately. Otherwise, we apply weakening.

**Inductive cases with possible variable capture**  $\multimap \mathbf{I}, \otimes \mathbf{E}, + \mathbf{E}, ! \mathbf{E}, \text{Let}, M_u \mathbf{E}$  We case: if  $x$  is immediately captured by the expression (e.g.  $\lambda$  for  $\multimap \mathbf{I}$  or  $x$  and  $y$  for the **let-pair** expression, etc.), we can apply our inductive hypothesis. Otherwise, it holds immediately.

**Inductive case:**  $\forall - \mathbf{I}$ . Holds by our inductive hypothesis and observing that the side condition necessary to apply the  $\forall \mathbf{I}$  rule is satisfied by the  $\Delta_0 \cap \Delta_1 = \emptyset$  condition.

**Remaining inductive cases.** The remaining cases ( $\multimap \mathbf{E}, \times \mathbf{I}, \times \mathbf{E}, \otimes \mathbf{I}, + \mathbf{I}_i, \text{Widen}, ! \mathbf{I}, \text{Subsumption}, \text{Factor}, \text{Ret}, \text{Rnd}, \forall \mathbf{E}$ ) follow immediately by application of the inductive hypothesis.

□

**LEMMA 3.20 (SUBSTITUTION DECOMPOSITION).** *For substitutions  $\sigma \sim_\alpha \sigma' : \Delta_0 + \Delta_1 \mid \Gamma + \Theta$ , there exists  $\sigma_\gamma \sim_\alpha \sigma'_\gamma : \Delta_0 \mid \Gamma$  and  $\sigma_\theta \sim_\alpha \sigma'_\theta : \Delta_1 \mid \Theta$ .*

**PROOF.** Follows by induction over the length of the relation  $\sigma \sim_\alpha \sigma'$ .

□

The below lemma is useful towards proving metric preservation:

LEMMA 3.21 (METRIC PRESERVATION UNDER CONTEXT EVALUATION STEPPING). *For a well-typed  $\Delta \mid \Gamma \vdash C[e] : \tau$  and substitutions  $\sigma, \sigma'$  such that  $\sigma \sim_Y \sigma' : \Delta \mid \Gamma$  and  $e \sigma \mapsto^* v$  and  $e \sigma' \mapsto^* v'$ . If  $C[v] \sigma \sim_{Y \cdot (\Delta \mid \Gamma)} C[v'] \sigma' : \tau$  then  $C[e] \sigma \sim_{Y \cdot (\Delta \mid \Gamma)} C[e] \sigma' : \tau$ .*

PROOF. By inspection of our stepping relation, we can tell that stepping is deterministic. Further, since we only define our rewrite relation over closed terms, we know that  $e \sigma, e \sigma'$  is closed and therefore constant under all substitutions. So, by the definition of  $\mathcal{R}, \mathcal{SD}$ , and context stepping, we know that both

$$C[e \sigma] \sigma, C[e \sigma'] \sigma' \in \mathcal{R}_\tau$$

and that

$$\mathcal{SD}_\tau(C[e] \sigma, C[e] \sigma') = \mathcal{SD}_\tau(C[e \sigma] \sigma, C[e \sigma'] \sigma') = \mathcal{SD}_\tau(C[v] \sigma, C[v'] \sigma') = Y \cdot (\Delta \mid \Gamma)$$

respectively and therefore  $C[e] \sigma \sim_{Y \cdot (\Delta \mid \Gamma)} C[e] \sigma'$  holds.  $\square$

Finally, we can prove our main error soundness theorem.

THEOREM 3.22 (METRIC PRESERVATION). *For any  $\Delta \mid \Gamma \vdash e : \tau$  and substitutions  $\sigma, \sigma'$  such that  $\sigma \sim_Y \sigma' : \Delta \mid \Gamma$ , then  $e \sigma \sim_{Y \cdot (\Delta \mid \Gamma)} e \sigma' : \tau$ .*

## 4 A PAIRED MODEL OF NEGATIVE FUZZ

Numerical Fuzz's error analysis relies on bounding Lipschitz function sensitivity. However, subtraction, and by extension addition with negative numbers, has unbounded Lipschitz sensitivity over the reals. This poses a challenge to a compositional type-based analysis of programs involving subtraction and negative numbers.

To avoid this problem, we translate our semantics to a *paired representation* where we associate for each real  $r \in \mathbb{R}$  a lattice of triples  $(r, a, b) \in \mathbb{R} \times \mathbb{R}^+ \times \mathbb{R}^+ \triangleq \mathbb{P} = \text{num}$  under the  $\leq$  relation over the first component such that  $r = a - b$ . Accordingly, we equip our set  $\mathbb{P}$  with a distance function  $d_{\text{num}}((r, a, b)(r', a', b'))$  defined as the maximum distance over the paired components (Definition 2.1). Instead of reasoning over the real component, we instantiate our graded type system to track distances and error over the always-growing paired components. We then instantiate our automatic typed-based bounds analysis and apply our main error theorem to translate our error bound back over the unpaired component.

The remainder of this section is organized as follows. In Section 4.1, we instantiate Negative Fuzz's modular interface (Definition 3.5) with our *paired representation*. In particular, we define and type our supported operations including subtraction under the paired representation. We describe our bounds analysis setup and provide intuition for our choice of domain. In Section 4.2, we use the results of our automatic type-based bounds analysis to translate the monadic grade over the paired components into useful bounds over the underlying unpaired real. We also present our main error theorem over the unpaired real. We defer introducing our approach to automatic type inference to Section 6.

### 4.1 Instantiating the modular interface

To soundly use the paired representation, we must instantiate the modular interface  $(\text{num}, d_{\text{num}}, \rho, u, \Sigma, (\mathbb{B}, \leq), \Sigma_{\text{bnd}})$  defined in Definition 3.5 and show that it satisfies the properties specified. In the subsequent subsections, we instantiate the interface with the appropriate rounding functions and round-off parameters (Section 4.1.1), supported arithmetic operations (Section 4.1.2), bounds domain and bounds operations (Section 4.1.3), and finally prove that our instantiation adheres to the spec demanded by Negative Fuzz (Sections 4.1.4-4.1.8).

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 ( $u$ )
Round towards $+\infty$	$\min\{y \in \mathbb{F} \mid y \geq x\}$	$\rho_{RU}(x)$	$\beta^{1-p}$
Round towards $-\infty$	$\max\{y \in \mathbb{F} \mid y \leq x\}$	$\rho_{RD}(x)$	$\beta^{1-p}$
Round towards 0	$\rho_{RU}(x)$ if $x < 0$ , otherwise $\rho_{RD}(x)$	$\rho_{RZ}(x)$	$\beta^{1-p}$
Round towards nearest <sup>3</sup>	$\{y \in \mathbb{F} \mid \forall z \in \mathbb{F},  x - y  \leq  x - z \}$	$\rho_{RN}(x)$	$\frac{1}{2}\beta^{1-p}$

**4.1.1 Defining the rounding function and round-off parameters for the paired representation.** We set  $num = \mathbb{P}$  equipped with  $\leq$  defined above. We also set  $\rho = \rho_{\mathbb{P}}$  (Definition ??) to “simulate” the round-off error on the unpaired component. Our choice of  $u$  depends on the format and rounding mode used. It is sound to use same corresponding choice of  $u$  representing the unit-roundoff constant as Numerical Fuzz (Lemma ??); we reproduce the corresponding choices from Numerical Fuzz broken out by floating-point format and rounding mode used in Table 1 and Table 2 respectively. We define our operations  $\Sigma$ , bound domain ( $\mathbb{B}, \leq$ ), and bound operations  $\Sigma_{bnd}$  below.

**4.1.2 Defining our operations over the paired representation.** We define  $\Sigma = \{\mathbf{add}, \mathbf{sub}, \mathbf{mul}\}$ , which compute and are typed over our paired representation in the following way:

$$\begin{aligned}
&\mathbf{add} : \forall \epsilon_1, \epsilon_2, \mathbf{num}_{\epsilon_1} \times \mathbf{num}_{\epsilon_2} \multimap \mathbf{num}_{\epsilon_1 + \epsilon_2} \\
&\mathbf{add} \{ \epsilon_1 \} \{ \epsilon_2 \} ((\langle r, a, b \rangle, \langle r', a', b' \rangle)) \mapsto (r + r', a + a', b + b') \\
&\mathbf{sub} : \forall \epsilon_1, \epsilon_2, \mathbf{num}_{\epsilon_1} \times \mathbf{num}_{\epsilon_2} \multimap \mathbf{num}_{\epsilon_1 - \epsilon_2} \\
&\mathbf{sub} \{ \epsilon_1 \} \{ \epsilon_2 \} ((\langle r, a, b \rangle, \langle r', a', b' \rangle)) \mapsto (r - r', a + b', b + a') \\
&\mathbf{mul} : \forall \epsilon_1, \epsilon_2, \mathbf{num}_{\epsilon_1} \times \mathbf{num}_{\epsilon_2} \multimap \mathbf{num}_{\epsilon_1 \cdot \epsilon_2} \\
&\mathbf{mul} \{ \epsilon_1 \} \{ \epsilon_2 \} (((\langle r, a, b \rangle, \langle r', a', b' \rangle))) \mapsto (r * r', a * b + b * b', b * a' + a * b')
\end{aligned} \tag{11}$$

**4.1.3 Defining our bounds domain and bound operations.** Our bounds analysis is inspired by interval analysis. For each component in our triple, we track the lower bound and upper bound. For example, for a pair  $(1, 2, 3)$  we might track that  $r^\downarrow$  (pronounced “r lower bound”) and  $r^\uparrow$  (pronounced “r upper bound”) and  $a^\downarrow, a^\uparrow, b^\downarrow, b^\uparrow$  each bound their respective components. To improve the precision of our bounds analysis, we observe that frequently only one of the paired components is greater than zero. In fact, an inspection of our supported operations shows that  $a > 0 \wedge b > 0$  can only be true when either: (1) the program contains subtraction, or (2) the program contains negative numbers. Further, we observe that when the  $(a > 0 \wedge b > 0)$  condition holds, the bound drastically tightens.

Accordingly, for a concrete tuple  $(r, a, b) \in \mathbb{P}$  we set our bound domain  $\mathbb{B}$  to track seven things:  $((r^\downarrow, r^\uparrow), (a^\downarrow, a^\uparrow), (b^\downarrow, b^\uparrow), d)$  under the following constraints:

- (1) Lower and upper bounds on  $r$ , with the following constraints  $r^\downarrow \leq r \leq r^\uparrow$
- (2) Lower and upper bounds on  $a$ , with the following constraints  $a^\downarrow \leq a \leq a^\uparrow$
- (3) Lower and upper bounds on  $b$ , with the following constraints  $b^\downarrow \leq b \leq b^\uparrow$

(4) A boolean  $d$  tracking whether the condition  $a > 0 \wedge b > 0$  is true.

To define  $\leq$  over our bounds domain, we first define a mapping of bounds to sets:

*Definition 4.1 (Mapping of bounds to sets.).* Our mapping from bounds to sets **toSet** :  $\mathbb{B} \rightarrow \mathbf{Set}$  is defined as follows:

$$\begin{aligned} \mathbf{toSet}((r^\downarrow, r^\uparrow), (a^\downarrow, a^\uparrow), (b^\downarrow, b^\uparrow), d) = \\ \{(r, a, b) \mid (r^\downarrow \leq r \leq r^\uparrow) \wedge (a^\downarrow \leq a \leq a^\uparrow) \wedge (b^\downarrow \leq b \leq b^\uparrow) \wedge ((a > 0 \wedge b > 0) = d) \} \quad (12) \\ \forall (r, a, b) \in \mathbb{R} \times \mathbb{R}^+ \times \mathbb{R}^+ \end{aligned}$$

Then, our  $\leq$  relation over  $\mathbb{B}$  is simply the  $\subseteq$  relation over the **toSet** interpretation of each bound:

*Definition 4.2 ( $\leq$  over  $\mathbb{B}$ .).*  $b_0 \leq b_1 \iff \mathbf{toSet}b_0 \subseteq \mathbf{toSet}b_1$

Finally, we define our bound operations  $\Sigma_{bnd} = \{+, -, \cdot\}$  mapping  $\mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$ . Each bound operation corresponds to the primitive operation in the surface syntax of the language. The correctness of the definition of each bound operation is ensured by the interface requirement that that each primitive operation's corresponding type obeys metric preservation.

$$\begin{aligned} & ((r^\downarrow, r^\uparrow), (a^\downarrow, a^\uparrow), (b^\downarrow, b^\uparrow), d) + ((r'^\downarrow, r'^\uparrow), (a'^\downarrow, a'^\uparrow), (b'^\downarrow, b'^\uparrow), d') = \\ & ((r_{add}^\downarrow, r_{add}^\uparrow), (a^\downarrow + a'^\downarrow, a^\uparrow + a'^\uparrow), (b^\downarrow + b'^\downarrow, b^\uparrow + b'^\uparrow), (d \wedge d' \wedge ((0 \leq r^\downarrow \wedge 0 \leq r'^\downarrow) \vee (r^\uparrow \leq 0 \wedge r'^\uparrow \leq 0)))) \\ & ((r^\downarrow, r^\uparrow), (a^\downarrow, a^\uparrow), (b^\downarrow, b^\uparrow), d) - ((r'^\downarrow, r'^\uparrow), (a'^\downarrow, a'^\uparrow), (b'^\downarrow, b'^\uparrow), d') = \\ & ((r_{sub}^\downarrow, r_{sub}^\uparrow), (a^\downarrow + b'^\downarrow, a^\uparrow + b'^\uparrow), (b^\downarrow + a'^\downarrow, b^\uparrow + a'^\uparrow), (d \wedge d' \wedge ((0 \leq r^\downarrow \wedge r'^\uparrow \leq 0) \vee (r^\uparrow \leq 0 \wedge 0 \leq r'^\downarrow)))) \\ & ((r^\downarrow, a^\downarrow, b^\downarrow), (r^\uparrow, a^\uparrow, b^\uparrow)) \cdot ((r'^\downarrow, a'^\downarrow, b'^\downarrow), (r'^\uparrow, a'^\uparrow, b'^\uparrow)) = \\ & ((r_{mul}^\downarrow, r_{mul}^\uparrow), (\min(a^\downarrow a'^\downarrow, b^\downarrow b'^\downarrow), \min(a^\uparrow a'^\uparrow, b^\uparrow b'^\uparrow)), (\max(a^\downarrow b'^\downarrow, b^\downarrow a'^\downarrow), \max(a^\uparrow b'^\uparrow, b^\uparrow a'^\uparrow), d \wedge d')) \\ & \quad \text{if } d \wedge d' \\ & ((r^\downarrow, a^\downarrow, b^\downarrow), (r^\uparrow, a^\uparrow, b^\uparrow)) \cdot ((r'^\downarrow, a'^\downarrow, b'^\downarrow), (r'^\uparrow, a'^\uparrow, b'^\uparrow)) = \\ & ((r_{mul}^\downarrow, r_{mul}^\uparrow), (a^\downarrow a'^\downarrow + b^\downarrow b'^\downarrow, a^\uparrow a'^\uparrow + b^\uparrow b'^\uparrow), (a^\downarrow b'^\downarrow + b^\downarrow a'^\downarrow, a^\uparrow b'^\uparrow + b^\uparrow a'^\uparrow), d \wedge d') \\ & \quad \text{otherwise} \end{aligned} \quad (13)$$

where

$$\begin{aligned} r_{add}^\downarrow &= \min(\{x + y \mid \forall x \in [r^\downarrow, r^\uparrow] \forall y \in [r'^\downarrow, r'^\uparrow]\}) \\ r_{add}^\uparrow &= \max(\{x + y \mid \forall x \in [r^\downarrow, r^\uparrow] \forall y \in [r'^\downarrow, r'^\uparrow]\}) \\ r_{sub}^\downarrow &= \min(\{x - y \mid \forall x \in [r^\downarrow, r^\uparrow] \forall y \in [r'^\downarrow, r'^\uparrow]\}) \\ r_{sub}^\uparrow &= \max(\{x - y \mid \forall x \in [r^\downarrow, r^\uparrow] \forall y \in [r'^\downarrow, r'^\uparrow]\}) \\ r_{mul}^\downarrow &= \min(\{x \cdot y \mid \forall x \in [r^\downarrow, r^\uparrow] \forall y \in [r'^\downarrow, r'^\uparrow]\}) \\ r_{mul}^\uparrow &= \max(\{x \cdot y \mid \forall x \in [r^\downarrow, r^\uparrow] \forall y \in [r'^\downarrow, r'^\uparrow]\}) \end{aligned}$$

We now prove that our instantiation adheres to the properties demanded by the interface spec of Negative Fuzz. By stapling together the following lemmas, the interface  $(\mathbb{P}, d_{\mathbb{P}}, \rho_{\mathbb{P}}, u, \Sigma, (\mathbb{B}, \leq), \Sigma_{\text{num}})$  satisfies the spec for instantiating Negative Fuzz specified in Definition 3.5.

#### 4.1.4 Property A: The num and distance function $d_{\text{num}}$ form a metric space.

LEMMA 4.3 (METRIC SPACE).  $\mathbb{P}$  forms a metric space under  $d_{\mathbb{P}}$ .

PROOF. Symmetry and reflexivity hold trivially. Triangle inequality holds by unfolding and application of triangle inequality with respect to  $d_{\mathbb{R}}$ .  $\square$

Our operations and bounds operate over the paired representation  $\mathbb{P}$ . In this manner, subtraction and negative numbers can be encoded using bounded-sensitivity operations over the non-negative reals, avoiding the problem of infinite sensitivity.

#### 4.1.5 Property B: Property of $u$ and rounding function $\rho$ .

To show a correspondance with the constants for  $u$  given in Numerical Fuzz and Negative Fuzz, we show that it is sound to use the same round-off error bound  $u$  associated with  $\rho_{\mathbb{R}}$  for  $\rho_{\mathbb{P}}$ . In other words, we wish to show that the translated rounding function can share the same round-off constant as the grade used in Numerical Fuzz over an unpaired representation.

Given a rounding function  $\rho_{\mathbb{R}}$  over the real with up to  $u$  round-off error, recall that we define a "simulated" rounding function  $\rho_{\mathbb{P}}$  over the paired representation (Definition ??) which satisfies the constraint that  $r = a - b$  at all times. We need to show that our round-off constants  $u$  still work for our new paired representation and paired rounding function  $\rho_{\mathbb{P}}$ .

LEMMA 4.4 (PROPERTY OF ROUNDING AND CONSTANT PARAMETER  $u$ ). We wish to show that given a unit round-off bound  $u$  and  $\rho_{\mathbb{R}}$  such that  $\forall r \in \mathbb{R}, d_{\mathbb{R}}(\rho(r), r) \leq u$ , then  $\forall e \in \mathcal{R}_{\text{num}}, \mathcal{SD}_{\text{num}}(\rho_{\mathbb{P}}(e), e) \leq u$ .

PROOF.  $(\rho_{\mathbb{P}}((r, a, b)), (r, a, b)) = (\rho_{\mathbb{R}}(r), a \cdot \frac{\rho_{\mathbb{R}}(r)}{r}, b \cdot \frac{\rho_{\mathbb{R}}(r)}{r})$ . So,  $d_{\mathbb{P}}((\rho_{\mathbb{R}}(r), a \cdot \frac{\rho_{\mathbb{R}}(r)}{r}, b \cdot \frac{\rho_{\mathbb{R}}(r)}{r}), (r, a, b)) = \max(\rho_{\mathbb{R}}(r), \rho_{\mathbb{R}}(r)) = \rho_{\mathbb{R}}(r)$ .  $\square$

#### 4.1.6 Property C: Property of $\text{op} \in \Sigma$ : metric preservation.

Our operations satisfy metric preservation.

LEMMA 4.5 (OPERATIONS SATISFY METRIC PRESERVATION). Each  $\text{op} \in \Sigma$  satisfies metric preservation.

PROOF. Holds by a case analysis for each operation in  $\Sigma$  and unfolding the definition of our metric function.  $\square$

#### 4.1.7 Property D: Partial order of bounds.

LEMMA 4.6 (PARTIAL ORDER OF  $\mathbb{B}$ ).  $(\mathbb{B}, \leq)$  form a partial order.

PROOF. Follows from the fact that  $\subseteq$  forms a partial order over  $\mathbb{S} \approx$ .  $\square$

#### 4.1.8 Property E: Property of $\text{bop} \in \Sigma_{\text{bnd}}$ : closure over bounds $(\mathbb{B})$ .

LEMMA 4.7 (CLOSURE OVER BOUNDS). For every operation  $\text{bop}(b_0, \dots, b_n) : \Sigma_{\text{num}}$ , we have that if  $c_0, \dots, c_n \in \mathbb{B}$  are  $\text{bop}(c_0, \dots, c_n) = c' : \text{bnd}$  holds.

PROOF. Follows by inspection of the definition of our three supported operations.  $\square$

## 4.2 Translating our error bounds from the paired to the unpaired representation

For a given  $\cdot \vdash e : M_q \text{num}_b$ , we wish to bound the error between  $e$  computed ideally over the reals versus the floating-point approximation. To do this, we prove our paired error theorems (Theorem 4.8, Theorem ??), which for a triple  $(r, a, b)$  relates error on the paired components  $a, b$  with error on  $r$ . By plugging in the theorems proved below, we are able obtain useful bounds over the unpaired representation for a given typed program. We have two main error theorems in

this paper, detailed below: a relative error theorem (Theorem 4.8) and an absolute error theorem (Theorem 4.10).

Note that for the below theorem we have the condition  $0 < r^\downarrow$ . This can be generalized to  $r^\downarrow$  and  $r^\uparrow$  having the same sign by a mirrored proof. When the lower and upper bounds on  $r$  straddle zero, we cannot produce a relative error bound.

**THEOREM 4.8 (PAIRED ERROR THEOREM; A PRIORI RELATIVE).** *For any numbers  $((r, a, b), (r', a', b')) \in \mathbb{P}^2$  and where we have  $r^\downarrow, r^\uparrow, a^\downarrow, a^\uparrow, b^\downarrow, b^\uparrow \in \mathbb{R}^+$  the following bounds*

- (1)  $d_{\mathbb{R}}(a, a'), d_{\mathbb{R}}(b, b') \leq q$
- (2)  $r \in [r^\downarrow, r^\uparrow]$  and  $0 < r^\downarrow$
- (3)  $a \in [a^\downarrow, a^\uparrow]$
- (4)  $b \in [b^\downarrow, b^\uparrow]$

we have that  $d_{\mathbb{R}}(r, r')$  is less than or equal to all of the following bounds:

- (a)  $\max(|\ln(e^{-q} + \frac{a^\uparrow}{r^\downarrow}(e^q - e^{-q}))|, |\ln(e^q + \frac{a^\downarrow}{r^\uparrow}(e^{-q} - e^q))|)$
- (b)  $\max(|\ln(e^q + \frac{b^\uparrow}{r^\downarrow}(e^q - e^{-q}))|, |\ln(e^{-q} + \frac{b^\downarrow}{r^\uparrow}(e^{-q} - e^q))|)$

**PROOF.** Our error metric is symmetric so for this theorem we wish to prove a bound on  $d_{\mathbb{R}}(r', r)$ . We prove each bound and, for each bound, split into two cases:  $r \leq r'$  or  $r > r'$ . In the first case, we have:

$$\begin{aligned} 1 \leq \frac{r'}{r} &\leq \frac{a' - be^{-q}}{a - b} = \frac{a' - be^{-q} + ae^{-q} - ae^{-q}}{a - b} \\ &= e^{-q} + \frac{a' - ae^{-q}}{a - b} \leq e^{-q} + \frac{ae^q - ae^{-q}}{a - b} = e^{-q} + \frac{a}{a - b}(e^q - e^{-q}) \\ &\leq e^{-q} + \frac{a^\uparrow}{r^\downarrow}(e^q - e^{-q}) \end{aligned} \quad (14)$$

Therefore,

$$0 \leq \ln\left(\frac{r'}{r}\right) \leq \ln(e^{-q} + \frac{a^\uparrow}{r^\downarrow}(e^q - e^{-q})) \quad (15)$$

In the second case, we have:

$$\begin{aligned} 1 \geq \frac{r'}{r} &\geq \frac{a' - be^q}{a - b} = \frac{a' - be^q + ae^q - ae^q}{a - b} \\ &= e^q + \frac{a' - ae^q}{a - b} \geq e^q + \frac{ae^{-q} - ae^q}{a - b} = e^q + \frac{a}{a - b}(e^{-q} - e^q) \\ &\geq e^q + \frac{a^\downarrow}{r^\uparrow}(e^{-q} - e^q) \end{aligned} \quad (16)$$

Therefore,

$$0 \geq \ln\left(\frac{r'}{r}\right) \geq \ln(e^q + \frac{a^\downarrow}{r^\uparrow}(e^{-q} - e^q)) \quad (17)$$

A sound bound for both cases is the maximum of the absolute value of Equation 15 and Equation 17, which is exactly our first bound (a). The proof strategy for the second bound (b) mirrors that of the first bound.  $\square$

**COROLLARY 4.9.** *For  $\cdot \vdash e : M_q \text{ num}_{((r^\downarrow, r^\uparrow), (a^\downarrow, a^\uparrow), (b^\downarrow, b^\uparrow), d)}$ ,  $e \mapsto ((r, a, b), (r', a', b'))$  where  $d_{\mathbb{R}}(r, r')$  is less than or equal to all of the following bounds:*

- (a)  $\max(|\ln(e^{-q} + \frac{a^\uparrow}{r^\downarrow}(e^q - e^{-q}))|, |\ln(e^q + \frac{a^\downarrow}{r^\uparrow}(e^{-q} - e^q))|)$
- (b)  $\max(|\ln(e^q + \frac{b^\uparrow}{r^\downarrow}(e^q - e^{-q}))|, |\ln(e^{-q} + \frac{b^\downarrow}{r^\uparrow}(e^{-q} - e^q))|)$

PROOF. By our logical relation and metric preservation theorem (Theorem 10), we know that  $e \mapsto ((r, a, b), (r', a', b'))$  such that  $d_{\mathbb{P}}((r, a, b), (r', a', b')) \leq q$ . Therefore,  $d_{\mathbb{R}}(a, a'), d_{\mathbb{R}}(b, b') \leq q$ . Applying the previous theorem proves the corollary.  $\square$

**THEOREM 4.10 (PAIRED ERROR THEOREM; A PRIORI ABSOLUTE).** *For any numbers  $((r, a, b), (r', a', b')) \in \mathbb{P}^2$  and where we have the following bounds for  $r^{\downarrow}, r^{\uparrow} \in \mathbb{R}, a^{\downarrow}, a^{\uparrow}, b^{\downarrow}, b^{\uparrow} \in \mathbb{R}^+$ :*

- (1)  $d_{\mathbb{R}}(a, a'), d_{\mathbb{R}}(b, b') \leq q$
- (2)  $r \in [r^{\downarrow}, r^{\uparrow}]$
- (3)  $a \in [a^{\downarrow}, a^{\uparrow}]$
- (4)  $b \in [b^{\downarrow}, b^{\uparrow}]$

*we have that  $|r - r'|$  is the maximum of the following:*

- (a)  $a^{\uparrow}(1 - e^{-q}) - b^{\downarrow}(1 - e^q)$
- (b)  $a^{\downarrow}(1 - e^{-q}) - b^{\uparrow}(1 - e^q)$
- (c)  $a^{\uparrow}(e^q - 1) - b^{\downarrow}(e^{-q} - 1)$
- (d)  $a^{\downarrow}(e^q - 1) - b^{\uparrow}(e^{-q} - 1)$

PROOF. We case into whether  $r \geq r'$  (or not) and  $r \geq 0$  (or not). This produces four cases:

**Case  $r \geq r'$  and  $r \geq 0$ .** We have that:

$$\begin{aligned} (a - b) - (a' - b') &\leq (a - b) - (ae^{-q} - be^q) \\ &= a(1 - e^{-q}) - b(1 - e^q) = a^{\uparrow}(1 - e^{-q}) - b^{\downarrow}(1 - e^q) \end{aligned} \quad (18)$$

**Case  $r \geq r'$  and  $r < 0$ .**

$$\begin{aligned} (a - b) - (a' - b') &\leq (a - b) - (ae^{-q} - be^q) \\ &= a(1 - e^{-q}) - b(1 - e^q) = a^{\downarrow}(1 - e^{-q}) - b^{\uparrow}(1 - e^q) \end{aligned} \quad (19)$$

**Case  $r < r'$  and  $r \geq 0$ .**

$$\begin{aligned} (a' - b') - (a - b) &\leq ((ae^q - be^{-q}) - (a - b)) \\ &= a(e^q - 1) - b(e^{-q} - 1) = a^{\uparrow}(e^q - 1) - b^{\downarrow}(e^{-q} - 1) \end{aligned} \quad (20)$$

**Case  $r < r'$  and  $r < 0$ .**

$$\begin{aligned} (a' - b') - (a - b) &\leq ((ae^q - be^{-q}) - (a - b)) \\ &= a(e^q - 1) - b(e^{-q} - 1) = a(e^q - 1) - b(e^{-q} - 1) \end{aligned} \quad (21)$$

By taking the maximum of all possible cases, we can obtain a sound a priori absolute error bound.  $\square$

**COROLLARY 4.11.** *For  $\vdash e : M_q \mathbf{num}_{((r^{\downarrow}, r^{\uparrow}), (a^{\downarrow}, a^{\uparrow}), (b^{\downarrow}, b^{\uparrow}), d)}$ , we know that  $e \mapsto^* ((r, a, b), (r', a', b'))$  where  $|r - r'|$  is the maximum of the following:*

- (1)  $a^{\uparrow}(1 - e^{-q}) - b^{\downarrow}(1 - e^q)$
- (2)  $a^{\downarrow}(1 - e^{-q}) - b^{\uparrow}(1 - e^q)$
- (3)  $a^{\uparrow}(e^q - 1) - b^{\downarrow}(e^{-q} - 1)$
- (4)  $a^{\downarrow}(e^q - 1) - b^{\uparrow}(e^{-q} - 1)$

PROOF. By our logical relation and metric preservation theorem (Theorem 10), we know that  $e \mapsto ((r, a, b), (r', a', b'))$  such that  $d_{\mathbb{P}}((r, a, b), (r', a', b')) \leq q$ . Therefore,  $d_{\mathbb{R}}(a, a'), d_{\mathbb{R}}(b, b') \leq q$ . Applying the previous theorem proves the corollary.  $\square$

## 5 MORE PRECISE TREATMENT OF ADDITION AND SUBTRACTION

In this section, we are interested in improving the precision of our analysis on programs that contain addition and subtraction. We first develop our motivating example to demonstrate the problem. Then, we introduce a new primitive **factor** that improves the precision of addition and subtraction. Next, we discuss how rewriting a program to use **factor** is entirely orthogonal to our bounds analysis and therefore can only strictly improve Negative Fuzz's error bounds. Finally, we discuss the soundness and semantics of **factor**.

*Problem intuition.* We return to our example pairwise summation program  $((w \tilde{+} x) \tilde{+} (y \tilde{+} z))$  from Section 2 whose type is  $M_{3 \cdot u}$  **num**:

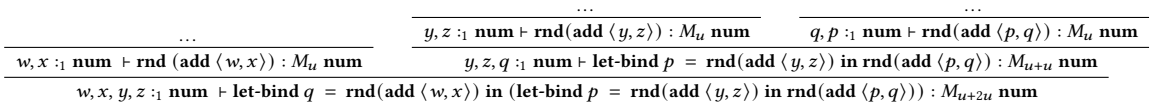
```
let-bind q = rnd (add ⟨w, x⟩) in
let-bind p = rnd (add ⟨y, z⟩) in
  rnd (add ⟨q, p⟩)
```

From the literature, we would expect that the more efficient pairwise summation algorithm to have type  $M_{2 \cdot u}$  **num**, a one-third tighter error bound than the iterative summation algorithm  $((w \tilde{+} x) \tilde{+} y) \tilde{+} z$  shown:

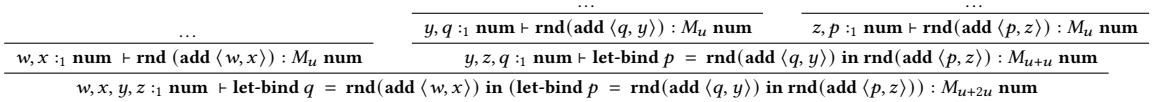
```
let-bind q = rnd (add ⟨w, x⟩) in
let-bind p = rnd (add ⟨q, y⟩) in
  rnd (add ⟨p, z⟩)
```

However, both programs have the same type and error bound.

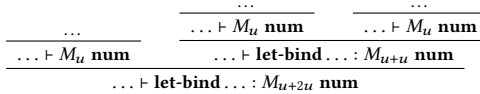
To better understand the restrictions imposed by the type system, we include the relevant portions of the typing trees for the above two programs in Figure 5:



(a) A partial typing tree of the program representing the pairwise summation algorithm on four numbers.



(b) A partial typing tree of the program representing the iterative summation algorithm on four numbers.



(c) A partial typing tree showing the commonalities of both summation algorithms. Note the grades and occurrences of **let-bind** in the typing tree.

Fig. 5. For presentational purposes, we remove polymorphic bound annotations from the **num** type and **add** operation and also omit the applications of the polymorphic typing rules.

The core problem is that it is syntactically impossible to first construct separate typing trees for  $w \tilde{+} x$  and  $y \tilde{+} z$  with error grades  $M_u$  **num**, then combine the two results using floating-point addition to obtain a well-typed computation of error grade  $M_{2u}$  **num**. In both programs (subfigures

a and b), we are forced to sequence the result of each **rnd** with the **let-bind** typing rule in order to use the result in the remainder of the computation. Even though the underlying arithmetic computation of pairwise summation corresponds to a perfect binary tree with less floating-point error, our usage of **let-bind** forces the typing tree and grades of our pairwise summation to mirror (subfigure c) the typing tree of our iterative summation program. This means that both programs have the same  $3u$  monadic error grade.

*Solution: the **factor** primitive.* Adding a primitive **factor** with type  $(M_q\tau_0) \times (M_q\tau_1) \multimap M_q(\tau_0 \times \tau_1)$  allows us to rewrite our running pairwise summation example with **factor** to have type  $M_{2u}$  **num**, which is lower than the  $3u$  error grade of the iterative summation program, shown below:

$$\begin{array}{c}
 \text{let-bind } a = \\
 \text{factor } \langle \text{rnd } (\text{add} \langle w, x \rangle), \rangle \text{ in} \\
 \text{rnd } (\text{add} \langle y, z \rangle) \text{rnd } (\text{add } a)
 \end{array}$$

$$\frac{
 \frac{
 \frac{
 \dots
 }{w, x, y, z : \text{num} \vdash \text{rnd}(\text{add} \langle w, x \rangle) : M_u \text{num}}
 }{w, x, y, z : \text{num} \vdash \langle \text{rnd}(\text{add} \langle w, x \rangle), \text{rnd}(\text{add} \langle y, z \rangle) \rangle : (M_u \text{num}) \times (M_u \text{num})}
 }{w, x, y, z : \text{num} \vdash \text{factor}(\text{rnd}(\text{add} \langle w, x \rangle), \text{rnd}(\text{add} \langle y, z \rangle)) : M_u(\text{num} \times \text{num})}
 }{
 \frac{
 \dots
 }{a : \text{num} \times \text{num} \vdash \text{rnd}(\text{add } a) : M_u \text{num}}
 }{
 w, x, y, z : \text{num} \vdash \text{let-bind } a = \text{factor}(\text{rnd}(\text{add} \langle w, x \rangle), \text{rnd}(\text{add} \langle y, z \rangle)) \text{ in } \text{rnd}(\text{add } a) : M_{u+u} \text{num}
 }$$

Fig. 6. A partial typing tree for the pairwise summation algorithm using the **factor** primitive.

To understand why, consider the partial typing tree displayed in Figure 6. Observe how the **factor** primitive allows us to first “share” the  $u$  monadic error grade and the same typing context to construct **add** $\langle w, x \rangle$  and **add** $\langle y, z \rangle$  on the left-hand-side of the typing tree. Then, only after the contexts and grades are shared, we use a single **let-bind** sequencing operation to pass the floating-point result to the final **add**. So, instead of having to use multiple iteratively sequenced **let-binds** to sequentially **add** the grades of repeated monadic computations, **factor** enables us to first take the *maximum* of any two grades (i.e. share the grade) before sequencing with **let-bind**. This technique scales for arbitrary sequences of addition or subtraction, reducing the monadic grade by a log-factor in size of the sequence.

**factor** is useful for subtraction as well. As **add** and **sub** both map **num**  $\times$  **num**  $\multimap$  **num**, we can rewrite programs with subtraction in a similar manner as addition. For example, in Figure 7 we demonstrate how to use **factor** to achieve tighter error bounds for programs with subtraction. Clearly, **factor** can help our analysis achieve tighter error bounds for both addition and subtraction.

$$\begin{array}{cc}
 \text{let-bind } q = \text{rnd } (\text{sub} \langle w, x \rangle) \text{ in} & \text{let-bind } a = \\
 \text{let-bind } p = \text{rnd } (\text{sub} \langle y, z \rangle) \text{ in} & \text{factor } \langle \text{rnd } (\text{sub} \langle w, x \rangle), \text{rnd } (\text{sub} \langle y, z \rangle) \rangle \text{ in} \\
 \text{rnd } (\text{sub} \langle q, p \rangle) & \text{rnd } (\text{sub } a)
 \end{array}$$

(a) Without factor: the type assigned is  $M_{3u}$  **num**.      (b) With factor: the type assigned is  $M_{2u}$  **num**.

Fig. 7. Side-by-side comparison of subtraction with and without the **factor** primitive. Note that we omit the bounds subscripts for presentational purposes.

*Semantics and soundness of **factor**.* In Negative Fuzz, we expect each step of our operational semantics to maintain the convention that the first component of a monadic pair represents the ideal computation and the second component of a monadic pair represents the rounded computation. This ensures that the monad grade bounds the distance between the ideal and floating-point computation. In Figure 8, we can see that the **rnd**, **ret** stepping rules, which both produce a monadic value of type  $M_q \mathbf{num}$  obey this invariant. We can also see that the **let-bind** stepping rule maintains the invariant and produces a monadic pair if the premises (of type  $M_q \mathbf{num}$ ) also produce a monadic pair:

$$\frac{\begin{array}{l} \mathbf{rnd} \ k \mapsto (k, \rho(k)) \quad \mathbf{ret} \ v \mapsto (v, v) \\ f[v_1/x] \mapsto^* (v_3, \tilde{v}_3) \quad f[v_2/x] \mapsto^* (v_4, \tilde{v}_4) \end{array}}{\mathbf{let-bind} \ x = (v_1, v_2) \ \mathbf{in} \ f \mapsto (v_3, \tilde{v}_4)}$$

Fig. 8. Stepping rules for **rnd**, **ret**, and **let-bind**. The stepping rule for **let-bind** is modified with  $\sim$  annotations to emphasize the rounded portions of the pairs.

The operational semantics of **factor** respects this convention by reassociating pairs of pairs of values, stepping **factor**  $((v_0, \tilde{v}_0), (v_1, \tilde{v}_1)) \mapsto ((v_0, v_1), (\tilde{v}_0, \tilde{v}_1))$ , which can be seen as producing a value of form  $((v_0, v_1), (v_0, v_1))$ . This ensures that **factor** is sound: if the inputs are in  $\mathcal{R}_{M_q \tau \times M_q \tau'}$  then the outputs are in  $\mathcal{R}_{M_q(\tau \times \tau')}$ . Further it obeys metric preservation: if  $v_0$  and  $\tilde{v}_0$  and  $v_1$  and  $\tilde{v}_1$  are both  $q$  apart, then  $(v_0, v_1)$  and  $(\tilde{v}_0, \tilde{v}_1)$  are also  $q$  apart.

## 6 TYPE INFERENCE

The Numerical Fuzz type inference algorithm automatically infers sensitivity and monadic grades. We extend the Numerical Fuzz's type inference algorithm for our new syntax (e.g. **factor**) and to automatically infer polymorphic bounds. This fully eliminates the burden of adopting our extension; programs that type-check in Numerical Fuzz will type check in Negative Fuzz. To use our type inference algorithm, users write a program with no bound polymorphism (instantiation or generalization), sensitivity annotations, or monadic grades. Type inference then attempts to synthesize a new well-typed term with bound polymorphism while simultaneously inferring sensitivities and grades in a typing context with erased sensitivity annotations.

To formally define the type inference problem, we first setup a sensitivity erasure  $-^\bullet$  over typing contexts and a bound erasure  $-^\bullet$  over types and terms. Our erasure functions help us to capture the precise relation between the (erased) term and the new, well-typed synthesized term (with bound polymorphism) with inferred sensitivities and monadic grades.

*Definition 6.1 (Sensitivity erasure).* For a fully annotated typing context  $\Gamma : \text{variables} \rightarrow \text{sensitivities} \times \text{types}$ , a fully erased typing context removes all sensitivity information and only returns the type:  $\Gamma^\bullet = \Gamma \circ \pi_2$ .

*Definition 6.2 (Bound erasure).* We define bound erasure  $(-^\bullet)$  over types and terms below:

$$\begin{array}{ll} \mathbf{unit}^\bullet \triangleq \mathbf{unit} & (!_s \tau)^\bullet \triangleq !_s \tau^\bullet \\ \mathbf{num}_i^\bullet \triangleq \mathbf{num} & (M_u \tau)^\bullet \triangleq M_u \tau^\bullet \\ (\forall \epsilon. \tau)^\bullet \triangleq \tau^\bullet & (\tau_0 \otimes \tau_1)^\bullet \triangleq (\tau_0^\bullet \otimes \tau_1^\bullet) \\ (\tau_0 \times \tau_1)^\bullet \triangleq (\tau_0^\bullet \times \tau_1^\bullet) & (\tau_0 \multimap \tau_1)^\bullet \triangleq (\tau_0^\bullet \multimap \tau_1^\bullet) \end{array} \quad (22)$$

We similarly define an erasure operation for terms:

$$\begin{aligned}
v^\bullet &\triangleq v & [e]^\bullet &\triangleq [e^\bullet] \\
x^\bullet &\triangleq x & (\mathbf{factor} \ e)^\bullet &\triangleq \mathbf{factor} \ e^\bullet \\
(\mathbf{op} \ e)^\bullet &\triangleq \mathbf{op} \ e^\bullet & (\mathbf{in}_i \ e)^\bullet &\triangleq \mathbf{in}_i \ e^\bullet \\
(e \ f)^\bullet &\triangleq e^\bullet \ f^\bullet & (\pi_i \ e)^\bullet &\triangleq \pi_i \ e^\bullet \\
(\mathbf{rnd} \ e)^\bullet &\triangleq \mathbf{rnd} \ e^\bullet & (\mathbf{ret} \ e)^\bullet &\triangleq \mathbf{ret} \ e^\bullet \\
(e\{i\})^\bullet &\triangleq e & \langle e, f \rangle^\bullet &\triangleq \langle e^\bullet, f^\bullet \rangle \\
(\Lambda e. e)^\bullet &\triangleq e & (e, f)^\bullet &\triangleq (e^\bullet, f^\bullet) \\
(\mathbf{case} \ e \ \mathbf{of} \ (\mathbf{in}_1 x. f_1 \mid \mathbf{in}_2 x. f_2))^\bullet &\triangleq (\mathbf{case} \ e^\bullet \ \mathbf{of} \ (\mathbf{in}_1 x. f_1^\bullet \mid \mathbf{in}_2 x. f_2^\bullet)) \\
(\mathbf{let} \ x = e \ \mathbf{in} \ f)^\bullet &\triangleq (\mathbf{let} \ x = e^\bullet \ \mathbf{in} \ f^\bullet) \\
(\mathbf{let-bind} \ x = e \ \mathbf{in} \ f)^\bullet &\triangleq (\mathbf{let-bind} \ x = e^\bullet \ \mathbf{in} \ f^\bullet) \\
(\mathbf{let-cobind} \ x = e \ \mathbf{in} \ f)^\bullet &\triangleq (\mathbf{let-cobind} \ x = e^\bullet \ \mathbf{in} \ f^\bullet) \\
(\mathbf{let-pair} \ (x, y) = e \ \mathbf{in} \ f)^\bullet &\triangleq (\mathbf{let-pair} \ (x, y) = e^\bullet \ \mathbf{in} \ f^\bullet)
\end{aligned}$$

We are now ready to define the type inference algorithm. Our type inference algorithm has the following shape:

$$\Gamma^\bullet; e^\bullet \Longrightarrow \Gamma'; e'; \tau$$

where we start a program with a typing context  $\Gamma^\bullet$  that has erased sensitivity information and an erased program  $e^\bullet$  with no polymorphic bounds, and produces a typing context  $\Gamma'$  with sensitivity information and fully annotated program  $e$  and type  $\tau$  such that  $\Delta \mid \Gamma' \vdash e' : \tau$  is derivable for some  $\Delta$  (our soundness criterion). Note that we leave the  $\Delta$  out of our type inference algorithm and implicitly reconstruct an appropriate  $\Delta$  inside our constructive soundness proof.

We start type inference by passing the empty context, along with the bound-erased program (written  $e^\bullet$ ) to the inference algorithm detailed in Figure 10. Conceptually, our syntax-directed type inference algorithm does two things:

- (1) We infer sensitivities via the type inference algorithm provided in [Kellison and Hsu 2024], extended to deal with our new **factor** primitive and polymorphic bounds as well as support expressions in greater places.
- (2) At the same time as our sensitivity inference, we infer our bounds. We rely on helper functions  $I$  and  $E$  (Figure 9) to discover which polymorphic bound variables needed to be automatically instantiated and eliminated.

We present our syntax-directed type inference algorithm in Figure 10 as a combined algorithm where the sensitivities and bounds are inferred simultaneously in one pass. We prove the soundness of the fused sensitivity-and-bound pass in Theorem 6.6 with a syntactic argument inducting over the inference algorithm. After running type inference, we simplify the closed bound expressions in the type using  $\llbracket - \rrbracket$ . We prove the semantic soundness of our simplification technique in Theorem 6.8.

*The type inference algorithm.* Our syntax-directed type inference algorithm is laid out in Figure 10. The majority of cases (Unit, Var, ! I, **op**, Ret, Rnd,  $\times$  I,  $\times$  E,  $\otimes$  I, **let**, **let-pair**, **let-bind**, **case**) come from Numerical Fuzz's type inference algorithm. We walkthrough the remaining cases:  $\multimap$  I relies on  $I(\tau^\bullet)$  to create new globally fresh bound variables.  $\multimap$  E relies on  $E(\tau)$  to extract the new bound variables and our type equality check to ensure that the extracted variables have been soundly instantiated. Both the  $I$  and  $E$  helper functions are fully defined in Figure 9. Finally, the **factor** case is straightforward.

1177  
1178  
1179  
1180  
1181  
1182  
1183  
1184  
1185  
1186  
1187  
1188  
1189  
1190  
1191  
1192  
1193  
1194  
1195  
1196  
1197  
1198  
1199  
1200  
1201  
1202  
1203  
1204  
1205  
1206  
1207  
1208  
1209  
1210  
1211  
1212  
1213  
1214  
1215  
1216  
1217  
1218  
1219  
1220  
1221  
1222  
1223  
1224  
1225

Fig. 9. Ancillary functions defining algorithmic type inference. `I` creates bounds for an type with erased bounds. `E` extracts the bounds used in an unerased type. For type checking to succeed on many real-world programs, it is important for `I` and `E` to align in the production and extraction of bound variables.

*Type inference soundness.* We now prove the soundness of our type inference algorithm. We use the following lemma for reasoning about the bound polymorphism variables in the  $\multimap I$  and  $\multimap E$  cases for the type inference soundness proof.

1203       LEMMA 6.3 (FREE VARS IN  $\Gamma$ ). *For any typing derivation for some  $\Delta \mid \Gamma \vdash e : \tau$ ,  $FTV(\Gamma) \subseteq \Delta$  and*  
1204  *$FTV(\tau) \subseteq \Delta$ .*

PROOF. By induction over the  $\vdash$  relation. □

1207 We use the following lemma for reasoning about the bound polymorphism variables in the  $\multimap I$   
1208 case for the type inference soundness proof.

LEMMA 6.4 (FREE VARS IN I.). *For any type  $\tau$ ,  $I(\tau) = \tau'; \epsilon_0 \dots \epsilon_n$ , we have that  $FTV(\tau') = \{\epsilon_0, \dots, \epsilon_n\}$ .*

1212 PROOF. By induction over the definition of  $I$ . □

We use the following lemma for reasoning about the bound polymorphism variables in the  $\multimap E$  case for the type inference soundness proof.

LEMMA 6.5 (WELL-FORMEDNESS OF BOUNDS PRODUCED BY  $E$ ). *For any types  $\tau$ ,  $E(\tau) \rightarrow i_0, \dots, i_n$  means that for every  $i_j$ , if  $FTV(\tau) \subseteq \Delta$  then  $\Delta \vdash i_j$ .*

PROOF. We apply weakening (over bound contexts, Lemma 3.14) to the conclusion. So it suffices to prove that  $FTV(\tau) \vdash i_j$ . We complete the proof by induction over  $\tau$ .  $\square$

Now we are able to prove the soundness of our type inference algorithm. Note that we do not prove that our type inference algorithm infers the tightest possible sensitivity or round-off grades. So, our soundness theorem states that the inferred type, typing contexts, and term, after erasing, are equal to the original erased type, erased context, and erased term ( $\Gamma^\bullet = \Gamma^{\bullet\bullet}$  and  $e^\bullet = e'^{\bullet\bullet}$ ).

$$\begin{array}{c}
\frac{}{\Gamma^\bullet; \langle \rangle^\bullet \Rightarrow \Gamma^0; \langle \rangle; \mathbf{unit}} \text{(Unit)} \quad \frac{}{\Gamma^\bullet, x : \tau, x^\bullet \Rightarrow \Gamma^0, x :_1 \tau; x; \tau} \text{(Var)} \\
\frac{}{\Gamma^\bullet; k^\bullet \Rightarrow \Gamma^0; k; \mathbf{num}_{inj(k)}} \text{(Const)} \quad \frac{\Gamma^\bullet; e^\bullet \Rightarrow \Gamma'; e'; \tau}{\Gamma^\bullet; [e]_s^\bullet \Rightarrow s * \Gamma'; [e']_s; !_s \tau} (! \text{I}) \\
\frac{I(\tau_0^\bullet) = \tau_0; \epsilon_0, \dots, \epsilon_n \quad \Gamma^\bullet, x : \tau_0; e^\bullet \Rightarrow \Gamma', x :_s \tau_0; e'; \tau \quad s \geq 1}{\Gamma^\bullet; \lambda(x : \tau_0^\bullet). e^\bullet \Rightarrow \Gamma'; \Lambda \epsilon_0, \dots, \epsilon_n (\lambda(x : \tau_0). e'); \forall \epsilon_0, \dots, \epsilon_n. \tau_0 \multimap \tau} (\multimap \text{I}) \\
\frac{\Gamma^\bullet; f^\bullet \Rightarrow \Gamma'; f'; \forall \epsilon_0, \dots, \epsilon_n. \tau_0 \multimap \tau \quad E(\tau_0') = i_0, \dots, i_n \quad \Gamma^\bullet; e^\bullet \Rightarrow \Theta; e'; \tau_0'}{\Gamma^\bullet; (f \ e)^\bullet \Rightarrow \Gamma' + \Theta; (f' \{i_0\} \dots \{i_n\}) e'; \tau[\epsilon_0/i_0, \dots, \epsilon_n/i_n]} (\multimap \text{E}) \\
\frac{\{\mathbf{op} : \forall \epsilon_0, \epsilon_1, \tau_0 \multimap \tau_1\} \in \Sigma}{\Gamma^\bullet; \mathbf{op}^\bullet \Rightarrow \Gamma; \mathbf{op}; \forall \epsilon_0, \epsilon_1, \tau_0 \multimap \tau_1} (\mathbf{op}) \\
\frac{\Gamma^\bullet; e^\bullet \Rightarrow \Gamma'; e'; (M_q \tau_0 \times M_r \tau_1)}{\Gamma^\bullet; \mathbf{factor} e^\bullet \Rightarrow \Gamma'; \mathbf{factor} e'; M_{\max(q,r)}(\tau_0 \times \tau_1)} (\mathbf{factor}) \\
\frac{\Gamma^\bullet; e^\bullet \Rightarrow \Gamma'; e'; \tau}{\Gamma^\bullet; \mathbf{ret} e^\bullet \Rightarrow \Gamma'; \mathbf{ret} e'; M_0 \tau} (\mathbf{Ret}) \quad \frac{\Gamma^\bullet; e^\bullet \Rightarrow \Gamma'; e'; \tau}{\Gamma^\bullet; \mathbf{rnd} e^\bullet \Rightarrow \Gamma'; \mathbf{rnd} e'; M_q \tau} (\mathbf{Rnd}) \\
\frac{\Gamma^\bullet; e^\bullet \Rightarrow \Gamma_0; e'; \tau_0 \quad \Gamma^\bullet; f^\bullet \Rightarrow \Gamma_1; f'; \tau_1}{\Gamma^\bullet; \langle f, e \rangle^\bullet \Rightarrow \max(\Gamma_0, \Gamma_1); \langle f', e' \rangle; \tau_0 \times \tau_1} (\times \text{I}) \quad \frac{\Gamma^\bullet; e^\bullet \Rightarrow \Gamma'; e'; \tau_0 \times \tau_1}{\Gamma^\bullet; (\pi_i \ e)^\bullet \Rightarrow \Gamma'; \pi_i e'; \tau_i} (\times \text{E}) \\
\frac{\Gamma^\bullet; f^\bullet \Rightarrow \Gamma_0; f'; \tau_1 \quad \Gamma^\bullet; e^\bullet \Rightarrow \Gamma_1; e'; \tau_0}{\Gamma^\bullet; \langle f, e \rangle^\bullet \Rightarrow \Gamma_0 + \Gamma_1; (f', e'); \tau_0 \otimes \tau_1} (\otimes \text{I}) \\
\frac{\Gamma^\bullet; e^\bullet \Rightarrow \Gamma'; e'; \tau_0 \quad \Gamma^\bullet, x : \tau_0; f^\bullet \Rightarrow \Theta, x :_s \tau_0; f'; \tau}{\Gamma^\bullet; (\mathbf{let} \ x = e \ \mathbf{in} \ f)^\bullet \Rightarrow s \cdot \Gamma + \Theta; \mathbf{let} \ x = e' \ \mathbf{in} \ f'; \tau} (\mathbf{let}) \\
\frac{\Gamma^\bullet; e^\bullet \Rightarrow \Gamma'; e'; \tau_0 \otimes \tau_1 \quad \Gamma^\bullet, x : \tau_0, y : \tau_1; f^\bullet \Rightarrow \Theta, x :_s \tau_0, y :_{s'} \tau_1; f'; \tau}{\Gamma^\bullet; (\mathbf{let-pair} \ (x, y) = e \ \mathbf{in} \ f)^\bullet \Rightarrow \max(s, s') \cdot \Gamma + \Theta; \mathbf{let-pair} \ (x, y) = e' \ \mathbf{in} \ f'; \tau} (\mathbf{let-pair}) \\
\frac{\Gamma^\bullet; e^\bullet \Rightarrow \Gamma'; e'; M_r \tau_0 \quad \Gamma^\bullet, x : \tau_0; f^\bullet \Rightarrow \Theta, x :_s \tau_0; f; M_q \tau}{\Gamma^\bullet; (\mathbf{let-bind} \ x = e \ \mathbf{in} \ f)^\bullet \Rightarrow s \cdot \Gamma' + \Theta; \mathbf{let-bind} \ x = e' \ \mathbf{in} \ f'; M_{s,r+q} \tau} (\mathbf{let-bind}) \\
\frac{\Gamma^\bullet; e^\bullet \Rightarrow \Gamma; e'; \tau_0 + \tau_1 \quad \Gamma^\bullet, x : \tau_0; f_1^\bullet \Rightarrow \Theta_0, x :_s \tau_0; f_1'; \tau' \quad s > 0 \quad \Gamma^\bullet, x : \tau_0; f_2^\bullet \Rightarrow \Theta_1, x :_s \tau_0; f_2'; \tau'}{\Gamma^\bullet; (\mathbf{case} \ e \ \mathbf{of} \ (\mathbf{in}_1 \ x. f_1 \mid \mathbf{in}_2 \ x. f_2))^\bullet \Rightarrow s \cdot \Gamma + \max(\Theta_0, \Theta_1); (\mathbf{case} \ e' \ \mathbf{of} \ (\mathbf{in}_1 \ x. f_1' \mid \mathbf{in}_2 \ x. f_2'))^\bullet; \tau'} (\mathbf{case})
\end{array}$$

Fig. 10. Algorithmic rules. Red text are user-required annotations. To help define the  $\Rightarrow$  relation, we define a  $\rightarrow$  relation that annotates (removes the bullet) for a given type.

**THEOREM 6.6 (ALGORITHMIC SOUNDNESS).** *If  $\Gamma^\bullet; e^\bullet \Rightarrow \Gamma'; e'; \tau$  then there exists some  $\Delta$  such that  $\Delta \mid \Gamma' \vdash e' : \tau$  has a typing derivation where  $\Gamma^\bullet = \Gamma'^\bullet$  and  $e^\bullet = e'^\bullet$ .*

**PROOF.** We induct over each typing / case in the type inference algorithm.

**unit.** Holds trivially.

**Var.** Let  $\Delta$  contain all the bound variables mentioned by  $\Gamma$  and  $\tau$ . Then we trivially have a typing derivation for  $\Delta \mid \Gamma, x : \tau \vdash x : \tau$ .

**Const.** Holds trivially.

**! I.** Holds trivially. Note that our sensitivity scaling factor  $s$  is user-annotated.

$\multimap$  **I.** By our inductive hypothesis, we know that there exists a  $\Delta'$  such that we have a typing derivation  $d$  for  $\Delta' \mid \Gamma', x :_s \tau_0 \vdash e' : \tau$ . We wish for each  $\epsilon_0, \dots, \epsilon_n$  to apply our  $\forall$  I rule to  $d$ .

For each application of our  $\forall$  I rule using variable  $\epsilon_i$ , we must satisfy, from the premisses, that: (1)  $\epsilon_i \in \Delta'$  and (2)  $\epsilon_i \notin FTV(\Gamma)$ :

Let  $\Delta = FTV(\tau_0) = \{\epsilon_0, \dots, \epsilon_n\}$  by Lemma 6.4. Then, we have that  $\Delta \subseteq \Delta'$  by Lemma 6.3 and therefore each  $\epsilon_0, \dots, \epsilon_n \notin \Delta'$ . We further have that  $FTV(\Gamma) \cap \Delta = \emptyset$  as  $I$  only generates fresh vars. So both of our two conditions have been satisfied for all  $\epsilon_i$ .

$\multimap$  **E.** By our inductive hypothesis, we know that there exist derivations:

- $d_0$  where  $\Delta' \mid \Gamma' \vdash f : \forall \epsilon_0, \dots, \epsilon_n, \tau_0 \multimap \tau$
- $d_1$  where  $\Delta'' \mid \Theta \vdash e : \tau'_0$

Note that  $\epsilon_0, \dots, \epsilon_n$  are fresh from  $\Delta''$  as each  $\epsilon$  generated is globally fresh in our type inference algorithm.

By weakening on our inductive hypothesis, there exists a  $\Delta$  such that we have a derivation for  $\Delta \mid \Theta \vdash e : \tau_0[\epsilon_0/i_0, \dots, \epsilon_n/i_n]$ . Then we are able to iteratively apply the  $\forall$  E typing rule to our derivation  $d_0$  for each  $\epsilon_0, \dots, \epsilon_n$ , obtaining us the following intermediate typing derivation  $d_3: \Delta' - \{\epsilon_0, \dots, \epsilon_n\} \cup \Delta'' \mid \Gamma' \vdash f' \{i_0\} \dots \{i_n\} : (\tau_0 \multimap \tau)[\epsilon_0/i_0, \dots, \epsilon_n/i_n]$  whose type is the same as the following  $\tau_0[\epsilon_0/i_0, \dots, \epsilon_n/i_n] \multimap \tau[\epsilon_0/i_0, \dots, \epsilon_n/i_n]$ .

Note that the  $\Delta \vdash i : \mathbf{bnd}$  condition in each application of the  $\forall E$  rule is satisfied by applying Lemma 6.5 and Lemma 6.3 on the polymorphic bound judgements. In particular, by Lemma 6.3, we have that  $FTV(\tau'_0[\epsilon_0/i_0, \dots, \epsilon_n/i_n]) \subseteq FTV(\tau'_0) \subseteq FTV(\Theta) \subseteq \Delta''$ . Applying Lemma 6.5 and our inductive hypothesis finishes each proof obligation. Now we are ready to apply our  $\multimap$  E typing rule with derivation  $d_1$  to obtain a derivation for:  $\Delta' \cup \Delta'' \vdash (f' \{i_0\} \dots \{i_n\})e' : \tau[\epsilon_0/i_0, \dots, \epsilon_n/i_n]$  which completes the case.

**Op.** Holds trivially.

**Factor.** Holds trivially.

**Ret.** Holds trivially.

**Rnd.** Holds trivially.

$\times$  **I.** Like Numerical Fuzz, we define  $\max$  as the least upper bound of  $\Gamma_0$  and  $\Gamma_1$ . Let  $\Delta_0$  and  $\Delta_1$  be the (implicit) polymorphic bound contexts generated by our inductive hypothesis. By weakening, we can produce a typing derivation for the following judgement:  $\Delta_0 \cup \Delta_1 \mid \max(\Gamma_0, \Gamma_1) \vdash \langle f', e' \rangle; \tau_0 \times \tau_1$  which completes the case.

$\times$  **E.** Holds trivially.

$\otimes$  **I.** Holds trivially.

**Let.** Holds trivially.

**Let-pair.** Holds by applying weakening to the second inductive hypothesis to obtain a derivation  $\Delta \mid \Gamma, x :_{\max(s, s')} \tau_0, y :_{\max(s, s')} \tau_1 \vdash f' : \tau$  and applying the corresponding  $\otimes$  E rule to complete the case.

**Let-bind.** Holds trivially.

$\otimes$  **E.** We finish by applying our  $\otimes$  E typing rule.

□

*Simplifying the type.* After running type inference we obtain a result in the form:  $\Delta \mid \Gamma \vdash e : \tau$ . In some cases,  $\tau$  might contain bound expressions. For example, after running type inference on the program  $\text{add}\langle 1, 2 \rangle$ , we might obtain the following type  $\vdash \text{add}\langle 1, 2 \rangle : M_u \text{ num}_{(1,1,0,\text{True})+(2,2,0,\text{True})}$ . Semantically, this type represents the same relation as:  $\vdash \text{add}\langle 1, 2 \rangle : M_u \text{ num}_{(3,3,0,\text{True})}$

To simplify the type, we extend the  $\llbracket - \rrbracket$  (Definition 3.7) to simplify types that possibly contain bound expressions. Importantly, the logical relation corresponding to the simplified type is the same logical relation as the original type. This is a semantic notion of soundness, which is why we present the evaluation of types and bound expressions separately from our syntax-directed type inference algorithm.

*Definition 6.7 (Semantic type simplification).* Our type evaluation scheme  $\llbracket - \rrbracket$  takes in a type  $\tau$  and returns a simplified type  $\tau'$ :

$$\begin{aligned}
 \llbracket \text{unit} \rrbracket &= \text{unit} \\
 \llbracket \text{num}_b \rrbracket &= \text{num}_{\llbracket b \rrbracket} \\
 \llbracket \tau_0 \times \tau_1 \rrbracket &= \llbracket \tau_0 \rrbracket \times \llbracket \tau_1 \rrbracket \\
 \llbracket \tau_0 \otimes \tau_1 \rrbracket &= \llbracket \tau_0 \rrbracket \otimes \llbracket \tau_1 \rrbracket \\
 \llbracket \tau_0 + \tau_1 \rrbracket &= \llbracket \tau_0 \rrbracket + \llbracket \tau_1 \rrbracket \\
 \llbracket \tau_0 \multimap \tau_1 \rrbracket &= \llbracket \tau_0 \rrbracket \multimap \llbracket \tau_1 \rrbracket \\
 \llbracket !_s \tau \rrbracket &= !_s \llbracket \tau \rrbracket \\
 \llbracket M_u \tau \rrbracket &= M_u \llbracket \tau \rrbracket \\
 \llbracket \forall \epsilon. \tau \rrbracket &= \forall \epsilon. \tau
 \end{aligned} \tag{23}$$

Note that for simplicity, we halt simplification when we see a  $\forall$ .

**THEOREM 6.8 (SOUNDNESS OF EVALUATING CLOSED BOUNDS).** *If  $e \in \mathbb{R}_\tau$  then  $e \in \mathbb{R}_{\llbracket \tau \rrbracket}$ .*

**PROOF.** We induct over our type  $\tau$  to show that  $\mathbb{R}_\tau = \mathbb{R}_{\llbracket \tau \rrbracket}$  at each step. The structural cases are immediate. The only non-structural case evaluates  $\text{num}_b$  to  $\text{num}_{\llbracket b \rrbracket}$ . Since  $\mathcal{R}_b = \mathcal{R}_{\llbracket b \rrbracket}$  refer to the same underlying set, we are done.  $\square$

## 7 IMPLEMENTATION AND EVALUATION

To evaluate our approach, we implement the type inference algorithm detailed in this paper in Rust. We evaluate the speed and precision of our implementation against competing tools on the FPBench [Damouche et al. 2016] and Satire [Tirpankar et al. 2025] benchmark suite. We only include FPBench programs that are newly supported, e.g. contain subtraction or range over negative numbers. Many of the programs in the FPBench benchmark suite are too small to draw conclusions about the scalability of our approach. Therefore, we draw upon large programs from the Satire benchmark suite [Tirpankar et al. 2025], which are taken from real-world programs. Due to the sheer size of the programs in the Satire suite, we only translated three program schemas into Negative Fuzz (with and without **factor**), Gappa, and FPTaylor in our evaluation.

For all of our benchmarks, we use the BINARY64 precision mode and set the rounding mode to round towards infinity. We convert all bounds to absolute error for an apples-to-apples comparison against competing tools. We run all of our benchmarks on an AWS c5.metal instance (96 vCPU, 192 GiB RAM). Each timing benchmark result reported is the median of 6 benchmark runs. Our approach has several assumptions and limitations. Like Numerical Fuzz, we assume that there is no under or overflow in the program. We do not support division or taking square roots; we leave this as a potential future direction.

*Small benchmark programs.* We compare the precision and performance of our approach on small benchmark programs from FPBench against two alternative approaches: FPTaylor [Solovyev et al. 2018] and Gappa [Daumas and Melquiond 2010].<sup>4</sup> FPTaylor and Gappa occupy a different part of the performance-precision design space compared to Negative Fuzz: both FPTaylor and Gappa aim for precise, sound floating-point analyses via global optimization or rewriting rather than speed or scalability.

We observe that Negative Fuzz (with **factor**) obtains competitive bounds within an order of magnitude with Gappa and FPTaylor. We also compare the performance of our NegFuzz implementation on small benchmarks in Table 4. Negative Fuzz is frequently orders of magnitude faster than Gappa and FPTaylor.

To understand how **factor** contributes the precision of our approach, we perform a simple ablation by manually translating each benchmark from FPBench into Negative Fuzz with and without the **factor** primitive. In all of the examples in Table 3, we can see that using the **factor** strictly improves the precision of our analysis. We did not specifically optimize our implementation of **factor** so it is difficult to draw any conclusions about **factor**'s performance overhead.

Table 3. Absolute error bounds on small benchmarks. Ratio is the ratio of the bound produced by NegFuzz to the tool under test (lower is better). The best value in each row is bolded.

Benchmark	NegFuzz	NegFuzz (factor)	Gappa	FPTaylor
add-assoc	2.67e-15 (1.0×)	1.78e-15 (0.7×)	<b>1.22e-15 (0.5×)</b>	1.33e-15 (0.5×)
delta4	1.09e-12 (1.0×)	4.76e-13 (0.4×)	2.12e-13 (0.2×)	<b>1.15e-13 (0.1×)</b>
himmilbeau	1.62e-11 (1.0×)	8.62e-12 (0.5×)	2.00e-12 (0.1×)	<b>1.18e-12 (0.1×)</b>
kepler0	9.53e-13 (1.0×)	6.81e-13 (0.7×)	1.98e-13 (0.2×)	<b>1.17e-13 (0.1×)</b>
kepler1	5.94e-12 (1.0×)	2.84e-12 (0.5×)	7.46e-13 (0.1×)	<b>3.92e-13 (0.1×)</b>
matrixDeterminant	2.27e-11 (1.0×)	6.66e-12 (0.3×)	5.98e-12 (0.3×)	<b>3.13e-12 (0.1×)</b>
matrixDeterminant2	2.27e-11 (1.0×)	6.66e-12 (0.3×)	5.98e-12 (0.3×)	<b>3.13e-12 (0.1×)</b>
rigidBody1	9.39e-13 (1.0×)	7.83e-13 (0.8×)	5.01e-13 (0.5×)	<b>4.26e-13 (0.5×)</b>
rigidBody2	1.83e-10 (1.0×)	9.13e-11 (0.5×)	6.49e-11 (0.4×)	<b>4.54e-11 (0.2×)</b>
sineOrder3	4.57e-15 (1.0×)	3.27e-15 (0.7×)	1.27e-15 (0.3×)	<b>9.20e-16 (0.2×)</b>
sqroot	4.86e-15 (1.0×)	1.73e-15 (0.4×)	1.04e-15 (0.2×)	<b>9.71e-16 (0.2×)</b>
sum	2.67e-14 (1.0×)	1.33e-14 (0.5×)	6.66e-15 (0.2×)	<b>4.44e-15 (0.2×)</b>
test01_sum3	2.67e-14 (1.0×)	1.33e-14 (0.5×)	6.66e-15 (0.2×)	<b>4.44e-15 (0.2×)</b>

*Large benchmark programs.* To understand how Negative Fuzz scales, we run three large parameterized benchmarks, which we discuss below. Our large-scale comparisons are primarily intended to compare against Satire, which is intended to be scalable by taking a global optimization approach that allows users to trade-off precision for analysis time. We include Gappa and FPTaylor to demonstrate that those two tools do not scale well and often timeout. We run each benchmark until a 450 second timeout is reached or over 64 GB of RAM is used. Both the Horner and iterative summation schemes have a fixed evaluation order that makes rewriting with **factor** unhelpful, so we do not bother writing benchmark variants with **factor**. We plot the results in Figure 11. Each column represents a set of parameterized benchmarks. The top row represents the absolute error bound achieved by each tool. The bottom row represents the median execution time of each program (across 6 runs). If any of the 6 runs failed due to timeout or OOM, we marked the result as a point on the uppermost edge of each plot. We discuss the benchmarks below:

<sup>4</sup>Note that the add-assoc benchmark is not from FPBench and is instead taken from our running example of pairwise summation on  $w, x, y, z \in [-1, 1]$ .

Table 4. Execution time (seconds) on small benchmarks. Ratios show slowdown relative to NegFuzz (lower is better). The best value in each row is bolded.

Benchmark	NegFuzz	NegFuzz (factor)	Gappa	FPTaylor
add-assoc	0.0015 (1.0×)	<b>0.0014 (0.9×)</b>	0.0159 (11×)	0.201 (136×)
delta4	<b>0.0019 (1.0×)</b>	0.0035 (1.8×)	0.2213 (114×)	0.511 (264×)
himmilbeau	<b>0.0015 (1.0×)</b>	0.0018 (1.3×)	0.0522 (36×)	0.023 (16×)
kepler1	<b>0.0030 (1.0×)</b>	0.0100 (3.3×)	0.3772 (124×)	12.353 (4053×)
matrixDeterminant	<b>0.0023 (1.0×)</b>	0.0033 (1.4×)	0.2309 (101×)	12.935 (5672×)
matrixDeterminant2	<b>0.0022 (1.0×)</b>	0.0025 (1.1×)	0.2168 (97×)	13.297 (5940×)
rigidBody1	<b>0.0014 (1.0×)</b>	0.0020 (1.4×)	0.0362 (25×)	0.194 (135×)
rigidBody2	<b>0.0020 (1.0×)</b>	0.0028 (1.4×)	0.0824 (41×)	0.202 (101×)
sineOrder3	0.0014 (1.0×)	<b>0.0014 (1.0×)</b>	0.0216 (16×)	0.020 (15×)
sqroot	<b>0.0018 (1.0×)</b>	0.0027 (1.5×)	0.1091 (59×)	0.028 (15×)
sum	<b>0.0015 (1.0×)</b>	0.0020 (1.3×)	0.0613 (40×)	0.391 (257×)
test01_sum3	<b>0.0017 (1.0×)</b>	0.0019 (1.1×)	0.0607 (35×)	0.417 (239×)
<b>Geo. mean</b>	0.0018 (1.0×)	0.0025 (1.4×)	0.0818 (45×)	0.402 (222×)

- (1) We evaluate the Horner polynomial evaluation scheme on polynomials of degree  $2^2$  to  $2^{11}$  and we observe that Negative Fuzz has comparable precision at significantly faster performance.
- (2) We evaluate matrix multiplication on matrices of  $2^2 \times 2^2$  to  $2^7 \times 2^7$ . On large inputs, we observe that Negative Fuzz (without **factor**) comparable precision at significantly faster performance. Rewriting the program with **factor** allows Negative Fuzz to achieve significantly better precision on large inputs than competing tools.
- (3) We evaluate the iterative summation algorithm (first discussed in Section 2) on sequences of addition from  $2^2$  to  $2^7$ . We observe that Negative Fuzz has comparable precision at significantly faster performance.

*Impact of factor.* We observe that rewriting a program with **factor** can only improve our error analysis. For a program of type  $M_q \text{ num}_b$ , our analysis depends on two pieces of data obtained from our type:

- (1) The grade on the monad  $q$ . In our evaluation benchmarks (Section 7), we observe that rewriting a program with **factor** only reduces the inferred monadic grade.
- (2) The result of the bound analysis  $b$ , which is entirely orthogonal to the usage of **factor**. In fact, we observe that for each benchmark in our evaluation rewritten to use **factor**, the inferred bounds subscript on **num** remains unchanged.

Therefore, we find that each program rewritten **factor** only improved our grade  $q$  while keeping the bound  $b$  the same. By inspecting our error theorems (Theorem 4.10 and Theorem 4.8) we note that a program rewritten to have a lower monadic grade  $q$ , all else equal, results in a strictly better final error bound.

*Evaluation takeaways.* From our evaluation tables and figures, we conclude that when compared with Gappa, FPTaylor, and Satire: (1) our type-based approach is faster, (2) yields useful and competitive error bounds, and (3) the **factor** primitive often results in improved error bounds.

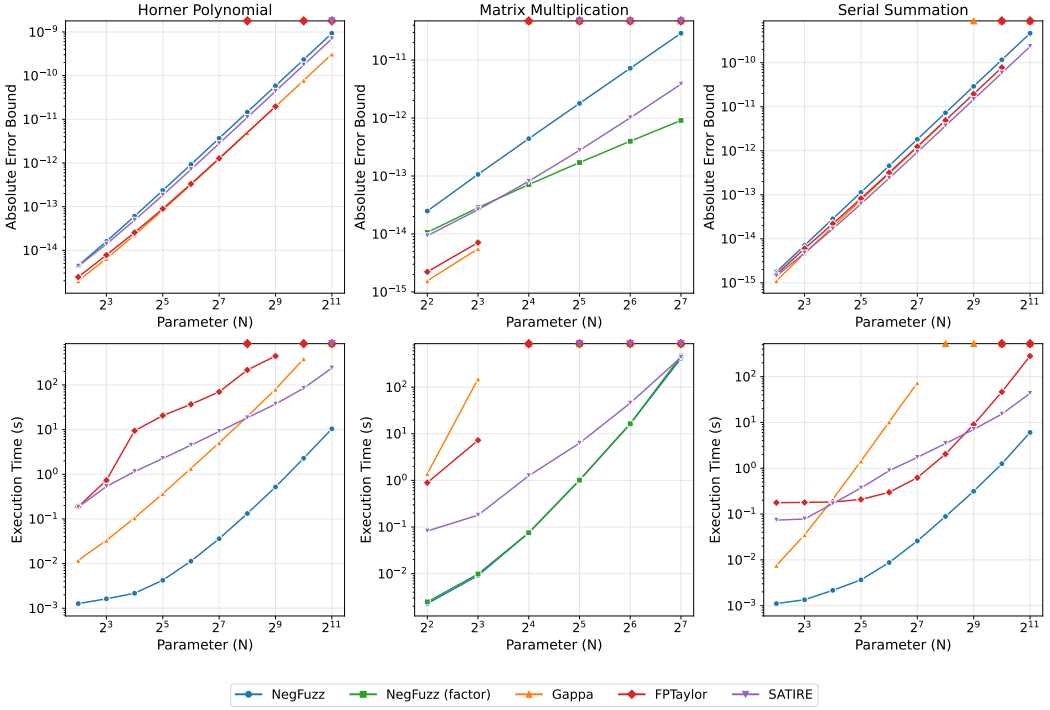


Fig. 11. Log-scale growth plots of absolute error and execution time of Negative Fuzz (with and without factor), Gappa, FPTaylor, and Satire on large benchmarks. For all plots, smaller is better.

## 8 RELATED WORK

*Automated numerical analysis techniques.* There are several extant type-based numerical analysis techniques. Most relevantly, this work extends Numerical Fuzz [Kellison and Hsu 2024] to support subtraction, negative numbers, and more precisely capture addition and subtraction. Numerical Fuzz's type system is based off the Fuzz line of work [?], which uses sensitivities and monads for reasoning about differential privacy. Bean [Kellison et al. 2025] is a typed-based approach to backwards error analysis, which is a different notion of numerical used in Numerical and Negative Fuzz. A floating-point program is backwards stable if the floating-point result is the correct result of running the ideal program given approximate nearby inputs. Backwards error analysis seeks to bound the maximum distance between the actual and approximate inputs. Martel [?] also uses a dependent type system to track round-off error. However, Martel proves a weaker subject reduction theorem and does not have an error soundness theorem.

*Other automated numerical analysis techniques.* There are many non-type-based approaches to automatic numerical analysis. For example, Fluctuat [?] and Gappa [Daumas and Melquiond 2010] both use interval analysis and can reason about programs with conditionals and loops. However, both tools suffer from scalability challenges. FPTaylor [Solovyev et al. 2018] uses symbolic Taylor expansions and global optimization to estimate round-off error. Both Gappa and FPTaylor are able to produce computer-checked certificates of their round-off error bounds. Abbasi and Darulova [?] also use Taylor expansions but take a more local and modular approach that allows analyses to be shared at different function call sites. Rosa [?] [?] also uses Taylor expansions for handling

propagation error. In contrast, our approach supports different rounding modes and higher-order functions. However, we do not support general recursion and restrict conditionals to not contain floating-point error.

*Interval-style bounds analysis in the type system.* The Checker framework [?] provides a modular approach to extending Java’s type system with new typing rules. Several interval analysis checkers have been written [?] [?], but they are designed to find bugs rather than for a roundoff error analysis. There is also work to study type inference from the perspective of a generic abstract interpretation analysis [?] and to be able to turn an abstract interpretation analysis into a type inference problem [?].

*Grothendieck construction of the integers from the naturals.* The idea of constructing the integers via a pairing of the naturals has existed since at least an unpublished manuscript by Richard Dedekind in 1872. The construction also appears in several introductory textbooks in algebra and category theory.

## 9 CONCLUSION

Our work is also the first type-based approach that is able to reason about forwards round-off error in the presence of subtraction and negative numbers. It is faster than comparable approaches, often by orders of magnitude, and offers competitive precision on a wide range of small and large benchmarks. Further, since Negative Fuzz uses a type-based approach to numerical analysis, our approach also offers several qualitative advantages over competing tools. For example, programs that call a library function would not need to type-check the underlying library code and could instead rely on the function type as an interface specification, saving type checking time. As another example of the potential benefit of avoiding a global optimization problem, type checking could also be performed in parallel or incrementally over the dependency graph of the program.

There are three immediate directions we suggest for future work. Firstly, future work might be able to extend Negative Fuzz to support addition, for example via a four-part representation  $(r, a, b, c, d)$  where  $r = \frac{a-b}{c-d}$ . Secondly, in the development of our bounds analysis, we discovered that the choice of bounds domain greatly impacted the precision of our analysis. Future work could use more complex bounds analyses to further improve precision. Thirdly, rewriting a program to use **factor** is quite rote; future work could automatically rewrite a program to an equivalent program with **factor**.

## REFERENCES

- Nasrine Damouche, Matthieu Martel, Pavel Panchekha, Chen Qiu, Alexander Sanchez-Stern, and Zachary Tatlock. 2016. Toward a Standard Benchmark Format and Suite for Floating-Point Analysis. In *umerical Software Verification. NSV 2016. Lecture Notes in Computer Science (Lecture Notes in Computer Science, Vol. 10152)*. Toronto, Canada, 63–77. <https://hal.science/hal-03971234>
- Eva Darulova and Viktor Kuncak. 2017. Towards a Compiler for Reals. *ACM Trans. Program. Lang. Syst.* 39, 2, Article 8 (March 2017), 28 pages. <https://doi.org/10.1145/3014426>
- Marc Daumas and Guillaume Melquiond. 2010. Certification of bounds on expressions involving rounded operators. *ACM Trans. Math. Softw.* 37, 1, Article 2 (Jan. 2010), 20 pages. <https://doi.org/10.1145/1644001.1644003>
- Ariel E. Kellison and Justin Hsu. 2024. Numerical Fuzz: A Type System for Rounding Error Analysis. *Proc. ACM Program. Lang.* 8, PLDI, Article 226 (June 2024), 25 pages. <https://doi.org/10.1145/3656456>
- Ariel E. Kellison, Laura Zielinski, David Bindel, and Justin Hsu. 2025. Bean: A Language for Backward Error Analysis. *Proc. ACM Program. Lang.* 9, PLDI, Article 221 (June 2025), 25 pages. <https://doi.org/10.1145/3729324>
- 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> arXiv:https://doi.org/10.1137/0715024

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. *ACM Trans. Program. Lang. Syst.* 41, 1, Article 2 (Dec. 2018), 39 pages. <https://doi.org/10.1145/3230733>

Tanmay Tirpankar, Arnab Das, and Ganesh Gopalakrishnan. 2025. Satire: Computing Rigorous Bounds for Floating-Point Rounding Error in Mixed-Precision Loop-Free Programs. (2025). arXiv:2503.05924 [cs.PL] <https://arxiv.org/abs/2503.05924>

## 10 METRIC PRESERVATION PROOF

**THEOREM 10.1 (METRIC PRESERVATION).** *For any  $\Delta \mid \Gamma \vdash e : \tau$  and substitutions  $\sigma, \sigma'$  such that  $\sigma \sim_\gamma \sigma' : \Delta \mid \Gamma$ , then  $e \sigma \sim_{\gamma \cdot (\Delta \mid \Gamma)} e \sigma' : \tau$ .*

**PROOF.** We induct over our typing derivation. The base cases (Var, **bvar**, Unit, Const) follow trivially. The subsumption and widening cases also follow trivially. We detail the remaining cases here. Note that the distance between **bnd** vars in  $\Delta$  is  $\infty$  if they correspond to different bounds. Therefore for the following cases, we omit repetitively reasoning about  $\Delta$  substitutions:

**Case  $\rightarrow$  I.** We wish to show that for any  $\Gamma \vdash \lambda x.e : \tau$  and substitutions  $\sigma \sim_\gamma \sigma' : \Gamma$  that  $\lambda x.e \sigma \sim_{\gamma \cdot \Gamma} \lambda x.e \sigma'$ . Unfolding, it suffices to show that both:

- (1)  $\lambda x.e \sigma, \lambda x.e \sigma'$  are in  $\mathcal{R}_{\tau_0 \rightarrow \tau}$
- (2)  $\mathcal{SD}_{\tau_0 \rightarrow \tau}(\lambda x.e \sigma, \lambda x.e \sigma') \leq \gamma \cdot \Gamma$

Observe that we have by our inductive hypothesis that for any  $\Delta, x : \tau' \vdash e : \tau$  and substitutions  $\delta[v_0/x] \sim_{\gamma'} \delta'[v_1/x] : \Gamma, x : \tau'$  that  $e \delta[v_0/x] \sim_{\gamma' \cdot \Gamma} e \delta'[v_1/x]$  holds. Let us now carry on with the proof:

**Property 1.** We need to show that  $(\lambda x.e) \sigma$  and  $(\lambda x.e) \sigma'$  are both in the relation  $\mathcal{R}_{\tau_0 \rightarrow \tau}$ .

The cases are symmetric so we only show one case. Unfolding the definition of  $\mathcal{R}_{\tau_0 \rightarrow \tau}$ , let  $w_0, w_1 \in \mathcal{VR}_{\tau_0}$ .  $(\lambda x.e)w_0 \mapsto e[w_0/x]$  and  $(\lambda x.e)w_1 \mapsto e[w_1/x]$  and so it suffices to show that  $e \sigma[w_0/x], e \sigma[w_1/x]$  are closed expressions falling in  $\mathcal{R}_{\tau_0}$ . This is true by our inductive hypothesis, instantiating with  $\sigma = \delta = \delta', v_0 = w_0$ , and  $v_1 = w_1$ .

**Property 2.** Unfolding the definition of  $\mathcal{SD}_{\tau_0 \rightarrow \tau}$ , it suffices to show that:

$$\mathcal{SD}_\tau((\lambda x.e) \sigma, (\lambda x.e) \sigma') \leq \gamma \cdot \Gamma$$

Stepping, it suffices to show that

$$\mathcal{SD}_\tau(e \sigma[w/x], e \sigma'[w/x]) \leq \gamma \cdot \Gamma$$

which holds by application of our inductive hypothesis when  $\delta = \sigma, \delta' = \sigma', v_0 = v_1 = w$ , and  $\gamma' = \gamma :: 0$ .

So, by our inductive hypothesis and unfolding the definitions of  $\mathcal{R}$  and  $\mathcal{SD}$ , we have properties (1) and (2) respectively.

**Case  $\rightarrow$  E.** By our inductive hypothesis, it suffices to prove this case for  $e f$ . We wish to show that for any  $\Gamma + \Theta \vdash e f : \tau$  and substitutions  $\sigma \sim_\alpha \sigma' : \Gamma + \Theta$  that

$$e f \sigma \sim_{\alpha \cdot (\Gamma + \Theta)} e f \sigma'$$

Using Lemma 3.20 (substitution decomposition), we construct potentially overlapping substitutions  $\sigma_0 \sim_\alpha \sigma_1 : \Gamma$  and  $\sigma'_0 \sim_\alpha \sigma'_1 : \Theta$  where  $FV(e) \subseteq \text{DOM}(\sigma_0) = \text{DOM}(\sigma_1)$  and  $FV(f) \subseteq \text{DOM}(\sigma'_0) = \text{DOM}(\sigma'_1)$  and  $\sigma_0, \sigma'_0 \subseteq \sigma$  and  $\sigma_1, \sigma'_1 \subseteq \sigma'$  and  $\gamma, \theta$  minimal. In other words,  $\sigma_0, \sigma_1$  correspond to the parts of  $\sigma$  and  $\sigma'$  respectively that are represented by  $\Gamma$ . Similarly,  $\sigma'_0, \sigma'_1$  correspond to the parts of  $\sigma$  and  $\sigma'$  respectively that are represented by  $\Theta$ . Note that our substitutions *must* overlap in the case where  $FV(e) \cap FV(f)$  is non-empty. By our inductive hypothesis, we know that  $e \sigma \mapsto^* \lambda x.M$  and  $e' \sigma \mapsto^* \lambda x.M'$  for some  $M, M'$ . Since substituting variables that are not free does not change the underlying term,

we know that the following equations must hold where  $\sigma^*, \sigma'^*$  are  $\sigma$  and  $\sigma'$  respectively with  $x$  removed:

$$\begin{aligned} (\lambda x.M) [\sigma_0] (w[\sigma_1]) &= ((\lambda x.M[\sigma^*]) w)[\sigma] = ((\lambda x.M) w)[\sigma] \mapsto M[\sigma][w/x] \\ (\lambda x.M') [\sigma'_0] (w[\sigma'_1]) &= ((\lambda x.M'[\sigma'^*]) w)[\sigma'] = ((\lambda x.M') w)[\sigma'] \mapsto M'[\sigma'][w/x] \end{aligned} \quad (24)$$

and by stepping

$$\begin{aligned} (\lambda x.M) [\sigma_0] (w[\sigma_1]) &\mapsto M[\sigma_0][w[\sigma_1]/x] \\ (\lambda x.M) [\sigma'_0] (w[\sigma'_1]) &\mapsto M'[\sigma'_0][w[\sigma'_1]/x] \end{aligned} \quad (25)$$

so by deterministic stepping

$$\begin{aligned} M[\sigma_0][w[\sigma_1]/x] &= M[\sigma][w/x] \\ M'[\sigma'_0][w[\sigma'_1]/x] &= M'[\sigma'][w/x] \end{aligned} \quad (26)$$

By applying our inductive hypothesis and unfolding our definition of  $R_{\tau_0 \rightarrow \tau}$ , we know that  $M, M'$  is 1-sensitive with respect to  $w$  according to  $\mathcal{SD}$ . This gives us:

$$\mathcal{SD}_\tau(M[w[\sigma]/x][\sigma], M[w[\sigma']/x][\sigma]) \leq \mathcal{SD}_{\tau_0}(w[\sigma], w[\sigma']) \leq \theta \cdot \Theta$$

and we also know that by our inductive hypothesis

$$\mathcal{SD}_\tau(e[\sigma] f[\sigma'], e[\sigma'] f[\sigma']) = \mathcal{SD}_\tau(M[w[\sigma']/x][\sigma], M'[w[\sigma']/x][\sigma']) \leq \gamma \cdot \Gamma$$

So by our triangle inequality of the syntactic distance between terms:

$$M[w[\sigma]/x] \sigma \sim_{\gamma \cdot \Gamma + \theta \cdot \Theta} M'[w[\sigma']/x] \sigma'$$

and therefore

$$(\lambda x.M) w \sigma \sim_{\gamma \cdot \Gamma + \theta \cdot \Theta} (\lambda x.M') w \sigma'$$

To complete the proof case, it suffices to prove

$$\alpha \cdot (\Gamma + \Theta) \geq \alpha \cdot \Gamma + \alpha \cdot \Theta$$

which holds by construction (via our substitution decomposition lemma).

**Case  $M_q e$  (let-bind).** We prove each property separately.

**Property 1.** Like the  $\rightarrow I$  case, we only need to prove one side due to symmetry. So we fix a substitution  $\sigma$ . By Lemma 3.21, it suffices to prove this case for a value  $(v_0, v_1)$ . We have by our inductive hypothesis that  $(v_0, v_1) \in \mathcal{R}_{M_r \tau_0}$ . So  $\mathcal{SD}(v_0, v_1) \leq r$ . By application of our inductive hypothesis, we know that (abusing notation)  $f[v_1/x], f[v_2/x] \in \mathcal{R}_{M_q}$  so  $f[v_1/x] \mapsto^* (v_3, v_4), f[v_2/x] \mapsto^* (v_5, v_6)$  where  $(v_3, v_4), (v_5, v_6) \in \mathcal{R}_{M_q}$ . We also know that

$$\sigma[v_0/x] \sim_{\vec{0}, r} \sigma[v_1/x] : \Gamma, x : \tau_0$$

that

$$f \sigma[v_0/x] \sim_{(\vec{0}, r) \cdot (\Gamma, x \mapsto s)} f \sigma[v_1/x] : M_q \tau$$

So therefore

$$\begin{aligned} (v_3, v_4) \sigma[v_1/x] &\sim_{(\vec{0}, r) \cdot (\Gamma, x \mapsto s)} (v_5, v_6) \sigma[v_2/x] : M_q \tau \\ (v_3, v_4) \sigma[v_1/x] &\sim_{r \cdot s} (v_5, v_6) \sigma[v_2/x] : M_q \tau \end{aligned} \quad (27)$$

So clearly  $(v_3, v_6) \in \mathcal{R}_{M_s, r+q}$  by application of triangle inequality:

$$r \cdot s + q \geq \mathcal{SD}_{\text{tau}}(v_3, v_5) + \mathcal{SD}_\tau(v_5, v_6) \geq \mathcal{SD}_\tau(v_3, v_6)$$

**Property 2.** We need to show that for all substitutions  $\sigma \sim_{\alpha} \sigma' : s \cdot \Gamma + \Theta$ :

$$\text{let-bind } x = e \text{ in } f \sigma \sim_{\alpha \cdot (s \cdot \Gamma + \Theta)} \text{let-bind } x = e \text{ in } f \sigma' : M_{s \cdot r + q} \quad (28)$$

By Lemma 3.20 (substitution decomposition) we have substitutions  $\sigma_Y \sim_{\alpha} \sigma'_Y : s \cdot \Gamma$  and  $\sigma_{\theta} \sim_{\alpha} \sigma'_{\theta} : \Theta$ .

$$e \sigma_Y \sim_{\alpha \cdot \Gamma} e \sigma'_Y : M_r \tau_0$$

By our proof of **Property 1**, we know that both sides of Equation 28 are in the relation  $\mathcal{R}_{M_{r \cdot s + q} \tau}$  and we're not stuck. We want to show both sides have syntactic distance less than  $\alpha \cdot (s \cdot \Gamma + \Theta)$ . We apply our inductive hypothesis once more and extend the  $\theta$  substitutions to obtain:

$$f \sigma_{\theta}[e \sigma_Y/x] \sim_{\alpha \cdot \Theta + (\alpha \cdot s \cdot \Gamma)} f \sigma'_{\theta}[e \sigma'_Y/x] : M_r \tau \quad (29)$$

Since both sides are neighborhood monads, it suffices to bound distances in only the first components of each side which means that Equation 29 implies Equation 28.

**Case (let-cobind).** We prove each property separately.

**Property 1.** We again only need to prove one side due to symmetry. By Lemma 3.21, it suffices to prove this case for a value  $[v]$   $\sigma$  for a  $\sigma$  compatible with  $\Gamma$ . We have by our inductive hypothesis that  $[v] \sigma \in \mathcal{R}_{!_s \tau_0}$ . Unfolding, we have that  $v \sigma \in \mathcal{R}_{\tau_0}$ . Stepping and applying our inductive hypothesis yields that  $f \sigma [v/x] \in \mathcal{R}_{\tau}$ .

**Property 2.** We need to show that for all substitutions  $\sigma \sim_{\alpha} \sigma' : t \cdot \Gamma + \Theta$ :

$$\text{let-cobind } x = e \text{ in } f \sigma \sim_{\alpha \cdot (t \cdot \Gamma + \Theta)} \text{let-cobind } x = e \text{ in } f \sigma' : \tau$$

By Lemma 3.20 (substitution decomposition) we have substitutions  $\sigma_Y \sim_{\alpha} \sigma'_Y : \Gamma$  and  $\sigma_{\theta} \sim_{\alpha} \sigma'_{\theta} : \Theta$ . By application of the inductive hypothesis, we know that:

$$e \sigma_Y \sim_{\alpha \cdot \Gamma} e \sigma'_Y : !_s \tau_0$$

where

$$e \sigma_Y = [v_0]$$

and

$$e \sigma'_Y : !_s \tau_0 = [v_1]$$

for some  $v_0, v_1$  so therefore by inspection of the relation:

$$s \cdot \mathcal{SD}_{\tau_0}(v_0, v_1) = \mathcal{SD}_{!_s \tau_0}([v_0], [v_1])$$

Let  $\mathcal{SD}_{\tau_0}(v_0, v_1) = c$ . Importantly, note that

$$s \cdot c = \alpha \cdot \Gamma \quad (30)$$

Then, we apply our inductive hypothesis once more and we extend the  $\theta$  substitutions and to obtain:

$$f \sigma_{\theta}[\sigma_Y/x] \sim_{\alpha \cdot \Theta + t \cdot s \cdot c} f \sigma'_{\theta}[\sigma'_Y/x] : \tau$$

Note that the distance  $\sim$  is obtained via the inductive hypothesis and the fact that it is in  $R_{M_{s \cdot r + q}}$  is obtained from our proof of **Property 1**. Substituting by Equation 30 we complete the case:

$$f \sigma_{\theta}[\sigma_Y/x] \sim_{\alpha \cdot \Theta + t \cdot \alpha \cdot \Gamma} f \sigma'_{\theta}[\sigma'_Y/x] : \tau$$

**Case (let).** We prove each property separately.

**Property 1.** We apply Lemma 3.21. By symmetry, it suffices to only prove one side.

Follows by application of our inductive hypothesis and some stepping.

**Property 2.** We need to show that for all substitutions  $\sigma \sim_{\alpha} \sigma' : s \cdot \Gamma + \Theta$ :

$$\text{let } x = e \text{ in } f \sigma \sim_{\alpha \cdot (s \cdot \Gamma + \Theta)} \text{let } x = e \text{ in } f \sigma' : \tau$$

By Lemma 3.20 we have substitutions  $\sigma_{\gamma} \sim_{\alpha} \sigma'_{\gamma} : \Gamma$  and  $\sigma_{\theta} \sim_{\alpha} \sigma'_{\theta} : \Theta$ . By applying our inductive hypothesis, we know that

$$e \sigma_{\gamma} \sim_{\alpha \cdot \Gamma} e \sigma'_{\gamma} : \tau_0$$

So we can extend our pair of  $\Theta$  substitutions by these two terms respectively and apply our inductive hypothesis yielding:

$$\text{let } x = e \text{ in } f \sigma \sim_{(\alpha \cdot s \cdot \Gamma) + (\alpha \cdot \Theta)} \text{let } x = e \text{ in } f \sigma' : \tau$$

which is equivalent to what we wanted to show.

**Case factor.** By Lemma 3.21, it suffices to prove this case for **factor**  $((v_1, v_2), (v_3, v_4))$ , **factor**  $((v'_1, v'_2), (v'_3, v'_4))$ . Unfolding our inductive hypothesis, we have that for substitutions  $\sigma \sim_{\gamma} \sigma' : \Gamma$ , that

$$\text{factor}((v_1, v_2), (v_3, v_4)) \sigma \sim_{\gamma \cdot \Gamma} \text{factor}((v'_1, v'_2), (v'_3, v'_4)) \sigma' : (M_q \tau_0) \times (M_r \tau_1)$$

Unfolding, we get:

$$\text{factor}((v_1 \sigma, v_2 \sigma), (v_3 \sigma, v_4 \sigma)) \sim_{\gamma \cdot \Gamma} \text{factor}((v'_1 \sigma', v'_2 \sigma'), (v'_3 \sigma', v'_4 \sigma')) : (M_q \tau_0) \times (M_r \tau_1)$$

So by stepping (and reassociating) we can see that the following logical relation holds as distances between neighborhood monads are measured in terms of the first components:

$$((v_1 \sigma, v_3 \sigma), (v_2 \sigma, v_4 \sigma)) \sim_{\gamma \cdot \Gamma} ((v'_1 \sigma', v'_3 \sigma'), (v'_2 \sigma', v'_4 \sigma')) : M_{\max(r, q)}(\tau_0 \times \tau_1)$$

**Case rnd.** We prove each property separately.

**Property 1.** We apply Lemma 3.21. The case holds under the assumption the constant language parameter  $u$  has the required properties (see Definition 3.5, property a) and application of our inductive hypothesis.

**Property 2.** Since distance is measured for the neighborhood monad on the first component, which in our case is the ideal (not rounded) component (see stepping rule), this holds trivially by application of the inductive hypothesis.

**Case ret.** We prove each property separately.

**Property 1.** We apply Lemma 3.21. Holds trivially by application of the inductive hypothesis.

**Property 2.** Since distance is measured for the neighborhood monad on the first component, which in our case is the ideal (not rounded) component (see stepping rule), this holds trivially by application of the inductive hypothesis.

**Case op.** Both properties hold by our language interface (Definition 3.5, property b) and application of the inductive hypothesis.

**Case ! I.** Holds unfolding of our logical relations, and our inductive hypothesis.

**Case  $\times$  I.** We prove each property separately.

**Property 1.** Holds by application of the inductive hypothesis and unfolding of the definition of  $\mathcal{R}_{\tau_0 \times \tau_1}$ .

**Property 2.** We have  $\sigma \sim_{\alpha} \sigma' : \Gamma + \Theta$  and wish to show that

$$(e, f) \sigma \sim_{\alpha \cdot (\Gamma + \Theta)} (e, f) \sigma'$$

By Lemma 3.20, we know that there exists substitutions  $\sigma_{\Gamma} \sim_{\alpha} \sigma'_{\Gamma} : \Gamma$  and  $\sigma_{\Theta} \sim_{\alpha} \sigma'_{\Theta} : \Theta$ . So by definition unfolding we have that:

$$(v \sigma_{\Gamma}, w \sigma_{\Theta}) \sim_{(\alpha \cdot \Gamma + \alpha \cdot \Theta)} (v \sigma'_{\Gamma}, w \sigma'_{\Theta}) : \tau_0 \times \tau_1$$

so therefore by an analysis of free variables and substitution

$$(v, w) \sigma \sim_{(\alpha \cdot \Gamma + \alpha \cdot \Theta)} (v, w) \sigma' : \tau_0 \times \tau_1$$

To complete the proof case, it suffices to prove:

$$\alpha \cdot (\Gamma + \Theta) \geq \alpha \cdot \Gamma + \alpha \cdot \Theta$$

which holds by construction (via our substitution decomposition lemma).

**Case  $\times$  E.** We prove each property separately.

**Property 1.** We apply Lemma 3.21. The case holds by application of the inductive hypothesis and stepping once.

**Property 2.** It suffices to show that for all  $\sigma \sim_Y \sigma' : \Gamma$  if  $e \sigma \sim_{Y \cdot \Gamma} e \sigma' : \tau_1 \times \tau_2$  then:

$$\pi_i e \sigma \sim_{Y \cdot \Gamma} \pi_i e \sigma' : \tau_i$$

which holds by stepping and some unfolding of the logical relation.

**Case  $\otimes$  I.** We prove each property separately.

**Property 1.** We apply Lemma 3.21. The case holds by application of the inductive hypothesis.

**Property 2.** It suffices to show that for all  $\sigma \sim_\alpha \sigma' : \Gamma + \Theta$ :

$$(e, f) \sigma \sim_{\alpha \cdot (\Gamma + \Theta)} (e, f) \sigma' : \tau_0 \otimes \tau_1$$

By Lemma 3.20, we know that  $\sigma \sim_\alpha \sigma' : \Gamma$  and  $\sigma \sim_\alpha \sigma' : \Theta$ . Applying our inductive hypothesis, we get that  $e \sigma \sim_{\alpha \cdot \Gamma} e \sigma' : \tau_0$  and  $e \sigma \sim_{\alpha \cdot \Theta} e \sigma' : \tau_1$  so by unfolding our logical relation we obtain

$$(e, f) \sigma \sim_{\alpha \cdot \Gamma + \alpha \cdot \Theta} (e, f) \sigma' : \tau_0 \otimes \tau_1$$

which is equivalent to what we wanted to show.

**Case  $\otimes$  E.** We prove each property separately.

**Property 1.** We apply Lemma 3.21. We again only need to prove one side due to symmetry. Let our bound expression be some value  $w$ . By our inductive hypothesis  $w = (v_0, v_1)$  for some  $v_0$  and  $v_1$ . Stepping and applying our inductive hypothesis yields that  $f \sigma [v_0/x][v_1/x] \in \mathcal{R}(\tau)$  for any  $\sigma$  compatible with  $\Theta$ .

**Property 2.** This case mirrors the proof for **Property 2** of the **let** case.

**Case  $+$  I.** We prove each property separately.

**Property 1.** We apply Lemma 3.21. Holds by application of the inductive hypothesis.

**Property 2.** It suffices to show that for all  $\sigma \sim_Y \sigma' : \Gamma$  if  $e \sigma \sim_{Y \cdot \Gamma} e \sigma' : \tau_0 + \tau_1$  then:

$$\mathbf{in}_i e \sigma \sim_{Y \cdot \Gamma} \mathbf{in}_i e \sigma' : \tau_0 + \tau_1$$

which holds by stepping and some unfolding of the logical relation.

**Case  $+$  E.** We prove each property separately.

**Property 1.** We apply Lemma 3.21. We again only need to prove one side due to symmetry. Let our bound expression be some value  $\mathbf{in}_i v$ . Stepping and applying our inductive hypothesis finishes this case.

**Property 2.** We wish to show that for all  $\sigma \sim_\alpha \sigma' : s \cdot \Gamma + \Theta$  that:

$$\mathbf{case } e \text{ of } (\mathbf{in}_1 x.f_1 \mid \mathbf{in}_2 x.f_2) \sigma \sim_{\alpha \cdot (s \cdot \Gamma + \Theta)} \mathbf{case } e \text{ of } (\mathbf{in}_1 x.f_1 \mid \mathbf{in}_2 x.f_2) \sigma' : \tau$$

By Lemma 3.20 we have substitutions

$$\sigma_\Gamma \sim_\alpha \sigma'_\Gamma : \Gamma$$

$$\sigma_\Theta \sim_\alpha \sigma'_\Theta : \Theta$$

that we can plug into our inductive hypothesis. Plugging in, we know that:  $e \sigma \mapsto \mathbf{in}_i v$  and  $e \sigma' \mapsto \mathbf{in}_j v'$  where  $\mathcal{SD}(\mathbf{in}_i v, \mathbf{in}_j v') \leq \gamma \cdot \Gamma$ . We now case over the following two scenarios:

**Subcase  $i = j$ .** This subcase mirrors the proof for **Property 2** of the **let** case.

**Subcase  $i \neq j$ .** In this case, we have:  $\mathcal{SD}(\mathbf{in}_i v, \mathbf{in}_j v') = \infty \leq \alpha \cdot \Gamma$ . Stepping and applying the inductive hypothesis, we obtain:

$$f_i[v/x]\sigma \sim_{\alpha \cdot (s \cdot \Gamma + \Theta)} f_j[v'/x]\sigma' : \tau$$

Since  $s$  is non-zero, by applying our inductive hypothesis we know that  $f_i$  and  $f_j$  are non-zero sensitive in  $x$  and since positive  $s \cdot \infty = \infty$ , we know that  $\alpha \cdot (s \cdot \Gamma + \Theta) = \infty$ . Therefore this subcase holds. Note that it is essential in this case that  $s$  is non-zero; If  $s$  is allowed to be zero, this would be a problem because  $0 \cdot \infty = 0$ .

The remaining cases in this proof deal with bound polymorphism:

**Case bop.** Holds by the spec on our language interface and from the fact that any differing  $\Delta$ -substitutions must be  $\infty$  apart.

**Case  $\forall \mathbf{I}$ .** We need to show that for any substitutions:

$$\sigma \sim_{\alpha} \sigma' : \Delta \mid \Gamma$$

that

$\Lambda \epsilon.e \sigma \sim_{\alpha \cdot (\Delta \mid \Gamma)} \Lambda \epsilon.e \sigma' : \forall \epsilon.\tau$ . Similarly to the above cases, we observe that if the  $\Delta$ -substitutions in  $\sigma$  and  $\sigma'$  are different, the distance is  $\infty$  and we are done. In the case that the  $\Delta$ -substitutions are the same, we unfold our definition of  $\mathcal{R}_{\forall \epsilon.\tau}$  and observe that since  $\epsilon \notin FTV(\Gamma)$ , we can apply our inductive hypothesis.

**Case  $\forall \mathbf{E}$ .** We similarly observe that if the  $\Delta$ -substitutions in  $\sigma$  and  $\sigma'$  are different, the distance is  $\infty$  and we are done. Unfolding our logical relation and applying our inductive hypothesis finishes the case.

□