TYPE-BASED APPROACHES TO ROUNDING ERROR ANALYSIS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Ariel Eileen Kellison

December 2024

TYPE-BASED APPROACHES TO ROUNDING ERROR ANALYSIS

Ariel Eileen Kellison, Ph.D.

Cornell University 2024

This dissertation explores the design and implementation of programming languages that represent rounding error analysis through typing. A rounding error analysis establishes an *a priori* bound on the rounding errors introduced by finite-precision arithmetic in a numerical program, providing a measure of the quality of the program as an approximation to its ideal, infinitely precise counterpart. This can be achieved by measuring the distance between the computed output and the ideal result, known as the forward error, or by determining the smallest distance by which the input must be perturbed to make the computed output match the ideal result, known as the backward error. Due to the complex interaction between the rounding errors produced locally by a program and those propagated from its inputs, performing a rounding error analysis is known to be challenging. Moreover, while most programs can be shown to have a forward error bound, even if trivially large, many programs cannot be shown to have a backward error bound. This makes the task of deriving backward error bounds for complex programs even more challenging than the corresponding task of deriving forward error bounds.

One way to view programs that use finite-precision arithmetic is as computations that *produce* rounding error, while also *consuming* and *amplifying* rounding errors passed to them as inputs from other functions. This perspective aligns closely with the notions of *effects* (what programs produce aside from values) and *coeffects* (how programs use their inputs). Both effectful and coeffectful program behaviors can be analyzed using type-based approaches, where graded monadic types track effects, and graded comonadic types track coeffects. While recent work has studied the interaction of effects and coeffects in various domains, the work presented in this dissertation is the first to investigate these behaviors in the context of numerical analysis and the propagation of rounding errors in numerical programs.

The first part of this dissertation demonstrates that it is possible to design languages for forward error analysis, as illustrated with **NumFuzz**, a functional programming language whose type system expresses quantitative bounds on rounding error. This type system combines a sensitivity analysis, enforced through a linear typing discipline, with a novel graded monad to track the accumulation of rounding errors. We establish the soundness of the type system by relating the denotational semantics of the language to both an exact and floating-point operational semantics. To illustrate the capabilities of our system, we instantiate **NumFuzz** with error metrics from the numerical analysis literature and incorporate rounding operations that accurately model key aspects of the IEEE 754 floating-point standard. Furthermore, to demonstrate the practical utility of **NumFuzz** as a tool for automated error analysis, we have developed a prototype implementation capable of inferring error bounds. This implementation produces bounds competitive with existing tools, while often achieving significantly faster analysis times.

The second part of this dissertation explores a type-based approach to backward error analysis with **Bean**, a first-order programming language with a linear type system that can express quantitative bounds on backward error. **Bean**'s type system combines a graded coeffect system with strict linearity to soundly track the flow of backward error through programs. To illustrate **Bean**'s potential as a practical tool for automated backward error analysis, we implement a variety of standard algorithms from numerical linear algebra in **Bean**, establishing fine-grained backward error bounds via typing in a compositional style. We also develop a prototype implementation of **Bean** that infers backward error bounds automatically. Our evaluation shows that these inferred bounds match worst-case theoretical relative backward error bounds from the literature, underscoring **Bean**'s utility in validating a key property of numerical programs: *numerical stability*.

BIOGRAPHICAL SKETCH

Ariel E. Kellison is a computer scientist who works on formal methods, with a focus on the formal verification of numerical software. She received her undergraduate degree in astrophysics (with honors in the major) from the University of California, Santa Cruz, in 2010. After graduation, she worked as a high-school mathematics teacher, where she developed a strong interest in proof and computation. In 2016, she joined the Nuprl research group at Cornell University, and began her PhD in computer science at Cornell in 2020. During her doctoral studies, she was supported by the Department of Energy Computational Science Graduate Fellowship and interned with the Digital Foundations and Mathematics Group at Sandia National Laboratories.

ACKNOWLEDGEMENTS

The work presented in this dissertation is the result of years of patient guidance and constant encouragement from my academic mentors: David Bindel, Andrew Appel, and Justin Hsu. David encouraged me to follow my curiosity in using ideas from formal methods to solve problems in numerical analysis, while Andrew and Justin patiently taught me the technical tools to explore and develop those ideas further. I am truly grateful for the time each of them has taken to teach me countless lessons about reading, writing, and computer arithmetic.

The core of this dissertation builds on papers co-authored with David Bindel, Justin Hsu, and Laura Zielinski; Appendix B.7, which describes the type-checking algorithm for **Bean**, is the work of Laura Zielinski.

Beyond my primary mentors, many others have shaped the work presented in this dissertation through their mentorship and collaboration. Bob Constable first introduced me to formal methods, and his enthusiasm for type theory continues to inspire me today. It is hard to say what my PhD research would have looked like without the influence of Bob and the rest of the Nuprl research group, including Mark Bickford, Richard Eaton, and Liron Cohen.

At Sandia National Laboratories, members of the formal methods group provided invaluable expertise. Geoffrey Hulette first directed me towards foundational approaches to verifying floating-point programs, Sam Pollard provided friendly expertise on the nuances of floating-point arithmetic, and Heidi Thornquist grounded our speculative discussions with practical, applications-focused insights.

I am also grateful for collaborations with Jean-Baptiste Jeannin and Mohit Tekriwal. Our work on formally verifying numerical algorithms in Coq led to many rewarding QEDs and helped solidify my understanding of the IEEE Standard.

TABLE OF CONTENTS

Biographical Sketch . . . iii
Acknowledgements . . . iv
Table of Contents . . . v

1 Introduction **1**
1.1 Rounding Error and Program Distances . . . 2
1.2 Type-Based Approaches to Rounding Error Analysis . . . 5
1.3 Goals and Contributions . . . 7
1.4 Dissertation Outline . . . 11

2 Background **13**
2.1 Floating-Point Arithmetic . . . 13
2.1.1 Rounding Functions . . . 14
2.1.2 Models of Floating-Point Arithmetic . . . 16
2.1.3 Measures of Accuracy . . . 17
2.2 Type Systems . . . 18
2.2.1 Effects and Graded Monadic Types . . . 20
2.2.2 Coeffects and Graded Comonadic Types . . . 22
2.2.3 Categorical Semantics . . . 24

3 A Language for Forward Error Analysis **28**
3.1 Introduction . . . 28
3.2 Type System . . . 34
3.2.1 Types . . . 34
3.2.2 Values and Terms . . . 35
3.2.3 Typing Relation . . . 37
3.3 Operational Semantics . . . 40
3.4 Denotational Semantics . . . 43
3.4.1 The Category of Metric Spaces . . . 43
3.4.2 A Graded Comonad on Met . . . 45
3.4.3 A Graded Monad on Met . . . 46
3.4.4 Interpreting **NumFuzz** . . . 50
3.5 Forward Error Soundness . . . 55
3.6 Example **NumFuzz** Programs . . . 57
3.7 Implementation and Evaluation . . . 68
3.7.1 Implementation . . . 68
3.7.2 Evaluation . . . 72
3.8 Related Work . . . 78
3.9 Conclusion . . . 82

4 A Language for Backward Error Analysis **86**
4.1 Introduction . . . 86
4.1.1 Backward Error Analysis in **Bean**: Motivating Examples . . . 90
4.2 Type System . . . 93
4.2.1 Types . . . 94
4.2.2 Typing Judgments . . . 94
4.2.3 Expressions . . . 95
4.2.4 Typing relation . . . 96
4.3 Denotational Semantics . . . 98
4.3.1 **Bel**: The Category of Backward Error Lenses . . . 98
4.3.2 Basic Constructions in **Bel** . . . 101
4.3.3 Interpreting **Bean** . . . 107
4.4 Backward Error Soundness . . . 112
4.4.1 Λ_S: A Language for Projecting **Bean** into **Set** . . . 114
4.4.2 Interpreting Λ_S . . . 116
4.5 Example **Bean** Programs . . . 119
4.6 Implementation and Evaluation . . . 125
4.6.1 Implementation . . . 125
4.6.2 Evaluation . . . 126
4.7 Related Work . . . 131
4.8 Conclusion . . . 135

A Appendix for NumFuzz **156**
A.1 Termination (Strong Normalization) . . . 156
A.2 The Neighborhood Monad . . . 158
A.3 Interpreting **NumFuzz** Terms . . . 160
A.4 Denotational Semantics . . . 164

B Appendix for Bean **168**
B.1 The Category of Backward Error Lenses . . . 168
B.2 Basic Constructions in **Bel** . . . 169
B.2.1 Tensor Product . . . 170
B.2.2 Coproducts . . . 175
B.3 Interpreting **Bean** Terms . . . 177
B.4 Interpreting Λ_S Terms . . . 185
B.5 Details for Soundness . . . 188
B.6 Proof of Backward Error Soundness . . . 199
B.7 Type Checking Algorithm . . . 201

CHAPTER 1

INTRODUCTION

O dear Ophelia!
I am ill at these numbers:
I have not art to reckon my groans.

Hamlet (Act II, Scene 2, Line 120)

Many fields of computing are concerned with designing algorithms for solving problems in continuous mathematics. These algorithms are usually specified using real numbers but are implemented in a finite-precision arithmetic, like floating-point arithmetic. This approximation leads to rounding errors which can degrade the accuracy of results. When accuracy guarantees are required, a *rounding error analysis* can provide an *a priori* bound on the rounding error in a floating-point result. There are many examples from scientific and engineering domains where this type of analysis is essential:

- **Solid modeling:** In computer-aided design (CAD) and computer graphics, rounding error bounds are used to guarantee the correctness of computer representations and manipulations of geometric shapes (Sherman et al., 2019; Hu et al., 1996).
- **Secure multiparty computation:** In encrypted signal processing, floating-point arithmetic is used to process signals that might be real-valued; a rigorous rounding error analysis guarantees that implementations will not leak sensitive information by producing exceptional values (Kamm and Willemson, 2015; Franz and Katzenbeisser, 2011; Aliasgari et al., 2013).
- **Numerical linear algebra:** In numerical linear algebra, specifications of basic operations like the dot product and matrix multiplication are often accompanied by a rounding error analysis describing their expected accuracy and stability (Anderson et al., 1999; Blackford et al., 2002; Li et al., 2002).
- **Zero-knowledge proofs for machine learning:** In neural-network inference, zero-knowledge

proofs guarantee that sensitive data remain secret (Chen et al., 2020; Weng et al., 2021); a rounding error analysis is central to the development of efficient methods for constructing these proofs (Garg et al., 2022).

However, while accuracy guarantees are needed in many areas of computing, there are very few tools available to help programmers construct them. In the examples listed above, the rounding error analyses are done by hand. This process requires specialized knowledge of how to reason about the subtle details of floating-point arithmetic, and becomes increasingly impractical as programs grow larger, rely on mixed precision computations, include dependencies on multiple modules, or use foreign function interfaces.

In this dissertation, we propose a novel approach to developing numerical programs with accuracy guarantees: designing *languages* that unify the tasks of writing programs and reasoning about their accuracy. Our thesis is that it is feasible to design and implement languages that represent rounding error analysis through typing, and that these languages have the potential to make reasoning about numerical accuracy convenient for programmers.

1.1 Rounding Error and Program Distances

We begin by briefly introducing some of the fundamental concepts related to designing languages for rounding error analysis. Suppose $\mathbf{P}$ is an algorithm specified on real numbers and $\tilde{\mathbf{P}}$ is an implementation of $\mathbf{P}$ that uses floating-point arithmetic, which might introduce rounding errors during computation. We can think of $\mathbf{P}$ and $\tilde{\mathbf{P}}$ as being non-equivalent but closely related programs: every floating-point operation in $\tilde{\mathbf{P}}$ approximates its real valued counterpart in $\mathbf{P}$ with an accuracy that depends on the number of bits in the floating-point format, as well as the rounding strategy that is used. The purpose of a rounding error analysis is to quantitatively determine how close $\mathbf{P}$ and $\tilde{\mathbf{P}}$ are by deriving an *a priori* bound on the effects of rounding errors. In numerical analysis, the

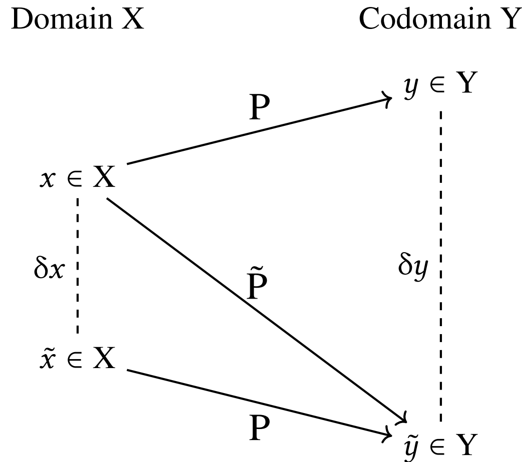


Figure 1.1: Forward and backward error.

notion of the "closeness" of programs—or *program distance*—can be determined by performing either a *forward* or *backward* error analysis (Higham, 2002).

Forward Error Analysis

Even if the programs $\mathbf{P}$ and $\tilde{\mathbf{P}}$ are closely related, they might produce different outputs when given the same input. This behavior leads to the following natural question: for a given input x, how close is the output $\tilde{\mathbf{P}}(x)$ to the target output $\mathbf{P}(x)$? The distance between $\tilde{y} = \tilde{\mathbf{P}}(x)$ and $y = \mathbf{P}(x)$ is known as the *forward error*, and a bound on the forward error is obtained by performing a *forward error analysis*. The forward error is represented by the quantity δy in Figure 1.1.

Accurate programs are typically characterized by having a small forward error. Unfortunately, automated tools that statically compute sound *a priori* bounds on forward errors suffer from significant limitations. These tools tend to overestimate rounding errors, are mostly restricted to analyzing straight-line programs (those without loops or conditional branches), and struggle to scale to larger programs with more than a few hundred floating-point operations. Another significant limitation of many of these tools is their inability to account for how programs amplify and propagate

rounding errors from their inputs—a critical factor for analyzing and scaling to realistic software [1].

Backward Error Analysis

We can also compare $\mathbf{P}$ and $\tilde{\mathbf{P}}$ by answering the following question: do $\mathbf{P}$ and $\tilde{\mathbf{P}}$ behave like equivalent programs when given non-equivalent but closely related inputs? More specifically, given a point x in the domain of $\tilde{\mathbf{P}}$, is there an input $\tilde{x}$ close to x such that $\mathbf{P}(\tilde{x}) = \tilde{\mathrm{P}}(x)$? This question is answered through a *backward error analysis*, and the distance between the input x and the input $\tilde{x}$ is known as the *backward error*. In Figure 1.1, the backward error is represented by the quantity δx. While an accurate program is characterized by a small forward error, a backward error analysis gives more insight into the quality of an approximating program: a large backward error suggests that the programs $\tilde{\mathbf{P}}$ and $\mathbf{P}$ are not closely related. Moreover, a small backward error implies a small forward error so long as $\mathbf{P}$ is robust to small perturbations in its inputs.

However, there are several challenges presented by backward error analysis. First, if $\mathbf{P}$ is not surjective, then a solution produced by $\tilde{\mathbf{P}}(x)$ might be outside of the codomain of $\mathbf{P}$. In such cases, the value $\tilde{x} \in \mathrm{X}$ in Figure 1.1 does not exist. Another complication with backward error analysis is that backward error guarantees are generally not compositional. This means that a bound on the backward error of a program cannot be reliably derived from bounds on the backward error of its individual components. As a result, statically analyzing backward error remains an open challenge.

Program Distance

The concept of program distance has recently emerged as a formalism that offers a more refined analysis of the behavior of programs compared to the standard notion of program equivalence (Gavazzo, 2019; Dal Lago and Gavazzo, 2022a; Azevedo de Amorim et al., 2017; Gavazzo, 2018).

[1] Recent work has begun to address this limitation (Titolo et al., 2024; Abbasi and Darulova, 2023).

This is particularly true for programs with effectful properties—programs whose behavior have an effect on the environment in which they are evaluated—and coeffectful properties—programs whose behavior depends on the environment in which they are evaluated. While various formalisms for reasoning about program distance have been used to study probabilistic languages (de Amorim et al., 2021; Crubille and Dal Lago, 2015) and languages for differential privacy (Wunder et al., 2023; Reed and Pierce, 2010; Chatzikokolakis et al., 2014), the characterization above of forward and backward error suggests that these ideas can also be applied to study languages for forward and backward error analysis. Intuitively, forward rounding error can be modeled using both effects and coeffects: the total forward rounding error resulting from the evaluation of a finite-precision program depends on the rounding error produced locally by the function, as well as the errors propagated from the inputs. On the other hand, backward error can be described as a coeffectful property: when backward error bounds exist, they characterize the relationship between a finite-precision computation and its ideal counterpart in terms of perturbations to the input space.

1.2 Type-Based Approaches to Rounding Error Analysis

One approach to developing correct numerical software is to write programs and the specifications of their numerical behavior together, using a typed programming language. In carefully designed languages that soundly represent rounding error analysis through typing, types can capture properties like accuracy requirements and acceptable error bounds. By representing these properties through typing, accuracy requirements can be enforced via type checking, ensuring that programs adhere to their intended numerical behavior. For many applications, this approach potentially eliminates the need for manual reasoning about numerical accuracy or reliance on external error analysis tools, as the language itself provides guaranteed rounding error bounds.

To illustrate this approach in practice, consider the following example. Elementary functions like sin, exp, and log are often implemented in math libraries using polynomial approximations,

and the accuracy of an implementation is highly dependent on the method used for polynomial evaluation. For instance, the following example is an IEEE binary64 polynomial approximation to the function 2^x for $0 < x \leq 1/512$:

$$P(x) = 1 + \frac{6243314768165347}{2^{53}} \cdot x + \frac{8655072058047061}{2^{55}} \cdot x^2 + \frac{3999491778607567}{2^{56}} \cdot x^3 + \frac{5548134412280811}{2^{59}} \cdot x^4$$

According to the analysis given by Muller (2016), the relative forward error due to rounding of an implementation exp2 : f64 → f64 of $P(x)$ should ideally be several orders of magnitude smaller than 1.11×10^{-16}, which is the tightest bound that can be derived if a naive implementation of polynomial evaluation is used. In a language with an expressive type system that explicitly tracks rounding error bounds at the level of types, type inference serves as a mechanism for automatically computing sound rounding error bounds for every program. In this setting, as a prelude to the syntax of the languages proposed in this dissertation, the type of the function exp2 that returns a value of type f64 and produces at most 1.11×10^{-16} relative forward error due to rounding is:

```
exp2 : f64 → M[1.11e-16]f64.
```

Functions that require highly accurate implementations of the polynomial approximation $P(x)$ can specify this fact in the type declarations of their arguments. For instance, consider the function foo, defined below:

```
fun foo (accurate_exp2 : f64 → M[2.17e-19]f64, y : f64) {
  let x = accurate_exp2(y); // call to an accurate exp2
  return x + 3.0;
}
```

This function is a *higher-order function* that takes as an argument a function with the following type:

```
accurate_exp2 : f64 → M[2.17e-19]f64
```

This type specifies a function that returns a value of type f64 with at most 2.17×10^{-19} relative error due to rounding. If a function that produces more than 2.17×10^{-19} relative rounding error is provided as an argument to foo, the program will fail to type check. Specifically, the type checker for the language enforces foo's accuracy requirement, rejecting cases where the argument does not meet this constraint. For example, passing the function exp2 : f64 → M[1.11e-16]f64 which allows a relative error of 1.11×10^{-16} as an argument to foo will fail to type check:

```
let z = foo(exp2); // ERROR mismatched types: expected 1.11e-16 ≤ 2.17e-19
```

Thus, by embedding types that track rounding error directly into the type system of a language, we can specify and enforce rigorous accuracy requirements without requiring external tools or performing a manual error analysis. The ability to track and verify accuracy constraints at the level of types ensures that programs adhere to their intended numerical behavior and provides strong guarantees about the reliability of numerical computations.

Moreover, higher-order functions like `foo` highlight how these type systems facilitate modular reasoning about accuracy. Functions can explicitly require arguments to meet strict error constraints, and the type checker for the language enforces these requirements, ensuring that only implementations with sufficiently small rounding error can be used. This guarantees correctness by construction, reducing the risk of subtle numerical errors propagating through larger programs.

To ensure that a program's type accurately represents its rounding error bounds, the type system must satisfy a soundness property, which we call *error soundness*. This property establishes that the rounding error bound expressed in a program's type, such as the value `2.17e-19` in the type `M[2.17e-19]f64`, is not merely an annotation but is *meaningful*. Specifically, if the type system assigns this type to a program, then the program is guaranteed adhere to this error bound during execution, thereby ensuring the correctness of its numerical behavior. Moreover, *error soundness* guarantees that these bounds are sound overapproximations of the true rounding error, providing a reliable foundation for reasoning about numerical accuracy. To establish error soundness for our languages, we construct denotational semantic models that assign precise meaning to types, capturing the notion of program distance as characterized by backward and forward error. We

1.3 Goals and Contributions

The goal of this dissertation is to demonstrate linguistic features and type systems that unify the tasks of reasoning about numerical accuracy and writing numerical programs. It presents novel programming languages designed for both forward error analysis and backward error analysis and details the development of formal categorical denotational semantics for these languages, along with implementations of type inference and type checking algorithms. The methods and languages presented in this dissertation serve as a blueprint for future languages that make reasoning about numerical accuracy more accessible and convenient for programmers.

Our contributions can be categorized into three main areas: denotational semantics, language design, and implementation. The remainder of this chapter is devoted to summarizing each of these contributions.

Contribution 1: Denotational Semantics

A desirable property for tools that automatically bound floating-point rounding errors is *soundness*: the bounds produced by the tool should overapproximate the true error. The first contribution of this dissertation is the introduction of categorical structures that soundly model forward and backward error analysis. These semantic structures give insight into the foundational concepts underlying programming languages that can soundly express rounding error bounds through typing.

For forward error analysis, Section 3.4 defines the *neighborhood monad*, a novel graded monad on the category of metric spaces. This monad uses grade information to model bounds on the distance between two closely related computations. Forward relative error bounds are obtained by instantiating the neighborhood monad with distance functions proposed in the numerical analysis literature that approximate relative error.

For backward error analysis, Section 4.3 introduces a semantic structure called *backward error lenses*, which describes computations suitable for backward error analysis. A backward error lens consists of a triple of related transformations that collectively satisfy a backward error guarantee. This structure supports the representation of standard numerical primitives and their associated backward error bounds. The *category of backward error lenses* (**Bel**) serves as a semantic framework for reasoning about backward error analysis.

Contribution 2: Language Design

Categorical frameworks for forward and backward error analysis provide an abstract domain for specifying programming language constructs—such as function definitions, pairs, and conditionals—that preserve the semantics of an error analysis. To this end, the second contribution of this dissertation is the design of two languages: **NumFuzz** for forward error analysis, and **Bean** for backward error analysis, each equipped with a type system that guarantees bounded error.

The type system for **NumFuzz** is presented in Section 3.2 and is based on *Fuzz* (Reed and Pierce, 2010), a linear call-by-value λ-calculus designed for differential privacy. *Fuzz* uses a linear type system and a graded comonadic type to statically perform a sensitivity analysis. **NumFuzz** extends *Fuzz*'s type system with a graded monadic type for tracking local rounding errors. Additionally, **NumFuzz** introduces typing rules combining sensitivity and local rounding error, enabling a compositional approach to analyzing the rounding error of larger programs. **NumFuzz**'s type system guarantees that programs have bounded forward error, a property we refer to as *forward error soundness*. We establish this guarantee in Section 3.5 by modeling the graded monadic type using the neighborhood monad described above, and by relating the categorical denotational semantics to an ideal and floating-point operational semantics.

Bean is a first-order *bidirectional* programming language based on numerical primitives. In bidirectional programming languages (Bohannon et al., 2008; Foster, 2009; Foster et al., 2012),

expressions usually denote a related pair of transformations: a *forward* transformation mapping inputs to outputs, as well as a *backward* transformation mapping an updated output together with an original input to a corresponding updated version of the input (Bohannon et al., 2008). To capture backward error analysis in **Bean**, each expression instead represents a *triple* of transformations: two forward transformations—one for the *ideal* program and one for the *approximate* program—and a backward transformation that relates these forward transformations under the constraints of a *backward error lens*. Section 4.2 introduces **Bean**'s type system, which uses strict linearity to ensure that when subexpressions are composed, the resulting larger expression also satisfies the constraints of a backward error lens. A graded coeffect system then computes static bounds on the backward error, based on bounds for the numerical primitives of the language. This type and effect system guarantees that **Bean** programs have bounded backward error, a property we refer to as *backward error soundness*. The proof of backward error soundness for **Bean** is given in Section 4.4.

Contribution 3: Implementation

Many analysis tools have been developed to automatically bound the forward rounding error of floating-point expressions; however, none of these tools have employed a type-based approach, nor do they address backward error analysis. The third contribution of this dissertation, described in Section 3.7 and Section 4.6, is the implementation of type checkers and grade inference algorithms for both **NumFuzz** and **Bean**. Both implementations build on the sensitivity inference algorithm introduced by Gaboardi et al. (2013).

In this type-based approach, error bounds can be inferred and checked for simple numerical programs: forward error bound in the implementation of **NumFuzz**, and backward error bounds in the implementation of **Bean**. This approach is attractive because it automatically provides a formal proof that a given program has a certain error bound.

We evaluate our implementation of **NumFuzz** using a variety of benchmarks from the literature

and demonstrate that it infers error bounds that are competitive with those produced by other tools. We also show that our implementation is capable of handling the largest benchmarks in the literature, such as a 128 x 128 matrix multiplication involving over 4 million floating-point operations. Although our prototype implementation currently only supports a limited set of primitive floating-point operations (addition, multiplication, division, square root), our empirical evaluation shows that the time complexity of the inference algorithm is linear in the size of the **NumFuzz** program. We therefore expect that any further extensions to **NumFuzz** to support additional primitive operations will not affect the complexity of the type checker and inference algorithm. This distinguishes our type-based approach from other tools, where the complexity depends on the specific floating-point operation being performed.

For **Bean**, we translate several large benchmarks into the language, demonstrating that its implementation effectively infers useful error bounds and scales to handle large numerical programs. Since **Bean** is the first tool to statically derive *sound* backward error bounds, a direct comparison with existing tools is limited. We therefore evaluate our implementation of **Bean** using three complementary methods. First, we compare the backward error bounds inferred by **Bean** to those from a dynamic analysis tool by Fu et al. (2015), which provides the only automated quantitative backward error bounds available, as well as to theoretical worst-case bounds from the literature. Additionally, we use forward error as a proxy by deriving forward error bounds from **Bean**'s backward error bounds, leveraging known values of the condition number. These forward error bounds are then compared those produced by other tools, including **NumFuzz**.

The artifact for our implementation of **NumFuzz** is available at https://zenodo.org/records/10967298, and the working branch is available at https://github.com/ak-2485/NumFuzz. The working branch of **Bean** is available at https://github.com/ak-2485/NumFuzz/tree/bean.

1.4 Dissertation Outline

This dissertation connects several topics that have not been previously linked, including floating-point arithmetic, type systems, category theory, and bidirectional programming languages. To support this integration, we present essential mathematical notations and preliminaries in Chapter 2.

In Chapter 3, we describe **NumFuzz**, covering its type system (Section 3.2), denotational semantics (Section 3.4), operational semantics (Section 3.3), and soundness guarantee (Section 3.5). We provide examples of how to soundly instantiate the language parameters in Section 3.6, and describe a prototype implementation along with its evaluation in Section 3.7. Related work on static analysis techniques for sensitivity and forward rounding error analysis is discussed in Section 3.8, and future directions for **NumFuzz** are summarized in Section 3.9. Omitted lemmas and proofs from this chapter are provided in Appendix A.

In Chapter 4, we describe **Bean**, introducing its type system (Section 4.2) and denotational semantics (Section 4.3). The soundness guarantee, along with the necessary operational constructions, is presented in Section 4.4. Examples in Section 4.5 demonstrate how **Bean** can be used to establish sound backward error bounds for various numerical problems through typing. The implementation and evaluation are described in Section 4.6. Related work on static analysis techniques for backward error analysis is covered in Section 4.7, and Section 4.8 concludes the chapter and describes future work. Omitted lemmas and proofs from this chapter are provided in Appendix B.

CHAPTER 2

BACKGROUND

The languages described in this dissertation link several topics: floating-point arithmetic, type systems, category theory, and bidirectional programming languages. While the role of type systems as tools for statically reasoning about the behavior of programs is well-established in many areas of computing, connecting this core idea from the theory of programming languages to numerical analysis is a novel contribution of the work described in this dissertation. To support the integration of these concepts, this part of the dissertation provides definitions, notation, and background results on floating-point arithmetic, as well as on type systems and their categorical semantics.

2.1 Floating-Point Arithmetic

A *finite* floating-point number x in a floating-point format $\mathbb{F} \subseteq \mathbb{R}$ has the form

$$x = (-1)^s \cdot m \cdot \beta^{e-p+1} \tag{2.1}$$

where $\beta \in \{b \in \mathbb{N} \mid b \geq 2\}$ is the *base*, $p \in \{prec \in \mathbb{N} \mid prec \geq 2\}$ is the *precision*, $m \in \mathbb{N} \cap [0, \beta^p)$ is the *significand*, $e \in \mathbb{Z} \cap [\mathrm{emin}, \mathrm{emax}]$ is the *exponent*, and $s \in \{0, 1\}$ is the *sign* of x. A complete definition of a floating-point format also specifies binary encodings, and describes how to handle *non-finite* special values, such as infinities and NaNs. For finite floating-point numbers, the parameter values for formats defined in the IEEE 754 standard for floating-point arithmetic (IEEE Computer Society, 2019) are given in Table 2.1; in all cases, emin = 1-emax. Although single (binary32)

Table 2.1: Floating-point format parameters according to the revised IEEE 754-2008 standard.

Parameter	**binary16**	**binary32**	**binary64**	**binary128**
p	11	24	53	113
emax	31	127	1023	16383

and double (binary64) precision floating-point formats have historically been the most widely used, modern architectures are increasingly supporting half precision (binary16).

Finite floating-point numbers are commonly categorized into two types: normal and subnormal. A number of the form given in Equation (2.1) is considered normal if its significand m and exponent e satisfy the bounds $e \geq \mathrm{emin}$ and $\beta^{p-1} \leq m < \beta^{p}$; otherwise, it is said to be subnormal. If the magnitude of the result of a floating-point operation is subnormal, then the operation is said to have *underflowed*. On the other hand, if the magnitude of the result exceeds the largest representable normal number in the format, the result is said to have *overflowed*. While overflows result in a total loss of precision, underflows in floating-point formats that support subnormal numbers result in a gradual, rather than total, loss of precision.

Even at higher precisions, most real numbers cannot be represented exactly by a finite floating-point number. Additionally, the result of most elementary operations on floating-point numbers cannot be represented exactly and must be *rounded* to the nearest representable value, following a specific rounding strategy. This process can introduce some *rounding error* in the result.

2.1.1 Rounding Functions

According to the IEEE 754 standard, rounding is viewed as an operation that maps real numbers to elements of the extended real numbers $\mathbb{R} \cup \{-\infty, +\infty\}$, which allows for representing the results of operations that overflow. Four rounding modes are specified in the standard: round towards $+\infty$, round towards $-\infty$, round towards 0, and round towards nearest (with defined tie-breaking schemes). The properties of these modes are given in Table 2.2.

In practice, we are often concerned with analyzing the rounding error without considering overflow. Thus, it is usually sufficient to define rounding into a given format as a function into the corresponding format with unbounded exponents (Harrison, 1997b; Boldo and Melquiond, 2017;

Table 2.2: Rounding Operations (Modes).

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

Boldo et al., 2023). Indeed, most textbook presentations of rounding error analysis also assume that the range of the floating-point format being rounded into is extended to arbitrarily small values; i.e., it is also assumed that there is no underflow. To formalize these assumptions, given a real number x and a floating-point format $\mathbb{F}$ with an unbounded exponent range, we define a *rounding function* $\rho : \mathbb{R} \to \mathbb{F}$ as a function that takes x and returns a nearby floating-point number $\rho(x) \in \mathbb{F}$.

In general, well-defined rounding functions are monotone and act as the identity function on the set of floating-point numbers (Muller et al., 2018):

- $\forall x, y \in \mathbb{R}.\ x \leq y \to \rho(x) \leq \rho(y)$
- $\forall x \in \mathbb{F}.\ \rho(x) = x$

The following result establishes a bound on the magnitude of the rounding error produced by rounding a real number, where u indicates the *unit roundoff* for the given rounding function and format.

Theorem 1. Given a real number $x \in \mathbb{R}$, and assuming no underflow or overflow occurs, the following equality holds (Higham, 2002, Theorem 2.2):

$$\rho(x) = x(1 + \delta), \quad \text{where } |\delta| \leq u.$$

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

2.1.2 Models of Floating-Point Arithmetic

The purpose of a rounding error analysis is to derive an a priori bound on the floating-point rounding errors that are produced during the execution of a program. Performing a rounding error analysis requires that we first establish a model describing the accuracy of the basic arithmetic operations. Following the IEEE 754 standard, each basic arithmetic operation ($+, -, *, \div, \sqrt{}$) should behave as if it first computed a correct, infinitely precise result, and then rounded this result using one of the functions in Table 2.2. Given the result in Theorem 1, this assumption on the computational behavior of each basic arithmetic operation leads to the following commonly used *standard rounding error model*:

Definition 1 (The Standard Rounding Error Model)**.** For any floating-point numbers $x, y \in \mathbb{F}$, and for some rounding function $\rho : \mathbb{R} \to \mathbb{F}$, the standard rounding error model for basic floating-point operations is given as follows (Higham, 2002):

$$\rho(x \; op \; y) = (x \; op \; y)(1 + \delta), \quad |\delta| \leq u, \quad op \in \{+, -, *, \div\}. \tag{2.2}$$

An alternative to the standard rounding error model was proposed by Olver (Olver, 1978), and was later applied to an early error analysis of Gaussian elimination (Olver and Wilkinson, 1982; Olver, 1982) and also of matrix computations (Pryce, 1984, 1985):

Definition 2 (Alternative Rounding Error Model)**.** For any floating-point numbers $x, y \in \mathbb{F}$, and for some rounding function $\rho : \mathbb{R} \to \mathbb{F}$, an alternative rounding error model for the basic floating-point operations is given as follows:

$$\rho(x \; op \; y) = (x \; op \; y)e^{\delta}, \quad |\delta| \leq \frac{u}{1-u}, \quad op \in \{+, -, *, \div\}. \tag{2.3}$$

If the rounding function is round towards $+\infty$, then we can guarantee a slightly tighter bound than the one given in Definition 2:

Lemma 1. For any floating-point numbers $x, y \in \mathbb{F}$ we have:

$$\rho_{\mathrm{RU}}(x \; op \; y) = (x \; op \; y)e^{\delta}, \quad |\delta| \leq u, \quad op \in \{+, -, *, \div\}. \tag{2.4}$$

2.1.3 Measures of Accuracy

While the most common measures of accuracy are *relative error* and *absolute error*, the alternative model given in Definition 2 naturally accommodates the notion of *relative precision*. We define each of these measures below.

Relative and Absolute Error Absolute error (er_{abs}) and relative error (er_{rel}) are commonly used to measure the error of approximating a value x by a value $\tilde{x}$.

$$er_{abs}(x, \tilde{x}) = |\tilde{x} - x| \tag{2.5}$$

$$er_{rel}(x, \tilde{x}) = |(\tilde{x} - x)/x| \quad \text{if } x \neq 0 \tag{2.6}$$

According to the standard model for floating-point arithmetic (Definition 1), the relative error of each of the basic floating-point operations is bounded by the unit roundoff. The relative and absolute error do not apply uniformly to all values: the absolute error is well-behaved for small values, while the relative error is well-behaved for large values.

Relative Precision We use the following definition of relative precision, adapted from Olver (1978):

Definition 3 (Relative Precision (RP))**.** The *relative precision* (RP) of $\tilde{x}$ as an approximation to x is given by

$$\mathrm{RP}(\tilde{x}, x) = \begin{cases} |\ln(\tilde{x}/x)| & \text{if } \mathrm{sgn}(\tilde{x}) = \mathrm{sgn}(x) \text{ and } \tilde{x}, x \neq 0 \\ 0 & \text{if } \tilde{x} = x = 0 \\ \infty & \text{otherwise.} \end{cases} \tag{2.7}$$

According to the alternative model for floating-point arithmetic (Definition 2), the relative precision of each of the basic floating-point operations is bounded by slightly more than the unit roundoff. If the rounding function is fixed as round towards $+\infty$, then the bound on the relative precision can be shown to be somewhat tighter using the standard model (Definition 1). In that case, the error variable δ in Definition 1 is non-negative, so that $|\ln(1+\delta)| \leq \delta$, and the relative precision of each of the basic floating-point operations is bounded by the unit roundoff:

$$\forall x, y \in \mathbb{R}.\ \mathrm{RP}(\rho(x\ op\ y), (x\ op\ y)) \leq |\ln(1+\delta)|, \quad |\delta| \leq u \tag{2.8}$$

The close relationship between relative precision and relative error can be seen by rewriting Definition 3 and Equation (2.6) as follows:

$$er_{rel}(x, \tilde{x}) = |\gamma|; \quad \tilde{x} = (1+\gamma)x \tag{2.9}$$

$$\mathrm{RP}(x, \tilde{x}) = |\gamma|; \quad \tilde{x} = e^{\gamma}x \tag{2.10}$$

If we consider the Taylor expansion of the exponential, then from Equation (2.9) and Equation (2.10) we can see that the relative precision is a close approximation to the relative error so long as $\gamma \ll 1$. Moreover, for some $\tilde{x}$ and x with $\mathrm{RP}(x, \tilde{x}) = \alpha$ and $\alpha < 1$, the following inequality holds:

$$er_{rel} = |e^{\alpha} - 1| \leq \alpha/(1-\alpha) \tag{2.11}$$

A main advantage of relative precision in comparison to relative error is that Definition 3 defines an extended pseudometric for all real numbers. Specifically, unlike relative error, relative precision satisfies the following properties:

1. Reflexivity: $\forall x \in \mathbb{R}.\ \mathrm{RP}(x, x) = 0$
2. Symmetry: $\forall x, y \in \mathbb{R}.\ \mathrm{RP}(x, y) = \mathrm{RP}(y, x)$
3. Triangle Inequlity: $\forall x, y, z \in \mathbb{R}.\ \mathrm{RP}(x, z) \leq \mathrm{RP}(x, y) + \mathrm{RP}(y, z)$

2.2 Type Systems

A type system is a principled system for assigning types to programs. This assignment is carried out using a set of rules, which inductively define a set of valid typing judgments. The typing judgment, which asserts that an expression e can be given type τ relative to a typing environment Γ for the free variables of e, has the form $\Gamma \vdash e : \tau$; the typing environment Γ can be viewed as a partial map from variables to types. The premise judgments in each rule are written above a horizontal inference line, and a single conclusion judgment is written below the line, with the name of the rule appearing to the left of the line. Given a typing environment Γ and expression e, if there is some τ such that $\Gamma \vdash e : \tau$, we say that e is *well-typed under context* Γ; if Γ is the empty context ($\emptyset$), we say e is *well-typed*, and write the judgment as $\vdash e : \tau$.

Type checking an expression e amounts to showing that the term is well-typed by constructing a derivation of the judgment $\vdash e : \tau$ for some type τ. For example, in the simply-typed lambda calculus extended with a let expressions, the program $\mathsf{let}\ x = 3\ \mathsf{in}\ x + 1$ is well-typed, with type int. The relevant syntax for types and terms includes integer literals n:

$$\begin{array}{ll} \text{Types} & \tau ::= \ \text{int} \mid \sigma \rightarrow \tau \\ \text{Expressions} & e ::= \ x \mid n \in \mathbb{N} \mid \lambda x : \tau.\ e \mid e\ f \mid e + f \mid \mathsf{let}\ x = e\ \mathsf{in}\ f \end{array}$$

To type check this program, only a few typing rules are needed:

$$\text{(Const)}\ \frac{n \in \mathbb{N}}{\Gamma \vdash n : \textbf{int}} \qquad \text{(Var)}\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

$$\text{(Add)}\ \frac{\Gamma \vdash e : \text{int} \qquad \Gamma \vdash f : \text{int}}{\Gamma \vdash e + f : \text{int}} \qquad \text{(Let)}\ \frac{\Gamma \vdash e : \sigma \qquad \Gamma, x : \sigma \vdash f : \tau}{\Gamma \vdash \mathsf{let}\ x = e\ \mathsf{in}\ f : \tau}$$

The type derivation for the program proceeds as follows:

$$
\dfrac{\text{(Const)}\ \dfrac{}{\vdash 3 : \mathsf{int}} \qquad \text{(Add)}\ \dfrac{\text{(Var)}\ \dfrac{}{x : \sigma \vdash x : \mathsf{int}} \qquad \text{(Const)}\ \dfrac{}{x : \sigma \vdash 1 : \mathsf{int}}}{x : \sigma \vdash x + 1 : \mathsf{int}}}{\vdash \mathsf{let}\ x = 3\ \mathsf{in}\ x + 1 : \mathsf{int}}\ \text{(Let)}
$$

Typically, types describe only the basic structure of the data that programs operate on, but it is possible for types to express more detailed information about programs. For instance, *graded monadic types* refine type information to describe and track *effects*—any observable behaviors that might arise during evaluation beyond the production of values. Examples of computations with effects include partial functions, raising errors or exceptions, performing input/output (IO), and, as a novel contribution of our work, *rounding*. On the other hand, *graded comonadic types* represent a different approach to refining types by focusing on describing and tracking *coeffects*—how programs depend on their environment, rather than the effects they have on it. Coeffects have been used to describe a wide range of program behaviors, including resource requirements, program sensitivity (Reed and Pierce, 2010), and, as part of this dissertation, *backward error*.

2.2.1 Effects and Graded Monadic Types

Graded monadic types unify two approaches to describing and tracking computational effects: effect systems and monads. Introduced by Gifford and Lucassen (Gifford and Lucassen, 1986; Lucassen, 1987; Lucassen and Gifford, 1988), and later developed by Talpin and Jouvelot (Talpin and Jouvelot, 1992; Talpin, 1993; Talpin and Jouvelot, 1994) and others (Nielson and Nielson, 1999), effect systems are a class of static analysis techniques that extend the types and typing rules of an underlying type system with annotations describing the effects that the primitive operations in the language might produce. For instance, in effect system, function types have the form $\sigma \xrightarrow{s} \tau$ where the effect annotation s is traditionally taken from a join semilattice $(\mathcal{S}, \sqcup, \bot, \sqsubseteq)$ (Mycroft et al., 2015). This type describes a computation that returns a value of type τ and may have an effect described by s when applied to a value of type σ.

The typing rules in effect systems track individual effects, and provide a fine-grained description of how effects accumulate and interact. An example typing rule from a simple effect system is the (Let-E) rule for let expressions below, which says that the overall effect is the (maximum) of the effect of the binding expression e and the effect of the body expression f:

$$\text{(Let-E)}\ \frac{\Gamma \vdash e : \sigma, r \qquad \Gamma, x : \sigma \vdash f : \tau, s}{\Gamma \vdash \mathsf{let}\ x = e\ \mathsf{in}\ f : \tau, r \sqcup s}$$

A key advantage to this approach is that effect annotations can often be automatically inferred using natural extensions of existing type inference algorithms (Nielson et al., 1999).

In a separate line of research, Moggi (1989, 1991) demonstrated that effectful computations can be modeled using *monads* from category theory. Syntactically, monads are incorporated into the language via a monadic type constructor, written as M, where a type $\mathrm{M}\tau$ represents computations that yield a value of type τ and may produce effects. At the syntactic level, this monadic type provides a coarser view of effects than the annotations in effect systems. For example, consider the following (Let-M) monadic typing rule for let expressions, which provides a syntax-directed way to sequence effectful computations:

$$\text{(Let-M)}\ \frac{\Gamma \vdash e : \mathrm{M}\sigma \qquad \Gamma, x : \sigma \vdash f : \mathrm{M}\tau}{\Gamma \vdash \mathsf{let}\ x = e\ \mathsf{in}\ f : \mathrm{M}\tau}$$

Clearly, the (Let-M) lacks the fine-grained detail provided by the (Let-E) rule for effect systems above: the (Let-M) rule indicates that the computation resulting from a monadic let-binding may have *some* effect, but it doesn't specify the exact nature or extent of the effect. In contrast, the (Let-E) rule precisely characterizes the maximum effect the result may have; i.e., $r \sqcup s$. Consequently, the utility of inference algorithms in this setting is less apparent when compared to effect systems.

Wadler and Thiemann (2003) were the first to propose that the monad structure introduced by Moggi could be generalized into a family of monads, and used an *effect-annotated monadic*

type to syntactically integrate monads and effect systems. Later, Katsumata (2014) introduced a denotational semantic framework for the effect-annotated monadic type based on *graded monads*, which we will soon introduce in Section 2.2.3. Consequently, *graded monadic types* embed graded monads into the syntax of a type system. The graded monadic type constructor M_r refines the monadic type constructor M into a family of type constructors indexed by the grade r, where r is an element of a *preordered monoid*:

Definition 4. A preordered monoid is a tuple $\mathcal{E} = (\mathrm{E}, \leq, 1, \odot)$ such that $(\mathrm{E}, \leq)$ is a preordered set and the binary operator $\odot$ is monotone with respect to $\leq$ in each argument.

One advantage of using elements of a preordered monoid rather than a join semilattice is that, although using the join operator ($\sqcup$) to compute the effect of let expressions is sound, it is not always precise (Katsumata, 2014). The corresponding (Let-G) typing rule for sequencing computations of graded monadic type is an intuitive combination of the (Let-E) rule and (Let-M) rule:

$$\text{(Let-G)}\ \frac{\Gamma \vdash e : \mathrm{M}_r\sigma \qquad \Gamma, x : \sigma \vdash f : \mathrm{M}_s\tau}{\Gamma \vdash \mathsf{let}\ x = e\ \mathsf{in}\ f : \mathrm{M}_{r \odot s}\tau}$$

2.2.2 Coeffects and Graded Comonadic Types

While graded monadic types provide an expressive mechanism for tracking how programs affect their environment, they cannot track how programs *depend* on their environment. Type systems that precisely characterize these *contextual program properties*, known as coeffects, have been developed by Brunel et al. (2014), Ghica and Smith (2014), Petricek et al. (2014, 2013), and Petricek (2016). The denotational semantics of these systems, which use *comonads*—the categorical dual of monads—is well-established. Here, we follow the presentation of Gaboardi et al. (2016) and Brunel et al. (2014) where comonads are embedded into the syntax using a *graded comonadic type* constructor, $!_s$.

The type constructor $!_s$ generalizes the exponential type constructor ! from linear type systems (Wadler, 1990). Linear type systems are derived from linear logic (Katsumata, 2014; Girard, 1987), where each assumption must be used exactly once. If an assumption A can be discarded or duplicated, it is marked as !A. Similarly, in linear type systems, the type constructor ! differentiates between linear, single-use data (denoted by a plain type τ) and non-linear, reusable data (denoted by the exponential type $!\tau$). For instance, if the body of a function can access a free variable x of type τ an arbitrary number of times, then x would be assigned the type $!\tau$.

Refining the type constructor $!\tau$ into a family of type constructors $!_s$ allows for tracking more fine-grained program properties, such as *bounding* or *limiting* the number of times a variable can be accessed. The first attempt to refine the exponential type in this way was introduced in bounded linear logic (Girard et al., 1992), where the annotation s on the constructor $!_s$ is a polynomial bounding the computational complexity of a program. More generally, if an expression can access a free variable x of type τ at most s times, then x would be assigned the type $!_s\tau$, where the grade s is an element of a *preordered semiring*:

Definition 5. A *preordered semiring* is a tuple $\mathcal{R} = (\mathrm{R}, \leq, 0, +, 1, \cdot)$ where $(\mathrm{R}, \leq)$ is a preordered set, $(\mathrm{R}, 0, +, 1, \cdot)$ is a semiring and both $+$ and $\cdot$ are monotone with respect to $\leq$ in both arguments.

In this framework, accurately tracking and limiting the usage of variables requires a redefinition of typing environments as partial maps from variables to both types *and* grades. This allows environments to not only assign types to variables but to also track the specific number of times each variable can be accessed, ensuring that the variable usage requirements of programs match the grade associated with each variable. For instance, if $\Gamma(x) = (\tau, r)$, then we have the binding $x :_r \tau$ in Γ. The type judgment $x :_r \sigma \vdash e : \tau$ then asserts that the expression e requires access to the variable x of type σ a total of r times. Environments defined in this way naturally support *sum*, *scaling*, and *translation* operations:

Definition 6. If two typing environments Γ and Δ always map shared variables to the same type, i.e., if $\Gamma(x) = (\sigma, s)$ and $x \in dom(\Delta)$ imply $\Delta(x) = (\sigma, r)$ for some grade r, then their *sum* is defined as

follows:

$$(\Gamma + \Delta)(x) \triangleq \begin{cases} (\sigma, s + r) & \text{if } \Gamma(x) = (\sigma, s) \text{ and } \Delta(x) = (r, \sigma) \\ \Gamma(x) & \text{if } x \notin dom(\Delta) \\ \Delta(x) & \text{if } x \notin dom(\Gamma) \end{cases}$$

Definition 7. The *scaling* operation scales the grades in a typing environment by a given grade:

$$(s \cdot \Gamma)(x) \triangleq \begin{cases} (\sigma, s \cdot r) & \text{if } \Gamma(x) = (\sigma, r) \\ \bot & \text{if } x \notin dom(\Gamma) \end{cases}$$

Definition 8. The *translation* operation translates grades in a typing environment by a given grade:

$$(s + \Gamma)(x) \triangleq \begin{cases} (\sigma, s + r) & \text{if } \Gamma(x) = (\sigma, r) \\ \bot & \text{if } x \notin dom(\Gamma) \end{cases}$$

Given these operations on typing environments, we can consider an example of a typing rule for let expressions, which provides sequencing for coeffectful computations:

$$\text{(Let-C)}\ \frac{\Gamma \vdash e :!_s \sigma \qquad \Gamma, x :_{r \cdot s} \sigma \vdash f : \tau}{r \cdot \Gamma \vdash \mathsf{let}\ x = e\ \mathsf{in}\ f : \tau}$$

This rule composes two computations: one specifying how many times an expression is capable of being used, and one that has a usage requirement. It states that the overall let expression uses the free variables in the binding expression a scale factor of r times when the binding expression with capability s is substituted for a variable used $r \cdot s$ times in the body of the let expression.

2.2.3 Categorical Semantics

The language guarantees presented in Section 3.5 and Section 4.4 of this dissertation are obtained using a denotational-semantic framework. In denotational semantics, the meaning of a type τ is represented by an object $[\![\tau]\!]$ in a mathematical domain, defined inductively over the structure of τ. Similarly, the meaning of an expression e is interpreted as an element $[\![e]\!]$ of $[\![\tau]\!]$. More generally, the meaning of an expression depends on its context and its type. A judgment $\Gamma \vdash e : \tau$ is therefore

typically interpreted as an element of the space $[\![\Gamma]\!] \to [\![\tau]\!]$. Contexts are traditionally interpreted as the product of the underlying types: $[\![\Gamma = x_1 : \tau_1 \ldots, x_n : \tau_n]\!] \triangleq [\![\tau_1]\!] \times \cdots \times [\![\tau_n]\!]$.

Our domains of interest are categories, both common (such as the category Met, which we will see in Section 3.4) and novel (such as the category **Bel**, which we will see in Section 4.3), and so we will also refer to our denotational semantics as categorical semantics. In this setting, types are interpreted as objects in a category, and typing judgments are interpreted as morphisms between these objects.

Although this section should contain the definitions and notation necessary for the presentations of categorical semantics in Section 3.4 and Section 4.3, more detailed explanations can be found in the introductory textbooks by Awodey (2010); Leinster (2014) and Abramsky and Tzevelekos (2011).

Definition 9. A *category* C consists of:

- A collection $\mathrm{O}b_{\mathrm{C}}$ of objects.
- A collection $\mathrm{H}om_{\mathrm{C}}(\mathrm{A}, \mathrm{B})$ of morphisms for every pair of objects $\mathrm{A}, \mathrm{B} \in \mathrm{O}b_{\mathrm{C}}$.
- The *composition* morphism $g \circ f$ in $\mathrm{H}om_{\mathrm{C}}(\mathrm{A}, \mathrm{C})$ for every pair of morphisms $f \in \mathrm{H}om_{\mathrm{C}}(\mathrm{A}, \mathrm{B})$ and $g \in \mathrm{H}om_{\mathrm{C}}(\mathrm{B}, \mathrm{C})$ such that, $h \circ (g \circ f) = (h \circ g) \circ f$ for any maps $f \in \mathrm{H}om_{\mathrm{C}}(\mathrm{A}, \mathrm{B})$, $g \in \mathrm{H}om_{\mathrm{C}}(\mathrm{B}, \mathrm{C})$, and $h \in \mathrm{H}om_{\mathrm{C}}(\mathrm{C}, \mathrm{D})$.
- For each object A, an *identity* morphism $id_{\mathrm{A}} \in \mathrm{H}om_{\mathrm{C}}(\mathrm{A}, \mathrm{A})$ corresponding to object A, which acts as the identity under composition: $f \circ id = id \circ f = f$.

Languages with graded monadic and graded comonadic types embedded in the syntax interpret these types using *graded monads* and *graded comonads* on a category C.

Graded Monads

Graded monads generalize the definition of a monad, and provide a mathematical structure for interpreting graded monadic types. Variations on the generalization have been proposed by Atkey (2009), Tate (2013), Katsumata (2014), and Orchard et al. (2014, 2020). The following elementary presentation is due to Katsumata (2014):

Definition 10. Let $\mathcal{E} = (\mathrm{E}, \leq, 1, \odot)$ be a preordered monoid. A $\mathcal{E}$-graded monad on a category C consists of the following data:

- An functor $\mathrm{T}_q : \mathrm{C} \to \mathrm{C}$ for every $q \in \mathcal{E}$
- A natural transformation $\mathrm{T}(q \leq q') : \mathrm{T}_q \to \mathrm{T}'_q$ for every $q \leq q'$ satisfying

$$\mathrm{T}(q \leq q') = \mathrm{id}_{\mathrm{T}_q}$$
$$\mathrm{T}(q' \leq q'') \circ (\mathrm{T}(q \leq q')) = \mathrm{T}(q \leq q'')$$

- The natural transformation $\eta : \mathrm{Id}_{\mathrm{C}} \to \mathrm{T}_1$ called the *unit map*.
- The natural transformation $\mu_{q,q'} : \mathrm{T}_q \circ \mathrm{T}_{q'} \to \mathrm{T}_{q \odot q'}$ called the *graded multiplication map*.

These data make the following diagrams commute:

$$\begin{array}{ccc} \mathrm{T}_q \circ \mathrm{T}_{q'} & \xrightarrow{\mu_{q,q'}} & \mathrm{T}_{q \odot q'} \\ {\scriptstyle \mathrm{T}(q \leq r) \circ \mathrm{T}(q' \leq r')} \downarrow & & \downarrow {\scriptstyle \mathrm{T}(q \odot q' \leq r \odot r')} \\ \mathrm{T}_r \circ \mathrm{T}_{r'} & \xrightarrow[\mu_{r,r'}]{} & \mathrm{T}_{r \odot r'} \end{array} \qquad \begin{array}{ccc} \mathrm{T}_q & \xrightarrow{\eta \circ \mathrm{T}_q} & \mathrm{T}_1 \circ \mathrm{T}_q \\ {\scriptstyle \mathrm{T}_q \circ \eta} \downarrow & \diagdown\!\!\diagdown & \downarrow {\scriptstyle \mu_{1,q}} \\ \mathrm{T}_q \circ \mathrm{T}_1 & \xrightarrow[\mu_{q,1}]{} & \mathrm{T}_q \end{array}$$

$$\begin{array}{ccc} T_q \circ T_{q'} \circ T_{q''} & \xrightarrow{T_q \circ \mu_{q',q''}} & T_q \circ T_{q' \odot q''} \\ \downarrow{\scriptstyle \mu_{q,q'} \circ T_{q''}} & & \downarrow{\scriptstyle \mu_{q,q' \odot q''}} \\ T_{q \odot q'} \circ T_{q''} & \xrightarrow[\mu_{q \odot q', q''}]{} & T_{q \odot q' \odot q''} \end{array}$$

Functors and natural transformations are defined as follows:

Definition 11. A *functor* $F : C \to D$ between categories C and D consists of:

- An object-map $F : Ob_C \to Ob_D$, assigning an object FA of D to every object A of C.
- A function on morphisms $F : Hom_C(A, B) \to Hom_D(A, B)$, assigning a morphism $Ff : FA \to FB$ of $Hom_D(A, B)$ to every morphism $f : A \to B$ of $Hom_C(A, B)$, so that composition and identities are preserved:

$$F(g \circ f) = Fg \circ Ff, \qquad F(id_{FA}) = id_{FA}$$

Definition 12. Let $F, G : C \to D$ be functors. A *natural transformation* $\alpha : F \to G$ consists of a family of morphisms $\alpha_A \in Hom_D(F(A), G(A))$, one per object $A \in Ob_C$, that commute with the functors F and G applied to any morphism: for every $f \in Hom_C(A, B)$, we have $F(f); \alpha_B = \alpha_A; G(f)$. Diagrammatically,

$$\begin{array}{ccc} F(A) & \xrightarrow{F(f)} & F(B) \\ \downarrow{\scriptstyle \alpha_A} & & \downarrow{\scriptstyle \alpha_B} \\ G(A) & \xrightarrow[G(f)]{} & G(B) \end{array}$$

Graded Comonads

Dual to graded monads and graded monadic types, *graded comonads* generalize the definition of a comonad and serve as the mathematical structure for interpreting graded comonadic types.

Here, we provide an abridged and elementary definition for graded comonads. Detailed definitions are given by Gaboardi et al. (2016), Brunel et al. (2014), and Katsumata (2018).

Definition 13. Let $\mathcal{S} = (\mathrm{S}, \leq, 0, +, 1, \odot)$ be a preordered semiring. A $\mathcal{S}$-graded *comonad* on a *symmetric monoidal category* C with tensor unit $\mathbf{I}$ consists of the following data:

1. For every $s \in \mathrm{S}$, a functor $\mathrm{D}_s : \mathrm{C} \to \mathrm{C}$.
2. For every $s \in \mathrm{S}$, a natural transformation $m_{s,\mathbf{I}} : \mathbf{I} \to \mathrm{D}_s\mathbf{I}$, called *0-monoidality*.
3. For every $s \in \mathrm{S}$, a natural transformation $m_{s,\mathrm{A},\mathrm{B}} : \mathrm{D}_s\mathrm{A} \otimes \mathrm{D}_s\mathrm{B} \to \mathrm{D}_s(\mathrm{A} \otimes \mathrm{B})$, called *2-monoidality*.
4. A natural transformation $\varepsilon_{\mathrm{A}} : \mathrm{D}_1\mathrm{A} \to \mathrm{A}$, called *dereliction*.
5. A natural transformation $w_{\mathrm{A}} : \mathrm{D}_0\mathrm{A} \to \mathbf{I}$, called *weakening*.
6. For every $r, s \in \mathrm{S}$, a natural transformation $c_{r,s,\mathrm{A}} : \mathrm{D}_{(r+s)}\mathrm{A} \to \mathrm{D}_r\mathrm{A} \otimes \mathrm{D}_s\mathrm{A}$, called *contraction*.
7. For every $r, s \in \mathrm{S}$, a natural transformation $\delta_{r,s,\mathrm{A}} : \mathrm{D}_{r\odot s}\mathrm{A} \to \mathrm{D}_r(\mathrm{D}_s\mathrm{A})$, called *digging*.

These six natural transformations satisfy over 20 equational axioms, which we will not write here.

CHAPTER 3

A LANGUAGE FOR FORWARD ERROR ANALYSIS

This chapter presents **NumFuzz** (NUMerical FUZZ), a typed higher-order functional programming language with a linear type system that can express quantitative bounds on forward error.

3.1 Introduction

From a numerical perspective, the **NumFuzz** approach to rounding error analysis follows a well-established method: a global, compositional rounding error analysis is modeled by combining a sensitivity analysis with a local rounding error analysis. A sensitivity analysis describes how small changes in input values can affect the overall output, while a local rounding error analysis focuses on how individual arithmetic operations, when rounded, contribute to the overall error. By decomposing a large-scale analysis into these smaller, analyzable parts, a global view of rounding error can be built up from local analyses.

What distinguishes the **NumFuzz** approach is how it formalizes this process in a type system. The key novelty lies in capturing the rounding error analysis at the level of types, enabling the static computation of rounding error bounds for floating-point programs. In this approach, *graded comonadic types* (see Section 2.2.2) describe function sensitivity and *graded monadic types* describe rounding error (see Section 2.2.1). The typing rules then provide a formal, logical method for reasoning about the interaction between function sensitivity and local rounding error, and for analyzing the rounding error of increasingly complex programs. Thus, the overall rounding error of a program can be derived from the known rounding error of numerical primitives. This approach is attractive because valid derivations correspond to formal proofs that a given program satisfies the error bound assigned to it by the type system. This guarantee is rigorously established by connecting a metric denotational semantics for **NumFuzz**, which specifies the mathematical meaning of **NumFuzz** programs as non-expansive maps between metric spaces, to both an ideal

and floating-point operational semantics, which specify the computational behavior of **NumFuzz** programs.

While sensitivity type systems have been previously proposed in the differential privacy literature, the key innovation of **NumFuzz** is extending this concept to explicitly account for the propagation of rounding errors in numerical computations. By encoding the interplay between sensitivity and rounding errors directly into the type system, **NumFuzz** ensures that the rounding error of floating-point programs can be reasoned about compositionally. From the perspective of language design, this approach offers a robust framework for developing software that requires precise control over numerical accuracy.

Sensitivity Type Systems

The core of **NumFuzz**'s type system is based on *Fuzz* (Reed and Pierce, 2010), a family of languages for differential privacy that use linear type systems to track function sensitivity. The fundamental idea in linear type systems is that functions must use their arguments exactly once. This contrasts with conventional type systems, where functions are unrestricted and can use their arguments an arbitrary number of times. To distinguish conventional functions from linear ones, the types of linear functions are written as $\sigma \multimap \tau$. Unrestricted functions are encoded in linear type systems as $!\sigma \multimap \tau$, where the ! constructor is used to indicate that the argument to the function does not need to adhere to a linear usage constraint; Section 2.2.2 provides more details on the ! constructor.

Sensitivity type systems like Fuzz build on the concept of linearity to represent *c-sensitive* functions. Intuitively, a function is c-sensitive if it can amplify distances between inputs by a factor of at most c. Formally, *c-sensitivity* is defined as follows:

Definition 14 (C-Sensitivity)**.** A function $f : X \to Y$ between metric spaces is said to be *c-sensitive* (or C-Lipschitz) *iff* $d_Y(f(x), f(y)) \leq c \cdot d_X(x, y)$ for all $x, y \in X$.

In sensitivity type systems, the function type $\sigma \multimap \tau$ describes functions that are 1-sensitive; these functions are also referred *non-expansive* functions because they do not amplify distances between inputs. Interpreting the function type in this way requires that both the input and output types of the function have an associated metric. This idea is quite natural when types are viewed as metric spaces, such as the real numbers $\mathbb{R}$ with the standard metric $d_{\mathbb{R}}(x, y) = |x - y|$. In this setting, functions like $g(x) = x$ and $h(x) = sin(x)$ are 1-sensitive and can be typed with the signature $\mathbb{R} \multimap \mathbb{R}$. To express functions with varying degrees of sensitivity, the ! constructor is refined into a family of *graded comonadic type constructors*, $!_s$, where the grade s indicates a *metric scaling* and is an element of the preordered semiring (Definition 5) $\mathcal{R}$ of extended non-negative real numbers $\mathbb{R}_{\geq 0} \cup \{\infty\}$. For example, the type $!_r\sigma$ scales the metric of the type σ by a factor of r. The type $!_r\sigma \multimap \tau$ then describes the type of a function that is *r-sensitive* with respect to its argument.

To adapt these core ideas from sensitivity type systems to reason about relative rounding error in **NumFuzz**, we rely on the *relative precision* (Definition 3), a pseudometric on the real numbers proposed by Olver (1978). If we denote the relative precision of $\tilde{x} \in \mathbb{R}$ as an approximation to $x \in \mathbb{R}$ as $\mathrm{RP}(\tilde{x}, x)$ then the function $f(x) = x^2$ is 2-sensitive under the RP metric:

$$\mathrm{RP}(f(x), f(y)) = \left| \ln \left(\frac{x^2}{y^2} \right) \right| \tag{3.1}$$

$$= 2 \cdot \mathrm{RP}(x, y) \tag{3.2}$$

Spelling this out, if we have two inputs v and $v \cdot e^{\delta}$, which are at distance δ under the RP metric, then applying the function $f(x) = x^2$ results in outputs v^2 and $(v \cdot e^{\delta})^2 = v^2 \cdot e^{2 \cdot \delta}$. These outputs are at a distance of at most $2 \cdot \delta$ under the RP metric.

The function $f(x) = x^2$ can be implemented in **NumFuzz** as follows:

$$\mathsf{f} : \ !_2\mathsf{num} \multimap \mathsf{num}$$

$$\mathsf{f} \triangleq \lambda x.\ \mathsf{mul}(x, x)$$

For now, we can think of the numeric type num as the real numbers $\mathbb{R}$ equipped with the RP metric. The type $!_2\mathsf{num} \multimap \mathsf{num}$ then indicates that the function is 2-sensitive under this metric.

Rounding Error in NumFuzz

So far, we have not considered rounding error: the function $\mathsf{f} \triangleq \lambda x.\, \mathrm{mul}(x, x)$ simply squares its argument without performing any rounding. To better understand how rounding error is modeled in **NumFuzz**, and how function sensitivity interacts with rounding error, consider the function $pow2 : \mathbb{R} \rightarrow \mathbb{R}$, which squares a real number and then rounds the result using an arbitrary rounding function ρ:

$$pow2(x) = \rho(x^2)$$

Using the alternative model for floating-point arithmetic (Definition 2), the error analysis is simple:

$$pow2(x) = (x \cdot x)e^{\delta} \tag{3.3}$$

where $|\delta|$ is the relative precision and $|\delta| \leq u/(1-u)$. Our insight is that type system can be used to perform this analysis, by modeling rounding as an *error producing* effectful operation. To see how this works, the function pow2 can be defined in **NumFuzz** as follows:

$$\mathsf{pow2} : \;\; !_2\mathrm{num} \multimap M_{\varepsilon}\mathrm{num}$$

$$\mathsf{pow2} \triangleq \lambda x.\, \mathbf{rnd}(\mathrm{mul}(x, x))$$

Here, **rnd** is a primitive operation that produces values of graded monadic type M_{ε}num, where ε is a constant that models the error due to a single rounding, measured as relative precision. Therefore, $\varepsilon \leq u/(1-u)$.

More generally, the type $M_r$num describes computations that produce numeric results and might also perform an arbitrary number of roundings. The grade r expresses an upper bound on the total rounding error produced by the computation, measured as relative precision. Thus, the type for pow2 captures the desired error bound from Equation (3.3): when applied to any input, pow2 produces an output that approximates its ideal, infinitely precise counterpart to within RP distance at most $u/(1-u)$.

To formalize this guarantee, our denotational semantics in Section 3.4 interprets values of graded monadic type $M_q\tau$ as pairs of values whose components are separated by a distance no greater than q. Next, our operational semantics in Section 3.3 specify two ways to execute programs of graded monadic type: under an ideal operational semantics, where rounding operations act as the identity function, and under a floating-point operational semantics, where rounding operations round their arguments following some prescribed rounding strategy. Then, our main soundness theorem in Section 3.5 connects our denotational and operational semantics, so that the first component of the interpretation of a value of type $M_q$num is the result under the ideal operational semantics and the second component is the result under the floating-point operational semantics. This theorem guarantees that programs of monadic type $M_q$num represent computations that produce values with at most q rounding error.

Composing Error Bounds

The type of $\mathsf{pow2} : !_2\text{num} \multimap M_u\text{num}$ actually guarantees a bit more than just a bound on the roundoff: it also guarantees that the function is 2-sensitive under an *ideal* semantics. This additional piece of information is crucial for analyzing how functions that produce rounding error compose.

To see why, consider the function $h(x) = x^4$. We can implement this function using pow2:

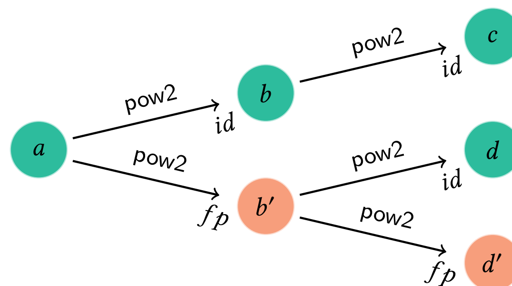
$$\mathsf{pow4} : \text{num} \multimap M_{3\varepsilon}\text{num}$$
$$\mathsf{pow4} \triangleq \lambda x.\ \mathbf{let_M}\ y = \mathsf{pow2}\ x\ \mathbf{in}\ \mathsf{pow2}\ y$$

The $\mathbf{let_M} - = - \ \mathbf{in} -$ construct sequentially composes two monadic, effectful computations. To keep the example readable, some **NumFuzz** syntax is elided. Thus, pow4 first squares its argument, rounds the result, then squares again, rounding a second time.

The bound 3ε on the total roundoff error deserves some explanation. In the typing rules for **NumFuzz** given in Section 3.2, we will see that this grade on the monadic type is computed as the

sum $2\varepsilon + \varepsilon$, where the first term 2ε is the error ε from the first rounding operation *amplified by* 2 *since this error is fed into the second call of* pow2*, a* 2*-sensitive function*, and the second term ε is the roundoff error from the second rounding operation.

Let $\mapsto_{id}$ denote the evaluation of pow4 under an ideal execution model, where **rnd** in the body of pow2 behaves like the identity function, and let $\mapsto_{fp}$ denote the evaluation of pow4 under a floating-point execution model, where **rnd** in the body of pow2 behaves like a specified rounding function. We can then visualize pow4 applied to a numeric value a as the following composition:



From left-to-right, the ideal and approximate results of pow2(a) are b and b', respectively; error soundness for **NumFuzz**, which we will see in Section 3.5 guarantees that the grade ε on the monadic return type of pow2 is an upper bound on the distance between these values. The ideal result of pow4(a) is c, while the approximate result of pow4(a) is d'. (The value d arises from mixing ideal and approximate computations, and does not fully correspond to either the ideal or approximate semantics.) The 2-sensitivity guarantee of pow2 ensures that the distance between c and d is at most twice the distance between b and b'—leading to the 2ε term in the error—while the distance between d and d' is at most ε. Applying the triangle inequality yields an overall error bound of at most $2\varepsilon + \varepsilon = 3\varepsilon$.

From a numerical perspective, the meaning of the two terms in the total error $2\varepsilon + \varepsilon$ of pow4 is clear: the first reflects how the function pow2 magnifies errors in the inputs—the sensitivity of the function, and the second reflects the *local* rounding error of pow2—how much error due to rounding is produced locally in the body of a function.

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

Figure 3.1: **NumFuzz** types, values, and terms; $s, q \in \mathbb{R}^{\geq 0} \cup \{\infty\}$.

3.2 Type System

This section describes the syntax of **NumFuzz**, which was briefly introduced in the previous section. **NumFuzz** is based on *Fuzz* (Reed and Pierce, 2010), a linear call-by-value λ-calculus, extended with explicit constructs for monadic types to model rounding. For simplicity we do not treat recursive types, and **NumFuzz** does not have general recursion.

3.2.1 Types

The syntax of **NumFuzz** types is given in Figure 3.1. The linear function type $\sigma \multimap \tau$, the graded comonadic (metric scaling) type $!_s\sigma$, and the graded monadic type $M_q\tau$ have already been introduced in Section 3.1. Additional background on these types is given in Section 2.2.2 and Section 2.2.1.

The base types in the language are a unit type and a base numeric type num. The unit type, combined with the binary sum type constructor + is used to encode Boolean types, i.e., $\mathbb{B} \triangleq \text{unit}+\text{unit}$. The sum type constructor itself represents a choice between two values, and is used to encode conditional statements. Like *Fuzz* and other linear type systems, **NumFuzz** supports two product types: a *multiplicative* product $\otimes$ and an *additive* product &.

3.2.2 Values and Terms

Aside from the monadic and comonadic constructs, most values in **NumFuzz** correspond to those found in a linear call-by-value typed λ-calculus without recursive types. These include variables, a unit value (), additive ⟨-,-⟩ products, multiplicative (-,-) products, sum constructors **inl** and **inr**, and lambda abstractions.

Languages with monadic types embedded in their syntax typically separate values and terms into two disjoint classes, and use **return** and **let** constructs to sequence monadic computations (Dal Lago and Gavazzo, 2022b; Torczon et al., 2024; Levy et al., 2003). In **NumFuzz**, although values and terms are not disjoint, all computations are explicitly sequenced using let expressions, **let** $x = v$ **in** e, and term constructors and eliminators are restricted to values. To sequence monadic and comonadic types, **NumFuzz** provides the eliminators $\textbf{let}_{\textbf{M}}$ $x = v$ **in** e and **let** $[x] = v$ **in** e, respectively. The constructs **rnd** v and **ret** v lift values of plain type to monadic type, while the comonadic construct $[v]$ indicates scaling the metric of the underlying type by a constant.

NumFuzz is parameterized by a set R of numeric constants with type num, a fixed constant ε representing the rounding error produced by the evaluation of a rounding function, and a signature Σ defining the primitive operations in the language: a set of operation symbols op $\in \mathcal{O}$, each with a type $\sigma \multimap \tau$, and a function $op : \mathrm{CV}(\sigma) \to \mathrm{CV}(\tau)$ mapping closed values of type σ to closed values of type τ. We write $\{\mathrm{op} : \sigma \multimap \tau\}$ in place of the tuple $(\sigma \multimap \tau, op : \mathrm{CV}(\sigma) \to \mathrm{CV}(\tau), \mathrm{op})$. For now, we make no assumptions on the functions op; we will see in Section 3.4 that, for soundness, we need to ensure that each function is non-expansive with respect to its type signature. In Section 3.6, we instantiate R, interpret num as a concrete set of numbers with a particular metric, and provide a concrete signature Σ.

$$\text{(Unit)}\ \frac{}{\Gamma \vdash () : \text{unit}} \qquad \text{(Const)}\ \frac{k \in R}{\Gamma \vdash k : \text{num}} \qquad \text{(Var)}\ \frac{x \notin dom(\Gamma) \qquad s \geq 1}{\Gamma, x :_s \sigma, \Delta \vdash x : \sigma}$$

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

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

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

$$(+\ \text{I}_\text{L})\ \frac{\Gamma \vdash v : \sigma}{\Gamma \vdash \mathbf{inl}\ v : \sigma + \tau} \qquad (+\ \text{I}_\text{R})\ \frac{\Gamma \vdash v : \tau}{\Gamma \vdash \mathbf{inr}\ v : \sigma + \tau}$$

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

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

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

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

$$\text{(MLet)}\ \frac{\Gamma \vdash v : M_r \sigma \qquad \Theta, x :_s \sigma \vdash f : M_q \tau}{s \cdot \Gamma + \Theta \vdash \mathbf{let_M}\ x = v\ \mathbf{in}\ f : M_{s \cdot r + q} \tau} \qquad \text{(Op)}\ \frac{\Gamma \vdash v : \sigma \qquad \{\text{op} : \sigma \multimap \text{num}\} \in \Sigma}{\Gamma \vdash \text{op}(v) : \text{num}}$$

Figure 3.2: Typing rules for **NUMFUZZ**, with $s, t, q, r \in \mathbb{R}^{\geq 0} \cup \{\infty\}$. **NUMFUZZ** is parametric in R (Const), Σ (Op), and ε (Rnd).

3.2.3 Typing Relation

The typing relation of **NumFuzz** is presented in Figure 3.2. Before stepping through each rule defining the relation, we provide some background on typing judgments and typing environments.

Typing environments in **NumFuzz** are defined as follows:

$$\Gamma, \Delta ::= \emptyset \mid \Gamma, x :_r \sigma$$

where grade annotations r are elements of the preordered semiring (Definition 5) of extended positive real numbers, i.e. $r \in ([0, \infty], \leq, 0, +, 1, \cdot)$. We extend the definition of multiplication as follows:

$$r \cdot \infty = \infty \cdot r = \begin{cases} 0 & \text{if } r = 0 \\ \infty & \text{otherwise} \end{cases} \tag{3.4}$$

A well-typed expression

$$x :_r \sigma \vdash e : \tau$$

represents a computation that is r-sensitive to perturbations in the variable x. Zero sensitivity (r=0) indicates that e is independent of x, while infinite sensitivity (r=∞) means that any perturbation in x can result in arbitrarily large changes in e.

Following Section 2.2.2, a typing environment Γ can also be viewed as a partial map from variables to types and sensitivities, where $(\sigma, r) = \Gamma(x)$ when $x :_r \sigma \in \Gamma$. The *sum* $\Gamma + \Delta$ of two typing environments (Definition 6) and the *scaling* $r \cdot \Gamma$ of a type environment by a grade (Definition 7) are defined as in Section 2.2.2.

With the structure of typing environments established, we now describe the rules in Figure 3.2. The (Const) rule allows any numeric constants with type num to be used under any environment; for now, these constants can be though of as real numbers, but their exact meaning will be fixed in example instantiations of the language given in Section 3.6. The (Var) rule allows a variable from the environment to be used so long as its sensitivity is at least 1. This rule also embeds a form of weakening into the system: the treatment of typing environments in the rule allowing variables to be

declared but not used, and also allows a variables to be declared with a higher sensitivity than is actually required. Intuitively, this captures the fact that r-sensitive functions are also s-sensitive for $r \leq s$.

The introduction and elimination rules for multiplicative products $\otimes$ and additive product & are identical to those used in *Fuzz*. To understand the difference between these two products, consider the treatment of typing environments in their respective introduction rules:

$$(\otimes \text{ I}) \frac{\Gamma \vdash v : \sigma \qquad \Theta \vdash w : \tau}{\Gamma + \Theta \vdash (v, w) : \sigma \otimes \tau} \qquad (\& \text{ I}) \frac{\Gamma \vdash v : \sigma \qquad \Gamma \vdash w : \tau}{\Gamma \vdash \langle v, w \rangle : \sigma \,\&\, \tau}$$

In the multiplicative product ($\otimes$ I), the components of the pair have free variables in summable environments, and the (variablewise) sensitivity of the resulting pair is determined by the sum of the environments. In the additive product (& I), the components of the pair share a typing environment, and the (variablewise) sensitivity of the pair is determined by this shared environment. Although the descriptors *multiplicative* and *additive* are inherited from linear logic (Wood and Atkey, 2022), they conflict with the context operations in sensitivity type systems, where multiplicative rules *add* their contexts, and additive rules *share* their contexts.

The typing rules for sequencing (Let) and case analysis (+ E) both require that the sensitivity s scaling the environment in the conclusion of the rule be strictly positive. While this restriction in the (Let) rule for let expressions is really only required for soundness in the presence of non-termination (Gavazzo, 2018) and can be omitted for a terminating calculus like **NumFuzz**, it is essential for soundness in the (+ E) rule, as described by Azevedo de Amorim et al. (2017).

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

The rules (MSub), (Ret), (Rnd), and (MLet) are the core rules for rounding error analysis in

NumFuzz. Intuitively, the monadic type $M_\varepsilon$num describes computations that produce numeric results while performing rounding, and incur at most ε in rounding error. The subsumption rule states that rounding error bounds can be loosened. The (Ret) rule states that we can lift terms of plain type to monadic type without introducing rounding error. The (Rnd) rule types the primitive rounding operation, which introduces roundoff errors. Here, ε is a fixed numeric constant describing the roundoff error incurred by a rounding operation. The precise value of this constant depends on the precision of the format and the specified rounding function; we leave ε unspecified for now. In Section 3.6, we will illustrate how to instantiate our language to different settings.

The monadic elimination rule (MLet) allows sequencing two rounded computations together. This rule formalizes the interaction between sensitivities and rounding, as illustrated by example in Section 3.1: the rounding error of the overall let expression $\textbf{let}_{\textbf{M}}\ x = v\ \textbf{in}\ f$ is upper bounded by the sum of the roundoff error of the value v scaled by the sensitivity of f to x, and the roundoff error of f.

Before introducing our operational semantics, we note that the static aspects of our system introduced so far satisfy the properties of weakening and substitution, and define the notion of a subenvironment. We write $e[v/x]$ for the capture-avoiding substitution of the value v for all free occurrences of the variable x in the expression e. We will use the notation $e[\vec{v}/dom(\Gamma)]$ to indicate the (simultaneous) substitution of all values v_i corresponding to variables x_i in the domain of the typing environment Γ into the expression e.

Definition 15 (Subenvironment)**.** The environment Δ is a *subenvironment* of Γ, written $\Delta \sqsubseteq \Gamma$, if whenever $\Gamma(x) = (\sigma, s)$ for some sensitivity s and type σ, then there exists a sensitivity s' such that $s' \geq s$ and $\Delta(x) = (\sigma, s')$.

Lemma 2 (Weakening)**.** Let $\Gamma \vdash e : \tau$ be a well-typed term. Then for any typing environment $\Delta \sqsubseteq \Gamma$, there is a derivation of $\Delta \vdash e : \tau$.

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

$$
\begin{aligned}
\mathrm{op}(v) &\mapsto op(v) \\
(\lambda x.e)\ v &\mapsto e[v/x] \\
\pi_i\langle v_1, v_2\rangle &\mapsto v_i \\
\textbf{let } [x] = [v] \textbf{ in } e &\mapsto e[v/x] \\
\textbf{let}_\mathbf{M}\ x = \textbf{ret } v \textbf{ in } e &\mapsto e[v/x] \\
\textbf{let } x \otimes y = (v, w) \textbf{ in } e &\mapsto e[v/x][w/y] \\
\textbf{case inl } v \textbf{ of } (\textbf{inl } x.e \mid \textbf{inr } y.f) &\mapsto e[v/x] \\
\textbf{case inr } v \textbf{ of } (\textbf{inl } x.e \mid \textbf{inr } y.f) &\mapsto f[v/x]
\end{aligned}
$$

$$
\textbf{let}_\mathbf{M}\ y = (\textbf{let}_\mathbf{M}\ x = \textbf{rnd } v \textbf{ in } f) \textbf{ in } g \mapsto \textbf{let}_\mathbf{M}\ x = \textbf{rnd } v \textbf{ in } (\textbf{let}_\mathbf{M}\ y = f \textbf{ in } g) \quad x \notin \mathrm{FV}(g)
$$

$$
\frac{e \mapsto e'}{\textbf{let } x = e \textbf{ in } f \mapsto \textbf{let } x = e' \textbf{ in } f}
$$

Figure 3.3: Evaluation rules for **NumFuzz**.

Lemma 3 (Substitution)**.** Let $\Gamma, \Delta \vdash e : \tau$ be a well-typed term, and let $\vec{v} \vDash \Delta$ be a well-typed substitution of closed values, i.e., we have derivations $\vdash v_i : \Delta(x_i)$ for every $x_i \in dom(\Gamma)$. Then there is a derivation of

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

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

3.3 Operational Semantics

The operational semantics described in this section specify the computational behavior of **NumFuzz** by defining an evaluation strategy for terms. To capture a forward rounding error analysis, we ultimately define two operational semantics: one that describes how terms evaluate under an ideal semantics, and one that describes how terms evaluate under a floating-point semantics. Our denotational semantics given in Section 3.4 then describe the distance between the values that terms

reduce to under these two different semantics; this connection is made precise in Corollary 1.

We start by defining a general small-step operational semantics, based on the operational semantics of *Fuzz* (Reed and Pierce, 2010), and then refine these general semantics into an ideal operational semantics and a floating-point operational semantics. The complete set of evaluation rules is given in Figure 3.3, where the judgment $e \mapsto e'$ indicates that the expression e takes a single step, resulting in the expression e'.

Although our language does not have recursive types, the $\textbf{let}_\textbf{M}$ construct makes it somewhat less obvious that the calculus is terminating: the evaluation rules for $\textbf{let}_\textbf{M}$ rearrange the term but do not reduce its size. Even so, a standard logical relations argument can be used to show that well-typed programs are terminating. If we denote the set of closed values of type τ by $\mathrm{CV}(\tau)$ and the set of closed terms of type τ by $\mathrm{CT}(\tau)$, so that $\mathrm{CV}(\tau) \subseteq \mathrm{CT}(\tau)$, and define $\mapsto^*$ as the reflexive transitive closure of the single step judgment $\mapsto$, then we can state our termination theorem as follows.

Theorem 2 (Termination)**.** If $\emptyset \vdash e : \tau$ then there exists $v \in \mathrm{CV}(\tau)$ such that $e \mapsto^* v$.

The proof of Theorem 2 follows by a standard logical relations argument. Below, we define the logical relation as the reducibility predicate $\mathcal{R}_\tau$ and state the key auxiliary lemmas used in the proof of Theorem 2. A detailed proof is given in Appendix A.1.

Definition 16. We define the reducibility predicate $\mathcal{R}_\tau$ inductively on types in Figure 3.4.

The proof of Theorem 2 relies on two key lemmas: Lemma 4 and Lemma 5. The definition of the reducibility predicate ensures that the proofs of these lemmas follow without difficulty. Lemma 4 is fairly standard and follows by induction on the typing derivation $\emptyset \vdash e : \tau$. The proof of Lemma 5 proceeds by induction on the depth of the predicate $\mathcal{VR}$.

Lemma 4. The predicate $\mathcal{R}_\tau$ is preserved by backward and forward reductions. Specifically, if $\emptyset \vdash e : \tau$ and $e \mapsto e'$ then $e \in \mathcal{R}_\tau \iff e' \in \mathcal{R}_\tau$.

$$
\begin{aligned}
\mathcal{R}_\tau &\triangleq \{e \mid e \in \mathrm{CT}(\tau) \ \wedge\ \exists v \in \mathrm{CV}(\tau).\, e \mapsto^* v \ \wedge\ v \in \mathcal{VR}_\tau\} \\
\mathcal{VR}_{\mathrm{unit}} &\triangleq \{\langle\rangle\} \\
\mathcal{VR}_{\mathrm{num}} &\triangleq \mathrm{R} \\
\mathcal{VR}_{!_s\tau} &\triangleq \{[v] \mid v \in \mathcal{R}_\tau\} \\
\mathcal{VR}_{\sigma\,\&\,\tau} &\triangleq \{\langle v, w\rangle \mid v \in \mathcal{R}_\sigma \ \wedge\ w \in \mathcal{R}_\tau\} \\
\mathcal{VR}_{\sigma\otimes\tau} &\triangleq \{(v, w) \mid v \in \mathcal{R}_\sigma \ \wedge\ w \in \mathcal{R}_\tau\} \\
\mathcal{VR}_{\sigma+\tau} &\triangleq \{v \mid \mathbf{inl}\, v \ \wedge\ v \in \mathcal{R}_\sigma \text{ or } \mathbf{inr}\, v \ \wedge\ v \in \mathcal{R}_\tau\} \\
\mathcal{VR}_{\sigma\multimap\tau} &\triangleq \{v \mid \forall w \in \mathcal{VR}_\sigma.\, vw \in \mathcal{R}_\tau\} \\
\mathcal{VR}_{M_u\tau} &\triangleq \bigcup_{n\in\mathbb{N}} \mathcal{VR}^n_{M_u\tau} \\
\mathcal{VR}^0_{M_u\tau} &\triangleq \{v \mid v \equiv \mathbf{ret}\, w \ \wedge\ w \in \mathcal{R}_\tau \text{ or } v \equiv \mathbf{rnd}\, k \ \wedge\ k \in \mathcal{R}_{\mathrm{num}}\} \\
\mathcal{VR}^{n+1}_{M_u\tau} &\triangleq \mathcal{VR}^n_{M_u\tau} \cup \Big\{ \mathsf{let}_{\mathsf{M}}\ x = v\ \mathsf{in}\ f \mid \exists\, \sigma, u_1, u_2, j.\ u \geq u_1 + u_2 \ \wedge\ n > j \\
&\qquad \wedge\ v \in \mathcal{VR}^j_{M_{u_1}\sigma} \ \wedge\ \left(\forall w \in \mathcal{VR}_\sigma, f[w/x] \in \mathcal{VR}^{n-j}_{M_{u_2}\tau}\right) \Big\}
\end{aligned}
$$

Figure 3.4: Reducibility Predicate.

Lemma 5 (Subsumption)**.** For any $m \in \mathbb{N}$, and for any monadic grades q, r such that $r \geq q$, if $e \in \mathcal{VR}^m_{M_q\tau}$, then $e \in \mathcal{VR}^m_{M_r\tau}$.

Ideal and Floating-Point Operational Semantics

Thus far, terms that include the primitive monadic rounding operation **rnd** have been treated as values, both in our presentation of the syntax of **NumFuzz** in Section 3.2 and in our definition of the evaluation rules, given in Figure 3.3. To define an ideal and floating-point operational semantics, we refine our syntax and semantics, so that the rounding operation is now an expression that steps to a number. The syntax of **NumFuzz** is updated as follows.

$$v, w ::= x \mid () \mid k \in \mathrm{R} \mid \langle v, w\rangle \mid (v, w) \mid \mathbf{inl}\ v \mid \mathbf{inr}\ v \mid \lambda x.\, e \mid [v] \mid \mathbf{ret}\, v \qquad \text{(Values)}$$

$$e, f ::= \cdots \mid \mathbf{rnd}\, v \qquad \text{(Terms)}$$

The evaluation rules are refined by defining two distinct step relations that capture the behavior of ideal and floating-point computations. Under the ideal semantics, the rounding operation behaves like the identity function, and under the floating-point semantics, the rounding operation behaves like a rounding function, ρ. For now, we make no assumptions on the function ρ, but further assumptions will be needed in our denotational semantics. It is sufficient to think of ρ as a well-defined rounding function as described in Section 2.1.

Definition 17. We define two step relations $e \mapsto_{id} e'$ and $e \mapsto_{fp} e'$ by augmenting the operational semantics in Figure 3.3 with the following rules:

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

3.4 Denotational Semantics

Our type system is designed to bound the distance between the outputs of two closely related computations: an ideal computation and its floating-point counterpart. In the previous section, we demonstrated how programs of graded monadic type can be executed under two different operational semantics, producing both an ideal, infinitely precise value and a floating-point value that may have incurred rounding error during execution. Formally, the operational semantics say nothing about the distance between the these two results. In this section, we will show that programs of type $M_{\varepsilon}\tau$ can be interpreted as pairs of computations that produce values separated by a distance of at most ε under a metric on $\mathbb{R}$. In the next section, we will connect the operational results from the previous section with the denotational results presented here to establish our main result, demonstrating that well-typed programs of type $M_{\varepsilon}\tau$ produce at most ε rounding error.

3.4.1 The Category of Metric Spaces

To formally capture the notion of the distance between program outputs, the denotational semantics for **NumFuzz** are based on the categorical semantics for Fuzz introduced by Azevedo de Amorim et al. (2017), where types are interpreted as *extended pseudo-metric spaces* and programs are interpreted and non-expansive maps in the category (Met) of these metric spaces.

Definition 18 (Extended Pseudo-Metric Space)**.** An *extended pseudo-metric space* $(\mathrm{X}, d_{\mathrm{X}})$ consists of a *carrier* set X, denoted by $|\mathrm{X}|$, and a *distance function* $d_{\mathrm{X}} : \mathrm{X} \times \mathrm{X} \to \mathbb{R}^{\geq 0} \cup \{\infty\}$ satisfying the following properties for all $a, b, c, \in \mathrm{X}$:

- reflexivity: $d(a, a) = 0$
- symmetry: $d(a, b) = d(b, a)$
- triangle inequality: $d(a, c) \leq d(a, b) + d(b, c)$

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

Now, before we define Met, we define non-expansive maps:

Definition 19. A *non-expansive map* $f : (\mathrm{X}, d_x) \to (\mathrm{Y}, d_{\mathrm{Y}})$ between extended pseudo-metric spaces consists of a set-map $f : \mathrm{X} \to \mathrm{Y}$ such that $d_{\mathrm{Y}}(f(x), f(x')) \leq d_{\mathrm{X}}(x, x')$.

Definition 20 (The Category of Metric Spaces (Met))**.** The category Met of *extended pseudo-metric spaces* is the category with the following data.

- The obejcts are extended pseudo-metric spaces.

- The morphisms from X to Y are non-expansive maps from X to Y.

The identity function is a non-expansive map, and non-expansive maps are closed under composition. Therefore, extended pseudo-metric spaces and non-expansive maps form a category Met.

The category Met supports several constructions that are useful for interpreting linear type systems:

- The Cartesian product $(\mathrm{A}, d_{\mathrm{A}})$ & $(\mathrm{B}, d_{\mathrm{B}})$ with carrier $\mathrm{A} \times \mathrm{B}$ and metric $d_{\mathrm{A} \,\&\, \mathrm{B}}((a, b), (a', b')) = \max(d_{\mathrm{A}}(a, a'), d_{\mathrm{B}}(b, b'))$.
- The tensor product $(\mathrm{A}, d_{\mathrm{A}}) \otimes (\mathrm{B}, d_{\mathrm{B}})$ with carrier $\mathrm{A} \times \mathrm{B}$ and metric $d_{\mathrm{A} \otimes \mathrm{B}}((a, b), (a', b')) = d_{\mathrm{A}}(a, a') + d_{\mathrm{B}}(b, b')$.
- Coproducts $(\mathrm{A}, d_{\mathrm{A}}) + (\mathrm{B}, d_{\mathrm{B}})$, where the carrier is the disjoint union $\mathrm{A} \uplus \mathrm{B}$ and the metric $d_{\mathrm{A}+\mathrm{B}}$ assigns distance ∞ to pairs of elements in different injections, and distance d_{A} or d_{B} to pairs of elements in A or B, respectively.
- Non-expansive functions $(\mathrm{A}, d_{\mathrm{A}}) \multimap (\mathrm{B}, d_{\mathrm{B}})$, where the carrier set is $\{f : \mathrm{A} \rightarrow \mathrm{B} \mid f \text{ non-expansive}\}$ and the metric is given by the supremum norm:

$$d_{\mathrm{A} \multimap \mathrm{B}}(f, g) = \sup_{a \in \mathrm{A}} d_{\mathrm{B}}(f(a), g(a)).$$

- Terminal objects are the singleton metric space $\mathrm{I} = (\{\star\}, d_{\mathrm{I}})$ with a single element and a constant distance function $d_{\mathrm{I}}(\star, \star) = 0$. Specifically, for every object $\mathrm{X} \in$ Met, there is a morphism $e_{\mathrm{X}} : \mathrm{X} \rightarrow \mathrm{I}$ given by $e_{\mathrm{X}} := x \mapsto \star$.

Theorem 3. The category $(\mathrm{Met}, \mathrm{I}, \otimes, \multimap)$ is a symmetric monoidal closed category (SMCC), where the unit object I is the metric space with a single element.

Theorem 3 follows by observation of the following: the functor $(- \otimes \mathrm{B})$ is left-adjoint to the functor $(\mathrm{B} \multimap -)$, so maps $f : \mathrm{A} \otimes \mathrm{B} \rightarrow \mathrm{C}$ can be curried to $\lambda(f) : \mathrm{A} \rightarrow (\mathrm{B} \multimap \mathrm{C})$, and uncurried.

3.4.2 A Graded Comonad on Met

As discussed in Section 2.2.2, graded comonadic types can be modeled by a categorical structure called a $\mathcal{S}$-*graded exponential comonad* (Brunel et al., 2014; Gaboardi et al., 2016; Katsumata, 2018). Given any metric space (A, d_A) and non-negative number r, there is an evident operation that scales the metric by r: $(A, r \cdot d_A)$. This operation can be extended to a graded comonad:

Definition 21. Let the pre-ordered semiring $\mathcal{S}$ be the extended non-negative real numbers $\mathbb{R}^{\geq 0} \cup \{\infty\}$ with the usual order, addition, and multiplication; $0 \cdot \infty$ and $\infty \cdot 0$ are defined to be 0. We define functors $\{D_s : \text{Met} \to \text{Met} \mid s \in \mathcal{S}\}$ such that $D_s : \text{Met} \to \text{Met}$ takes metric spaces (A, d_A) to metric spaces $(A, s \cdot d_A)$, and non-expansive maps $f : A \to B$ to $D_s f : D_s A \to D_s B$, with the same underlying map.

We also define the following associated natural transformations:

- For $s, t \in \mathcal{S}$ and $s \leq t$, the map $(s \leq t)_A : D_t A \to D_s A$ is the identity; note the direction.
- The map $m_{s,I} : I \to D_s I$ is the identity map on the singleton metric space.
- The map $m_{s,A,B} : D_s A \otimes D_s B \to D_s(A \otimes B)$ is the identity map on the underlying set.
- The map $w_A : D_0 A \to I$ maps all elements to the singleton.
- The map $c_{s,t,A} : D_{s+t} A \to D_s A \otimes D_t A$ is the diagonal map taking a to (a, a).
- The map $\epsilon_A : D_1 A \to A$ is the identity.
- The map $\delta_{s,t,A} : D_{s \cdot t} A \to D_s(D_t A)$ is the identity.

These maps are all non-expansive and it can be shown that they satisfy the diagrams (Gaboardi et al., 2016) defining a $\mathcal{S}$-graded exponential comonad.

3.4.3 A Graded Monad on Met

Our type system is designed to bound the distance between various kinds of program outputs. Intuitively, types should be interpreted as *metric spaces*, which are sets equipped with a distance function satisfying several standard axioms. Azevedo de Amorim et al. (2017) identified the following slight generalization of metric spaces as a suitable category to interpret *Fuzz*.

The categorical structures we have seen so far are enough to interpret the non-monadic fragment of our language, which is essentially the core of the *Fuzz* language (Azevedo de Amorim et al., 2017). As proposed by Gaboardi et al. (2016), this core language can model effectful computations using a graded monadic type, which can be modeled categorically by (i) a *graded strong monad*, and (ii) a *distributive law* modeling the interaction of the graded comonad and the graded monad.

The Neighborhood Monad

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

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

Definition 22. Let the pre-ordered monoid $\mathcal{R}$ be the extended non-negative real numbers $\mathbb{R}^{\geq 0} \cup \{\infty\}$ with the usual order and addition. The *neighborhood monad* is defined by the functors $\{T_r : \text{Met} \to \text{Met} \mid r \in \mathcal{R}\}$ and associated natural transformations as follows:

- The functor $\mathrm{T}_r : \mathrm{Met} \to \mathrm{Met}$ takes a metric space M to a metric space with underlying set

  $$|\mathrm{T}_r \mathrm{M}| \triangleq \{(x, y) \in \mathrm{M} \mid d_{\mathrm{M}}(x, y) \leq r\}$$

  and metric

  $$d_{\mathrm{T}_r \mathrm{M}}((x, y), (x', y')) \triangleq d_{\mathrm{M}}(x, x').$$

- The functor T_r takes a non-expansive function $f : \mathrm{A} \to \mathrm{B}$ to $\mathrm{T}_r f : \mathrm{T}_r \mathrm{A} \to \mathrm{T}_r \mathrm{B}$ with

  $$(\mathrm{T}_r f)((x, y)) \triangleq (f(x), f(y))$$

- For $r, q \in \mathcal{R}$ and $q \leq r$, the map $(q \leq r)_{\mathrm{A}} : \mathrm{T}_q \mathrm{A} \to \mathrm{T}_r \mathrm{A}$ is the identity.
- The unit map $\eta_{\mathrm{A}} : \mathrm{A} \to \mathrm{T}_0 \mathrm{A}$ is defined via: $\eta_{\mathrm{A}}(x) \triangleq (x, x)$.
- The graded multiplication map $\mu_{q,r,\mathrm{A}} : \mathrm{T}_q(\mathrm{T}_r \mathrm{A}) \to \mathrm{T}_{r+q} \mathrm{A}$ is defined via:

  $$\mu_{q,r,\mathrm{A}}((x, y), (x', y')) \triangleq (x, y').$$

The definitions of T_r are evidently functors. The associated maps are natural transformations, and define a graded monad (Katsumata, 2014; Fujii et al., 2016).

Lemma 6. Let $q, r \in \mathcal{R}$. For any metric space A, the maps $(q \leq r)_{\mathrm{A}}$, η_{A}, and $\mu_{q,r,\mathrm{A}}$ are non-expansive maps and natural in A.

The proof of Lemma 6 is provided in Appendix A.2.

Lemma 7. The functors T_r and its associated maps form a $\mathcal{R}$-graded monad on Met.

Proof. Establishing this fact requires checking that the diagrams in Definition 10 commute, which follows directly by unfolding definitions. □

As we will soon see, the monad structure defined so far is insufficient for interpreting the graded monadic sequencing rule (MLet). Simply put, the issue is that sequencing requires the input

and output types of programs to match, but the natural transformations we have defined for the neighborhood monad lack a mechanism to ensure this in our semantics. This issue also arises for the standard (not graded) monadic sequencing rule described in Section 2.2.1, for which Moggi (1991) originally proposed the use of *strong monads* to address the issue. The basic idea is that a strong monad is a monad along with an additional natural transformation $\sigma_{A,B} : A \otimes TB \to T(A \otimes B)$ known as the *strength map*. Generalizations of the notion of strong monads have been presented by Atkey (2009) and Katsumata (2014). We follows the presentation of Gaboardi et al. (2016):

Lemma 8. The neighborhood monad (Definition 22) together with the *tensorial strength maps* $st_{r,A,B} : A \otimes T_r B \to T_r(A \otimes B)$ defined as

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

for every $r \in \mathcal{R}$ form a $\mathcal{R}$-strong graded monad on Met.

We check the non-expansiveness (Definition 19) and naturality (Definition 12) of the tensorial strength map in Appendix A.2.

A Graded Distributive Law

Gaboardi et al. (2016) showed that languages supporting graded coeffects and graded effects can be modeled with a graded comonad, a graded monad, and a graded distributive law. In our setting, we have the following family of maps defining the interaction between the neighborhood monad $(T_r)_{r \in \mathcal{R}}$ and the graded comonad $(D_s)_{s \in \mathcal{S}}$.

Lemma 9. Let $s \in \mathcal{S}$ and $r \in \mathcal{R}$ be grades, and let A be a metric space. Then identity map on the carrier set $|A| \times |A|$ is a non-expansive map

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

Moreover, these maps are natural in A.

It is straightforward to verify the non-expansiveness (Definition 19) and naturality (Definition 12) of the distributive map. Details are provided in Appendix A.2.

Similarly, it is straightforward to show that the maps $\lambda_{s,r,A}$ form a graded distributive law in the sense of Gaboardi et al. (2016): for $s \leq s'$ and $r \leq r'$ the identity map $T_{s \cdot r}(D_s A) \to T_{s' \cdot r'}(D_{s'} A)$ is also natural in A, and the four diagrams required for a graded distributive law all commute (Gaboardi et al., 2016, Fig. 8), but since we do not rely on these properties we will omit these details.

3.4.4 Interpreting NumFuzz

We are now ready to define an interpretation of **NumFuzz** in the category Met.

Interpreting Types

We interpret each type τ as a metric space $[\![\tau]\!]$, using the constructions described in the previous sections: the basic constrictions in Met along with the graded comonad and graded monad.

Definition 23. We define the interpretation of types by induction on the type syntax:

$$
\begin{array}{ll}
[\![\mathrm{unit}]\!] \triangleq I = (\{\star\}, 0) & [\![\mathrm{num}]\!] \triangleq (R, d_R) \\
[\![A \otimes B]\!] \triangleq [\![A]\!] \otimes [\![B]\!] & [\![A \,\&\, B]\!] \triangleq [\![A]\!] \,\&\, [\![B]\!] \\
[\![A + B]\!] \triangleq [\![A]\!] + [\![B]\!] & [\![A \multimap B]\!] \triangleq [\![A]\!] \multimap [\![B]\!] \\
[\![!_s A]\!] \triangleq D_s [\![A]\!] & [\![M_r A]\!] \triangleq T_r [\![A]\!]
\end{array}
$$

It is not yet necessary to fix the interpretation of the base type num. For now, (R, d_R) can be any metric space.

Interpreting Judgments

We interpret a typing judgment of the form $\Gamma \vdash e : \tau$ as a morphism in Met from the metric space $[\![\Gamma]\!]$ to the metric space $[\![\tau]\!]$. Since all morphisms in Met are non-expansive, this interpretation ensures a version of *metric preservation* for **NumFuzz**, which is the central language guarantee for *Fuzz* (Reed and Pierce, 2010; Azevedo de Amorim et al., 2017). Intuitively, this property guarantees that well-typed programs respect the sensitivity bound assigned by the type system.

To interpret typing judgments, we first require an interpretation of typing contexts, as well as few auxiliary maps. The interpretation of typing contexts is defined inductively as follows:

$$[\![\cdot]\!] \triangleq I = (\{\star\}, 0)$$

$$[\![\Gamma, x :_s \tau]\!] \triangleq [\![\Gamma]\!] \otimes D_s[\![\tau]\!]$$

The interpretation of typing judgments is defined inductively over the typing derivation (Figure 3.2). Since many of the typing rules rely on the scaling $(s \cdot \Gamma)$ and summing $(\Gamma + \Delta)$ of contexts, we need to give a clear semantics to these context operations.

First, given any binding $x :_r \tau \in \Gamma$, there is a non-expansive map from $[\![\Gamma]\!]$ to $[\![\tau]\!]$ projecting out the x-th position. Formally, projections are defined via the weakening maps $(0 \leq s)_A ; w_A : D_s A \to I$ and the unitors, but we will use notation that treats an element $\gamma \in [\![\Gamma]\!]$ as a function, so that $\gamma(x) \in [\![\tau]\!]$. We use this notation to state the following lemma about the sum of two contexts.

Lemma 10. Let Γ and Δ such that $\Gamma + \Delta$ is defined. Then there is a non-expansive map $c_{\Gamma,\Delta}$: $[\![\Gamma + \Delta]\!] \to [\![\Gamma]\!] \otimes [\![\Delta]\!]$ given by:

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

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

Finally, we use the graded comonad to interpret the scaling of a context:

Lemma 11. Let Γ be a context and $s \in \mathcal{S}$ be a sensitivity. Then the identity function is a non-expansive map from $[\![s \cdot \Gamma]\!] \to D_s[\![\Gamma]\!]$.

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

Definition 24. (Interpretation of **NumFuzz** Terms.) Fix $\rho : \mathrm{R} \to \mathrm{R}$ to be a (set) function such that for every $r \in \mathrm{R}$ we have

$$d_{\mathrm{R}}(r, \rho(r)) \leq \varepsilon.$$

Furthermore, for every operation $\{\mathrm{op} : \sigma \multimap \tau\} \in \Sigma$ fix an interpretation $[\![\mathbf{op}]\!] : [\![\sigma]\!] \to [\![\tau]\!]$ such that, for every closed value $\emptyset \vdash v : \sigma$, we have $[\![\mathbf{op}]\!]([\![v]\!]) = [\![op(v)]\!]$. Given these assumptions, we can interpret each well-typed program $\Gamma \vdash e : \tau$ as a non-expansive map $[\![\Gamma \vdash e : \tau]\!] : [\![\Gamma]\!] \to [\![\tau]\!]$ by induction on the typing derivation, via case analysis on the last rule.

We demonstrate the construction for several cases here, including all cases involving terms of monadic type. The remaining cases can be found in Appendix A.3. To reduce notation, we elide the the unitors $\lambda_{\mathrm{A}} : \mathrm{I} \otimes \mathrm{A} \to \mathrm{A}$ and $\rho_{\mathrm{A}} : \mathrm{A} \otimes \mathrm{I} \to \mathrm{I}$; the associators $\alpha_{\mathrm{A,B,C}} : (\mathrm{A} \otimes \mathrm{B}) \otimes \mathrm{C} \to \mathrm{A} \otimes (\mathrm{B} \otimes \mathrm{C})$; and the symmetries $\sigma_{\mathrm{A,B}} : \mathrm{A} \otimes \mathrm{B} \to \mathrm{B} \otimes \mathrm{A}$.

(Const). Define $[\![\Gamma \vdash k : \mathrm{num}]\!] : [\![\Gamma]\!] \to [\![\mathrm{num}]\!]$ to be the constant function returning $k \in \mathrm{R}$.

(Op). By assumption, we have an interpretation $[\![\mathbf{op}]\!]$ for every operation in the signature Σ. We can then define:

$$[\![\Gamma \vdash \mathbf{op}(v) : \tau]\!] \triangleq [\![\Gamma \vdash v : \sigma]\!]; [\![\mathbf{op}]\!]$$

(Ret). Let $f = [\![\Gamma \vdash v : \tau]\!]$. Define

$$[\![\Gamma \vdash \mathbf{ret}\, v : \mathrm{M}_0 \tau]\!] \triangleq f; \eta_{[\![\tau]\!]}.$$

(MSub). Let $f = [\![\Gamma \vdash e : M_r\tau]\!]$. Define

$$[\![\Gamma \vdash e : M_{r'}\tau]\!] \triangleq f; (r \leq r')_{[\![\tau]\!]}.$$

(Rnd). Let $f = [\![\Gamma \vdash k : \text{num}]\!]$. Define

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

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

(MLet). Let $f = [\![\Gamma \vdash e : M_r\sigma]\!]$ and $g = [\![\Theta, x :_s \sigma \vdash e' : M_q\tau]\!]$. Define

$$[\![s \cdot \Gamma + \Theta \vdash \textsf{let}_{\textsf{M}}\ x = e\ \textsf{in}\ f : M_{s\cdot r+q}\tau]\!] \triangleq c_{s\cdot[\![\Gamma]\!],[\![\Theta]\!]}; h_1; h_2.$$

The maps h_1 and h_2 are defined as follows. First, apply the comonad to f and then compose with the distributive law to get:

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

Repeatedly pre-composing with the map $m_{s,A,B} : D_sA \otimes D_sB \to D_s(A \otimes B)$ produces a map $[\![s \cdot \Gamma]\!] \to T_{s\cdot r}D_s[\![\sigma]\!]$. Then, composing in parallel with $id_{[\![\Theta]\!]}$ and then post-composing with the strength map $st_{s\cdot r,[\![\Theta]\!],[\![\sigma]\!]}$ yields:

$$h_1 = ((m; D_s f; \lambda_{s,r,\sigma}) \otimes id_{[\![\Theta]\!]}); \otimes st_{s\cdot r,[\![\Theta]\!],[\![\sigma]\!]} : [\![\Theta]\!] \otimes [\![s \cdot \Gamma]\!] \to T_{s\cdot r}([\![\Theta]\!] \otimes D_s[\![\sigma]\!])$$

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

$$h_2 = T_{s\cdot r}g; \mu_{s\cdot r,q,[\![\sigma]\!]} : T_{s\cdot r}([\![\Theta]\!] \otimes D_s[\![\sigma]\!]) \to T_{s\cdot r+q}[\![\tau]\!].$$

The composition $h_1; h_2$ yields a map from $[\![\Theta]\!] \otimes [\![s \cdot \Gamma]\!]$ to $T_{s\cdot r+q}[\![\tau]\!] = [\![M_{s\cdot r+q}\tau]\!]$, as required.

Ideal and Floating-Point Denotational Semantics

The metric semantics we have just defined interprets each **NumFuzz** program as a non-expansive map. Ultimately, we aim to show that values of monadic type $M_r\sigma$ are interpreted as pairs of values: the first being the result under the ideal operational semantics and the second being the result under an approximate, floating-point operational semantics. These operational semantics were defined in Section 3.3.

In this section, we define two *denotational semantics* that capture the ideal and floating-point behaviors of our programs, respectively. We then relate the metric semantics from the previous section to the ideal and floating-point denotational semantics in a *pairing lemma* (Lemma 18).

We develop both the ideal and floating-point semantics in **Set**, where maps are not required to be non-expansive. Intuitively, while we can define an ideal semantics of well-typed programs as non-expansive maps in Met, programs under the floating-point semantics are not guaranteed to be non-expansive—a tiny change in the input to a rounding operation could lead to a relatively large change in the rounded output. We therefore develop both semantics in **Set**, where maps are not required to be non-expansive.

Definition 25. Let $\Gamma \vdash e : \tau$ be a well-typed program. We can define two semantics in **Set**:

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

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

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

$$(\!|M_q\tau|\!)_{id} = (\!|!_s\ \tau|\!)_{id} \triangleq (\!|\tau|\!)_{id}$$

$$(\!|M_q\tau|\!)_{fp} = (\!|!_s\ \tau|\!)_{fp} \triangleq (\!|\tau|\!)_{fp}$$

The ideal and floating point interpretations of well-typed programs are both straightforward, by induction on the derivation of the typing judgment. The only interesting case is for the rule (Rnd)

for the rounding operation:

$$(\!|\Gamma \vdash \mathbf{rnd}\ k : \mathrm{M}_\varepsilon \mathrm{num}|\!)_{id} \triangleq (\!|\Gamma \vdash k : \mathrm{num}|\!)_{id}$$

$$(\!|\Gamma \vdash \mathbf{rnd}\ k : \mathrm{M}_\varepsilon \mathrm{num}|\!)_{fp} \triangleq (\!|\Gamma \vdash k : \mathrm{num}|\!)_{fp}; \rho$$

where $\rho : \mathrm{R} \to \mathrm{R}$ is a rounding function.

We now relate the metric semantics with the ideal and floating-point semantics we have just defined. Let $\mathrm{U} : \mathrm{Met} \to \mathbf{Set}$ be the forgetful functor mapping each metric space to its underlying set, and each morphism of metric spaces to its underlying function on sets. We have:

Lemma 12 (Pairing)**.** Let $\emptyset \vdash e : \mathrm{M}_r\mathrm{num}$. Then we have:

$$\mathrm{U}[\![e]\!] = \langle (\!|e|\!)_{id}, (\!|e|\!)_{fp} \rangle$$

in **Set**. The first projection of $\mathrm{U}[\![e]\!]$ is $(\!|e|\!)_{id}$, and the second projection is $(\!|e|\!)_{fp}$.

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

Case. For the base case $n = 0$, we know that e reduces to either **ret** v or **rnd** v. We can conclude since by inversion v must be a real constant, and $\mathrm{U}[\![v]\!] = (\!|v|\!)_{id} = (\!|v|\!)_{fp}$.

Case. For the inductive case $n = m + 1$, we know that e reduces to $\mathbf{let_M}\ x = \mathbf{rnd}\ k\ \mathbf{in}\ f$. By the logical relation, we have $f[v/x] \in \mathrm{R}^m_{\mathrm{M}_r\mathrm{num}}$ for all values v such that $\emptyset \vdash v : \mathrm{num}$. By induction we then have:

$$(\!|f[k/x]|\!)_{id} \triangleq (\!|\mathbf{let_M}\ x = \mathbf{rnd}\ k\ \mathbf{in}\ f|\!)_{id} = \mathrm{U}[\![f[k/x]]\!]; \pi_1$$

$$(\!|f[\rho(k)/x]|\!)_{fp} \triangleq (\!|\mathbf{let_M}\ x = \mathbf{rnd}\ k\ \mathbf{in}\ f|\!)_{fp} = \mathrm{U}[\![f[\rho(k)/x]]\!]; \pi_2$$

Thus we just need to show:

$$\mathrm{U}[\![\mathbf{let_M}\ x = \mathbf{rnd}\ k\ \mathbf{in}\ f]\!] = \langle \mathrm{U}[\![f[k/x]]\!]; \pi_1, \mathrm{U}[\![f[\rho(k)/x]]\!]; \pi_2 \rangle$$

where we have judgments $\emptyset \vdash \textbf{rnd}\ k : M_r\text{num}$ and $x :_s \text{num} \vdash f : M_q\text{num}$. We can conclude by applying the substitution lemma (Lemma 24) and unfolding the definition of $[\![\textsf{let}_{\textsf{M}}\ x = \textsf{rnd}\ k\ \textsf{in}\ f]\!]$. □

3.5 Forward Error Soundness

The primary guarantee for NumFuzz is forward error soundness, which ensures that well-typed programs of graded monadic type satisfy the error bound indicated by their type. In this section, we prove this guarantee by demonstrating that the ideal and floating-pint operational semantics, as described in Section 3.3, are computationally sound: stepping a well-typed NumFuzz term does not change its semantics. Forward error soundness then follows as a corollary to computational soundness and the pairing lemma (Lemma 12), which relates the metric semantics to both the ideal and floating-point semantics.

Lemma 13 (Computational Soundness)**.** Let $\emptyset \vdash e : \tau$ be a well-typed closed term, and suppose $e \mapsto_{id} e'$. Then there is a derivation of $\emptyset \vdash e' : \tau$ and the semantics of both derivations are equal: $(\!|\vdash e : \tau|\!)_{id} = (\!|\vdash e' : \tau|\!)_{id}$. The same holds for the floating-point denotational and operational semantics.

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

Case: $\textbf{rnd}\ k \mapsto_{id} \textbf{ret}\ k$. Suppose that $\emptyset \vdash \textbf{rnd}\ k : M_q\text{num}$, where $\varepsilon \le q$. Then the rules (Ret) and (MSub) can be used to derive the judgment $\cdot \vdash \textbf{ret}\ k : M_q\text{num}$, and

$$(\!|\emptyset \vdash \textbf{rnd}\ k : M_q\text{num}|\!)_{id} = (\!|\emptyset \vdash k : \text{num}|\!)_{id} = (\!|\emptyset \vdash \textbf{ret}\ k : M_q\text{num}|\!)_{id}.$$

Case: $\textbf{rnd}\ k \mapsto_{fp} \textbf{ret}\ \rho(k)$. Suppose that $\emptyset \vdash \textbf{rnd}\ k : M_q\text{num}$, where $\varepsilon \le q$. Then the rules (Ret) and (MSub) can be used to derive the judgment $\emptyset \vdash \textbf{ret}\ k : M_q\text{num}$, and

$$(\!|\emptyset \vdash \textbf{rnd}\ k : M_q\text{num}|\!)_{fp} = (\!|\emptyset \vdash \rho(k) : \text{num}|\!)_{fp} = (\!|\emptyset \vdash \textbf{ret}\rho(k) : M_q\text{num}|\!)_{fp}.$$

$\square$

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

Corollary 1 (Forward Error Soundness)**.** Let $\emptyset \vdash e : M_r \text{num}$ be a well-typed program. Then $e \mapsto^*_{id}$ **ret** v_{id} and $e \mapsto^*_{fp}$ **ret** v_{fp} such that $d_{[\![\text{num}]\!]}(\llparenthesis v_{id} \rrparenthesis_{id}, \llparenthesis v_{fp} \rrparenthesis_{fp}) \leq r$.

Proof. Under the ideal and floating point semantics, the only values of monadic type are of the form **ret** v. Since these operational semantics are type-preserving and normalizing, we must have

$$e \mapsto^*_{id} \textbf{ret}\ v_{id} \quad \text{and} \quad e \mapsto^*_{fp} \textbf{ret}\ v_{fp}.$$

By computational soundness (Lemma 13), we have

$$\llparenthesis e \rrparenthesis_{id} = \llparenthesis \textbf{ret}\ v_{id} \rrparenthesis_{id} \quad \text{and} \quad \llparenthesis e \rrparenthesis_{fp} = \llparenthesis \textbf{ret}\ v_{fp} \rrparenthesis_{fp}.$$

Now, by pairing (Lemma 12), we have

$$U[\![e]\!] = \langle \llparenthesis \textbf{ret}\ v_{id} \rrparenthesis_{id}, \llparenthesis \textbf{ret}\ v_{fp} \rrparenthesis_{fp} \rangle.$$

Since the forgetful functor is the identity on morphisms, we have $U[\![e]\!] = [\![e]\!] : I \to T_r[\![\text{num}]\!]$ in Met. By definition, $\llparenthesis \textbf{ret}\ v_{id} \rrparenthesis_{id} = \llparenthesis v_{id} \rrparenthesis_{id}$ and $\llparenthesis \textbf{ret}\ v_{fp} \rrparenthesis_{fp} = \llparenthesis v_{fp} \rrparenthesis_{fp}$. Thus, $\langle \llparenthesis v_{id} \rrparenthesis_{id}, \llparenthesis v_{fp} \rrparenthesis_{fp} \rangle$ is an element of $T_r[\![num]\!]$ and we conclude by the definition of the monad T_r:

$$d_{[\![\text{num}]\!]}(\llparenthesis v_{id} \rrparenthesis_{id}, \llparenthesis v_{fp} \rrparenthesis_{fp}) \leq r.$$

$\square$

3.6 Example NumFuzz Programs

This section illustrates how to instantiate **NumFuzz** in practice. Recall that **NumFuzz** is parameterized by a set R of numeric constants of type num, a fixed constant ε representing an upper

bound on the rounding error produced by evaluating a rounding function, and a signature Σ defining the primitive operations in the language. Our language guarantee of forward error soundness (Corollary 1) holds under the following assumptions on these parameters.

First, the interpretation $[\![\mathrm{num}]\!] = (R, d_R)$ of the numeric type must be a metric space. Second, for every operation $\{\mathrm{op} : \sigma \multimap \tau\} \in \Sigma$, we must fix an interpretation $[\![\mathbf{op}]\!] : [\![\sigma]\!] \to [\![\tau]\!]$ such that, for every closed value $\emptyset \vdash v : \sigma$, we have $[\![\mathbf{op}]\!]([\![v]\!]) = [\![op(v)]\!]$. Simply stated, this means any primitive operation included in an instantiation of **NumFuzz** must be a non-expansive map. Third, the interpretation $[\![\mathbf{rnd}]\!]$ of the primitive rounding operation must use a well-defined rounding function $\rho : R \to R$ such that, for every $r \in R$, we have:

$$d_R(r, \rho(r)) \leq \varepsilon.$$

This choice of the rounding function fixes the language constant ε.

Now, we will explore how instances of these parameters can be soundly chosen in practice. In the next section, we evaluate how an implementation of this instance compares to existing sound tools that automatically bound the relative rounding error of floating-point programs.

Interpreting num. If we interpret our numeric type num as the set of strictly positive real numbers $\mathbb{R}_{>0}$ with the relative precision (RP) metric (Definition 3), then we can use **NumFuzz** to perform a relative error analysis as described by Olver (1978).

Defining primitive operations. Using the RP metric, we can extend the language with four primitive arithmetic operations, typed as follows:

$$\begin{aligned}
\text{add} &: (\text{num} \;\&\; \text{num}) \multimap \text{num} \\
\text{mul} &: (\text{num} \otimes \text{num}) \multimap \text{num} \\
\text{div} &: (\text{num} \otimes \text{num}) \multimap \text{num} \\
\text{sqrt} &: \;![0.5]\text{num} \multimap \text{num}
\end{aligned}$$

Above, we use the syntax $![s]$ in place of $!_s$ and the syntax $M[q]$ in place of M_q for readability. This is also the syntax of our implementation of **NumFuzz**, which we will introduce in the next section.

In order to soundly add these operations as primitives to the language, we must verify that their interpretations are each non-expansive maps. If we fix the interpretation of these operations as their natural mathematical counterparts over the positive real numbers, then these interpretations are non-expansive functions.

As an example, consider the operation add : (num & num) $\multimap$ num. Showing non-expansiveness amounts to verifying, for every $a, b, a', b' \in \mathbb{R}_{>0}$, that

$$\text{RP}((a+b), (a'+b')) \leq \max(\text{RP}(a,a'), \text{RP}(b,b')).$$

If we let $\alpha = \text{RP}(a, a')$ and $\beta = \text{RP}(b, b')$, then we have that

$$\begin{aligned}
\text{RP}((a+b), (a'+b')) &\leq \text{RP}((a+b) \cdot \max(e^{\alpha}, e^{\beta}), (a+b)) \\
&= \max(\text{RP}(a,a'), \text{RP}(b,b'))
\end{aligned}$$

as required. Similarly, for the operation mul : (num $\otimes$ num) $\multimap$ num. Showing non-expansiveness amounts to verifying, for every $a, b, a', b' \in \mathbb{R}_{>0}$, that

$$\text{RP}((a \cdot b), (a' \cdot b')) \leq \text{RP}(a, a') + \text{RP}(b, b').$$

Recall that when we first introduced the multiplicative ($\otimes$) product and additive (&) product in Section 3.2, the descriptors *multiplicative* and *additive* inherited from linear logic conflicted with

the context operations in sensitivity type systems. Here, however, they are fitting: multiplication is most naturally typed with the multiplicative product, while addition is most naturally typed with the additive product. While we could also type add with the multiplicative product, the type is coarser than necessary and would ultimately result in looser rounding error bounds. (Note that while we can soundly type add using both the additive and multiplicative product, we can not soundly type mul with the additive product). To fully understand why, we first need to interpret our rounding operation **rnd**. For now, suppose we added addition typed with the multiplicative product, $\mathrm{add}' : (\mathrm{num} \otimes \mathrm{num}) \multimap \mathrm{num}$, to the primitive operations. Then, consider the following typing derivations:

$$\dfrac{\dfrac{x :_1 \mathrm{R} \vdash x : \mathrm{R} \qquad x :_1 \mathrm{R} \vdash x : \mathrm{R}}{x :_1 \mathrm{R} \vdash \langle x, x\rangle : \mathrm{R} \,\&\, \mathrm{R}}\,(\&\ \mathrm{I}) \qquad \{\mathrm{add} : \mathrm{R} \,\&\, \mathrm{R} \multimap \mathrm{R} \in \Sigma\}}{x :_1 \mathrm{R} \vdash \mathrm{add}\ \langle x, x\rangle : \mathbb{R}}\,(\mathrm{Op})$$

$$\dfrac{\dfrac{x :_1 \mathrm{R} \vdash x : \mathrm{R} \qquad x :_1 \mathrm{R} \vdash x : \mathbb{R}}{x :_2 \mathrm{R} \vdash (x, x) : \mathrm{R} \,\&\, \mathbb{R}}\,(\otimes\ \mathrm{I}) \qquad \{\mathrm{add}' : \mathrm{R} \otimes \mathrm{R} \multimap \mathrm{R} \in \Sigma\}}{x :_2 \mathrm{R} \vdash \mathrm{add}'\ (x, x) : \mathbb{R}}\,(\mathrm{Op})$$

In the first derivation, the expression $\mathrm{add}\ \langle x, x\rangle$ is 1-sensitive in x. This is because, in the left branch where the pair introduction rule (& I) is used, the pair $\langle x, x\rangle$ is typed in the same environment as the components, resulting in the pair being 1-sensitive in x. In contrast, in the second derivation, the pair introduction rule ($\otimes$ I) is used, and the environments used to type each component are summed as

$$x :_2 \mathrm{R} = x :_1 \mathrm{R} + x :_1 \mathrm{R},$$

making the pair 2-sensitive in x. The overall expression $\mathrm{add}'\ (x, x)$ is therefore 2-sensitive in x. We will soon see how this difference would propagate through an error analysis.

Choosing the rounding function. Given our interpretation $[\![\mathrm{num}]\!] = (\mathbb{R}_{>0}, RP)$ for the numeric type, we require the rounding function ρ to be a function such that for every $x \in \mathbb{R}_{>0}$, we have

$\mathrm{RP}(x, \rho(x)) \leq \epsilon$; that is, the rounding function must satisfy an accuracy guarantee with respect to the metric RP on $\mathbb{R}_{>0}$. If we choose $\rho_{\mathrm{RU}} : \mathbb{R} \to \mathbb{R}$ to be round towards $+\infty$, then by Lemma 1 we have that $\mathrm{RP}(x, \rho(x)) \leq u$, where u is the unit roundoff.

Now, using the **rnd** operation, we can write the floating-point counterparts of the primitive operations add, mul, div, and sqrt defined above in **NumFuzz**. Consider the example for add:

$$\mathrm{addfp} : (\mathrm{num}\ \&\ \mathrm{num}) \multimap \mathrm{M}[u]\mathrm{num}$$

$$\mathrm{addfp}\ z \triangleq \textbf{let}\ y = \mathrm{add}\ z\ \textbf{in rnd}\ y$$

Observe that the type $\mathrm{M}[u]$ reflects our interpretation of **rnd**, which produces at most unit roundoff (u) rounding error when evaluated. Error soundness (Corollary 1) guarantees that the function addfp approximates an infinitely precise computation with relative precision u. Similarly, for the function add, we could lift the result to monadic type using the return construct (**ret**), and error soundness would guarantee that the function is an exact approximation to an infinitely precise computation. The functions mul, div, and sqrt can be similarly defined, with the following type signatures:

$$\mathrm{mulfp} : (\mathrm{num} \otimes \mathrm{num}) \multimap \mathrm{M}[u]\mathrm{num}$$

$$\mathrm{divfp} : (\mathrm{num} \otimes \mathrm{num}) \multimap \mathrm{M}[u]\mathrm{num}$$

$$\mathrm{sqrtfp} : \ ![0.5]\mathrm{num} \multimap \mathrm{M}[u]\mathrm{num}$$

We can now see how the choice of type signature for our primitive operations impacts the result of a rounding error analysis. First, let $\mathrm{addfp}' : (\mathrm{num} \otimes \mathrm{num}) \multimap \mathrm{M}[u]\mathrm{num}$ be the program that rounds the result of the primitive operation $\mathrm{add}' : \mathrm{num} \otimes \mathrm{num} \multimap \mathrm{num}$, described above. Then,

consider the following **NumFuzz** programs[1]:

$$\text{fun1} : \text{num} \multimap M[2u]\text{num}$$

$$\text{fun1} \triangleq \lambda y.\ \textbf{let}_{\textbf{M}}\ x = \textbf{rnd}\ y\ \textbf{in}\ \text{addfp}\ \langle x, x\rangle$$

$$\text{fun2} :\ ![2]\text{num} \multimap M[3u]\text{num}$$

$$\text{fun2} \triangleq \lambda y.\ \textbf{let}\ [y'] = y\ \textbf{in}\ \textbf{let}_{\textbf{M}}\ x = \textbf{rnd}\ y'\ \textbf{in}\ \text{addfp}'\ (x, x)$$

The programs fun1 and fun2 both take a value of numeric type as an input, round this value, and sum the result with itself using floating-point addition. However, fun2 is 2-sensitive in its argument, while fun1 is only 1-sensitive in its argument. We will see that this is because fun1 uses addfp : $(\text{num} \,\&\, \text{num}) \multimap M[u]\text{num}$ for floating-point addition, while fun2 uses $\text{addfp}' : (\text{num} \otimes \text{num}) \multimap M[u]\text{num}$ for floating-point addition. The consequence of this choice is seen in the return types of the function signatures: the type of fun1 guarantees at most $2u$ roundoff error, while the type of fun2 guarantees a looser bound, of at most $3u$ roundoff error.

To see how this works, consider the corresponding derivations of the typing judgments. For fun1, we have the following valid derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{y :_1 \mathrm{R} \vdash y : \mathrm{R}}\,(\text{Var})}{y :_1 \mathrm{R} \vdash \textbf{rnd}\ y : M_u \mathrm{R}}\,(\text{Rnd}) \qquad \dfrac{\begin{array}{c}[\text{A1}]\\ \vdots \\ x :_1 \mathrm{R} \vdash \text{add}\ \langle x, x\rangle : \mathrm{R}\end{array} \qquad \begin{array}{c}[\text{B1}]\\ \vdots \\ y :_1 \mathrm{R} \vdash \textbf{rnd}\ y : \mathrm{R}\end{array}}{x :_1 \mathrm{R} \vdash \text{addfp}\ \langle x, x\rangle : M_u \mathbb{R}}\,(\text{Let})}{1 \cdot y :_1 \mathrm{R} \vdash \textbf{let}_{\textbf{M}}\ x = \textbf{rnd}\ y\ \textbf{in}\ \text{addfp}\ \langle x, x\rangle : M_{1 \cdot u + u}\mathbb{R}}\,(\text{MLet})}{\emptyset \vdash \text{fun1} : M_{2u}\mathbb{R}}\,(\text{Abs})$$

In the right branch of the derivation above, we have already seen the derivation [A1], showing that the expression add $\langle x, x\rangle$ is 1-sensitive in x.

For fun2, the derivation is:

[1] Recall that the **let** $[y'] = y$ **in** $-$ deconstructor is used to sequence values with comonadic type.

$$\dfrac{\dfrac{\dfrac{}{y :_1 \mathrm{R} \vdash y : \mathbb{R}}\ (\text{Var})}{y :_1 \mathrm{R} \vdash \textbf{rnd}\ y : \mathrm{M}_u\mathbb{R}}\ (\text{Rnd}) \qquad \dfrac{\begin{matrix}[\text{A2}] \\ \vdots \\ x :_2 \mathrm{R} \vdash \text{add}'\ (x, x) : \mathrm{R}\end{matrix} \qquad \begin{matrix}[\text{B2}] \\ \vdots \\ y :_1 \mathrm{R} \vdash \textbf{rnd}\ y : \mathbb{R}\end{matrix}}{x :_2 \mathrm{R} \vdash \text{addfp}'\ (x, x) : \mathrm{M}_u\mathbb{R}}\ (\text{Let})}{\dfrac{2 \cdot y :_1 \text{num} \vdash \textbf{let}_{\textbf{M}}\ x = \textbf{rnd}\ y\ \textbf{in}\ \text{addfp}'\ (x, x) : \mathrm{M}_{2 \cdot u + u}\text{num}}{\emptyset \vdash \text{fun2} : \mathrm{M}_{3u}\text{num}}\ (\text{Abs})}\ (\text{MLet})$$

In the right branch of the derivation above, we have already seen the derivation [A2] showing that the expression $\text{add}'\ (x, x)$ is 2-sensitive in x. These derivations illustrate the importance of using the best possible type for primitive operations.

Now that we have described an instantiation of **NumFuzz** and justified our choice of the types for primitive operations in this instance, we can proceed to consider some more detailed examples.

Examples

The examples presented in this section use the actual syntax of an implementation of **NumFuzz**, introduced in Section 3.7. The implementation closely follow the language syntax presented in Figure 3.2, with some additional syntactic sugar, defined below:

$$(\mathsf{x} = \mathsf{e};\ \mathsf{f}) \equiv \textbf{let}\ x = e\ \textbf{in}\ f \qquad \text{(pure sequencing)}$$

$$(\mathsf{let}\ \mathsf{x} = \mathsf{v};\ \mathsf{f}) \equiv \textbf{let}_{\textbf{M}}\ x = e\ \textbf{in}\ f \qquad \text{(monadic sequencing)}$$

$$(\mathsf{let}\ [\mathsf{x}] = \mathsf{v};\ \mathsf{f}) \equiv \textbf{let}\ [x] = v\ \textbf{in}\ f \qquad \text{(comonadic sequencing)}$$

For top-level programs, we write (function ID args {v} e) to denote the let-binding $\textbf{let}\ \mathsf{ID} = v\ \textbf{in}\ e$, where v is a lambda term with arguments args. We write additive and multiplicative pairs as ⟨–,–⟩ and (–,–), respectively. Finally, for types, we write M[u]num to represent monadic types with a numeric grade u and we write ![s] to represent comonadic types with a numeric grade s.

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

```
// Multiply-add operation
function MA (x: num, y: num, z: num)
{
 let a = mulfp (x,y); // monadic sequencing of multiplication with rounding
 addfp ⟨a,z⟩
}
```

We can soundly type the program MA in **NumFuzz** as

$$\mathsf{MA} : \mathsf{num} \multimap \mathsf{num} \multimap \mathsf{num} \multimap M[2u]\mathsf{num},$$

where the index 2u on the return type indicates that the roundoff error is at most twice the unit roundoff, due to the two separate rounding operations in mulfp and addfp. The monadic sequencing let a = mulfp (x,y) allows us to use the result of mulfp (x,y)—which has a monadic type— as an argument to addfp, which accepts pure (non-monadic) numeric arguments.

Multiply-add is extremely common in numerical code, and modern architectures typically support a *fused* multiply-add (FMA) operation. This operation performs a multiplication followed by an addition, $x \cdot y + z$, as though it were a single floating-point operation. Consequently, the FMA operation incurs a single rounding error instead of two. The **NumFuzz** implementation of the FMA operation is:

```
// Fused multiply-add operation
function FMA (x: num, y: num, z: num) {
 a = mul (x,y); // multiplication without rounding
 b = add ⟨a,z⟩; // addition without rounding
 rnd b
}
```

We can soundly type the program MA in **NumFuzz** as

$$\mathsf{FMA} : \mathsf{num} \multimap \mathsf{num} \multimap \mathsf{num} \multimap M[u]\mathsf{num}.$$

The index u on the return type of FMA is reflects the reduced rounding error when compared to MA.

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

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

and computes the result using only n multiplications and n additions. Using **NumFuzz**, we can perform an error analysis on a version of Horner's scheme that uses the FMA operation to evaluate second-order polynomials of the form $p(\vec{a}, x) = a_2 x^2 + a_1 x + a_0$ where x and all a_is are non-zero positive constants. The implementation Horner2 in **NumFuzz**:

```
// Horner's scheme for a second order polynomial
function Horner2
 (a0: num, a1: num, a2: num, x: ![2]num)
{
 let [x1] = x; // comonadic sequencing
 s1 = FMA a2 x1 a1;
 let z = s1; // monadic sequencing
 FMA z x1 a0
}
```

We can soundly type Horner2 in **NumFuzz** with the following signature:

$$\text{Horner2} : \text{num} \multimap \text{num} \multimap \text{num} \multimap ![2]\text{num} \multimap M[2u]\text{num},$$

which guarantees that Horner2 produces at most 2u rounding error, measured as relative precision.

As a consequence of the metric interpretation of programs (Section 3.4.4), the type of Horner2 also ensures bounded sensitivity of the ideal semantics, which corresponds to the polynomial

$$p(\vec{a}, x) = a_2 x^2 + a_1 x + a_0.$$

For any $\vec{a}, \vec{u} \in \mathbb{R}^3_{>0}$, and for any $x, x' \in \mathbb{R}_{>0}$, we can measure the sensitivity of Horner2 to rounding errors introduced by the inputs: if $\text{RP}(x, x') = q$ and $\text{RP}(a_i, u_i) = r$ for each $i \in 0, 1, 2$, then

$$\begin{aligned} \text{RP}(p(\vec{a}, x), p(\vec{u}, x')) &\leq \text{RP}(\vec{a}, \vec{u}) + 2 \cdot \text{RP}(x, x') \\ &= \sum_{i=0}^{2} \text{RP}(a_i, u_i) + 2 \cdot \text{RP}(x, x') \\ &= 3r + 2q \end{aligned} \tag{3.5}$$

The term $2q$ reflects that Horner2 is 2-sensitive in the variable x. The fact that we take the sum of the RP distances over the a_i's follows from the metric on the function type (Section 3.4.4). Since **NumFuzz** supports currying (see Theorem 3), the metric is the same as for the multiplicative (tensor) product.

The interaction between the sensitivity of the function under its ideal semantics and the local rounding error incurred in the body of the function Horner2 over exact inputs is illustrated by the function Horner2_with_error, which takes arguments that have rounding error:

```
// Horner's scheme for a second order polynomial with input error
function Horner2_with_error
  (a0: M[u]num, a1: M[u]num, a2: M[u]num, x: ![2](M[u]num)) {
  let [x1] = x; // comonadic sequencing
  let x' = x1; // monadic sequencing needed for FMA
  let a0' = a0;
  let a1' = a1;
  let a2' = a2;
  Horner2 a0' a1' a2' x'
}
```

We can soundly type Horner2_with_error in **NumFuzz** with the following signature:

$$\text{Horner2} : \text{M[u]num} \multimap \text{M[u]num} \multimap \text{M[u]num} \multimap \text{![2](M[u]num)} \multimap \text{M[7u]num},$$

From the type, we see that **NumFuzz** guarantees that Horner2_with_error produces at most 7u rounding error: from Eq. (3.5) it follows that the sensitivity of the function contributes 5u, and rounding error incurred by evaluating Horner2 over exact inputs contributes the remaining 2u.

NumFuzz is a higher-order language, and it is possible to implement Horner's scheme for polynomials of fixed degree n by first writing a n-ary monadic fold function—a higher-order function—and then applying it to a product of n coefficients a_i. The type system of **NumFuzz** is capable of expressing the fold function along with its roundoff error. For example, fold3 is a 2-ary monadic fold function:

```
// A fold-like function over lists of length 3
function fold3
  (a : num ⊗ num ⊗ num, g : ![2](num ⊸ num ⊸ M[u]num)) {
  let [g'] = g;
  let (a0, b) = a;
  let (a1, a2)= b;
  s = g' a2 a1;
  let z = s;
  g' z a0
}
```

The type of fold3 is:

$$\text{fold3} : \text{num} \otimes \text{num} \otimes \text{num} \multimap \text{M[u]num} \multimap ![2](\text{num} \multimap \text{num} \multimap \text{M[u]num}) \multimap \text{M[2u]num}.$$

We can use this fold-like function to implement Horner2 like so:

```
// Horner's scheme for a second order polynomial using a fold-like function
function Horner2
  (a: num ⊗ num ⊗ num, x: ![2]num) {
  let [x1] = x; // comonadic sequencing
  g = fun (a: num) {fun (b: num) {FMA a x1 b}};
  fold3 a [g{2}]
}
```

We will see why the annotation {2} is required in Section 3.7.

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

program behaves on two different inputs. In **NumFuzz**, the rounding error of a program with a case analysis is then a measure of the maximum rounding error that occurs in any single branch.

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

$$\begin{aligned}
&\text{case1} : \ !_{\infty}\mathbb{R} \multimap M_u\mathbb{R} \\
&\text{case1}\ x \triangleq\ \textbf{let}\ [c] = \text{is_pos}\ \textbf{in}\ x \\
&\qquad\qquad\quad \text{if}\ c\ \text{then mulfp}(x, x)\ \text{else}\ \textbf{ret}\ 0.
\end{aligned}$$

From the signature of case1, we see that the relative precision (RP) is unit roundoff, due to the single rounding in mulfp (x, x).

3.7 Implementation and Evaluation

3.7.1 Implementation

We have developed a prototype type checker for **NumFuzz** in OCaml, based on the sensitivity-inference algorithm due to de Amorim et al. (2014) developed for a dependently-typed extension of *Fuzz* (Gaboardi et al., 2013). Given an environment Γ, a term e, and a type σ, the goal of type checking is to determine if a derivation $\Gamma \vdash e : \sigma$ exists. For sensitivity type systems, type checking and type inference can be achieved by solving the sensitivity inference problem. The sensitivity inference problem is defined using *context skeletons* $\Gamma^{\bullet}$ which are partial maps from variables to

NumFuzz types. If we denote by $\overline{\Gamma}$ the context Γ with all sensitivity assignments removed, then the sensitivity inference problem is defined (de Amorim et al., 2014, Definition 5) as follows.

Definition 26 (Sensitivity Inference)**.** Given a skeleton $\Gamma^\bullet$ and a term e, the *sensitivity inference problem* computes an environment Γ and a type σ with a derivation $\Gamma \vdash e : \sigma$ such that $\Gamma^\bullet = \overline{\Gamma}$.

We solve the sensitivity inference problem using the algorithm given in Figure 3.5.

Given a term e and a skeleton environment $\Gamma^\bullet$, the algorithm produces an environment $\Gamma^\bullet$ with sensitivity information and a type σ. Calls to the algorithm are written as $\Gamma^\bullet; e \Rightarrow \Delta; \sigma$.

Every step of the algorithm corresponds to a derivation in **NumFuzz**. The syntax of the algorithmic rules differs from the syntax of **NumFuzz** (Figure 3.2) in two places: the argument of lambda terms require type annotations $(x : \sigma)$, and the box constructor requires a sensitivity annotation $([v\{s\}])$. The algorithmic rules for these constructs are as follows:

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

Following de Amorim et al. (2014) the algorithm uses a *bottom-up* rather than a *top-down* approach. In the *top-down* approach, given a term e, type σ, and environment Γ, the environment is split and used recursively to type the subterms of the expression e. The *bottom-up* approach avoids splitting the environment Γ by calculating the minimal sensitivities and roundoff errors required to type each subexpression. The sensitivities and errors of each subexpression are then combined and compared to Γ and σ using subtyping. The subtyping relation in **NumFuzz** is defined in Figure 3.7 and captures the fact that a k-sensitive function is also k'-sensitive for $k \leq k'$. Importantly, subtyping is admissible in **NumFuzz**.

Theorem 4. The typing judgment $\Gamma \vdash e : \tau'$ is derivable given a derivation $\Gamma \vdash e : \tau$ and a type τ' such that $\tau \sqsubseteq \tau'$.

$$\text{(Var)}\ \frac{}{\Gamma^{\bullet}, x : \sigma; x \Rightarrow \Gamma^{0}, x :_{1} \sigma; \sigma} \qquad \text{(Const)}\ \frac{}{\Gamma^{\bullet}; k \in R \Rightarrow \Gamma^{0}; \text{num}}$$

$$\text{(Unit)}\ \frac{}{\Gamma^{\bullet}; e \in \text{unit} \Rightarrow \Gamma^{0}; \text{unit}} \qquad \text{(Let)}\ \frac{\Gamma^{\bullet}; e \Rightarrow \Gamma; \tau \quad \Gamma^{\bullet}, x; f \Rightarrow \Theta, x :_{s} \tau; \sigma \quad s > 0}{\Gamma^{\bullet}; \textbf{let } x = e \textbf{ in } f \Rightarrow s * \Gamma + \Theta; \sigma}$$

$$(\multimap \text{I})\ \frac{\Gamma^{\bullet}, x : \sigma; e \Rightarrow \Gamma, x :_{s} \sigma; \tau \quad s \geq 1}{\Gamma^{\bullet}; \lambda(x : \sigma).e \Rightarrow \Gamma; \sigma \multimap \tau} \qquad (\multimap \text{E})\ \frac{\Gamma^{\bullet}; v \Rightarrow \Gamma; \sigma \multimap \tau \quad \Gamma^{\bullet}; w \Rightarrow \Delta; \sigma' \quad \sigma' \sqsubseteq \sigma}{\Gamma^{\bullet}; vw \Rightarrow \Gamma + \Delta; \tau}$$

$$(\ \&\ \text{I})\ \frac{\Gamma^{\bullet}; v \Rightarrow \Gamma_1; \sigma \quad \Gamma^{\bullet}; w \Rightarrow \Gamma_2; \tau}{\Gamma^{\bullet}; \langle v, w \rangle \Rightarrow \max(\Gamma_1, \Gamma_2); \sigma \ \&\ \tau} \qquad (\ \&\ \text{E})\ \frac{\Gamma^{\bullet}; v \Rightarrow \Gamma; \tau_1 \ \&\ \tau_2}{\Gamma^{\bullet}; \pi_i\ v \Rightarrow \Gamma; \tau_i}$$

$$(\otimes \text{I})\ \frac{\Gamma^{\bullet}; v \Rightarrow \Gamma_1; \sigma \quad \Gamma^{\bullet}; w \Rightarrow \Gamma_2; \tau}{\Gamma; (v, w) \Rightarrow \Gamma_1 + \Gamma_2; \sigma \otimes \tau} \qquad (\otimes \text{E})\ \frac{\Gamma^{\bullet}; v \Rightarrow \Delta; \sigma \otimes \tau \quad \Gamma^{\bullet}, x : \sigma, y : \tau; e \Rightarrow \Gamma, x :_{s_1} \sigma, y :_{s_2} \tau; \rho}{\Gamma^{\bullet}; \textbf{let } (x, y) = v \textbf{ in } e \Rightarrow \Gamma + \max(s_1, s_2) * \Delta; \rho}$$

$$(+\text{I})\ \frac{\Gamma^{\bullet}; v \Rightarrow \Gamma; \sigma}{\Gamma^{\bullet}; \textbf{inl}\ \ v \Rightarrow \Gamma; \sigma + \tau} \qquad (+\text{E})\ \frac{\Gamma^{\bullet}, x; e \Rightarrow \Theta, x :_{s} \sigma; \rho_1 \quad \Gamma^{\bullet}, y; f \Rightarrow \Theta, y :_{s} \tau; \rho_2 \quad \Gamma^{\bullet}; v \Rightarrow \Gamma; \sigma + \tau}{\Gamma^{\bullet}; \text{case } v \text{ of } (\textbf{inl } x.e \mid \textbf{inr } y.f) \Rightarrow \bar{s} * \Gamma + \Theta; \max(\rho_1, \rho_2)}$$

$$(!\ \text{I})\ \frac{\Gamma^{\bullet}; v \Rightarrow \Gamma; \sigma}{\Gamma^{\bullet}; [v\{s\}] \Rightarrow s * \Gamma; !_{s}\sigma} \qquad (!\ \text{E})\ \frac{\Gamma^{\bullet}; v \Rightarrow \Gamma; !_{s}\sigma \quad \Gamma^{\bullet}; x \Rightarrow \Theta, x :_{t*s} \sigma; e : \tau}{\Gamma^{\bullet}; \textbf{let } [x] = v \textbf{ in } e \Rightarrow t * \Gamma + \Theta; \tau}$$

$$\text{(Ret)}\ \frac{\Gamma^{\bullet}; v \Rightarrow \Gamma; \tau}{\Gamma^{\bullet}; \textbf{ret}v \Rightarrow \Gamma; M_0\tau} \qquad \text{(Rnd)}\ \frac{\Gamma^{\bullet}; v \Rightarrow \Gamma; \text{num}}{\Gamma^{\bullet}; \textbf{rnd}\ v \Rightarrow \Gamma; M_q\text{num}}$$

$$\text{(Op)}\ \frac{\Gamma^{\bullet}; v \Rightarrow \Gamma; \sigma \quad \{\text{op} : \sigma \multimap \text{num}\} \in \Sigma}{\Gamma^{\bullet}; \text{op}(v) \Rightarrow \Gamma; \text{num}} \qquad (M_u\ \text{E})\ \frac{\Gamma^{\bullet}; v \Rightarrow \Gamma; M_r\sigma \quad \Gamma^{\bullet}, x; f \Rightarrow \Theta, x :_{s} \sigma; M_q\tau}{\Gamma^{\bullet}; \textbf{let}_{\mathbf{M}}\ x = v \textbf{ in } f \Rightarrow s * \Gamma + \Theta; M_{s*r+q}\tau}$$

$$\bar{s} \triangleq \begin{cases} s & s > 0 \\ \epsilon & \text{otherwise} \end{cases}$$

Figure 3.5: Algorithmic rules for **NumFuzz**. Γ^0 denotes an environment where all variables have sensitivity zero. The supertype (max) and subtype ($\sqsubseteq$) relations on **NumFuzz** types are given in Figure 3.6 and Figure 3.7, respectively.

$$
\begin{aligned}
\max(\sigma_1 \mathbin{\&} \tau_1, \sigma_2 \mathbin{\&} \tau_2) &\triangleq \max(\sigma_1, \sigma_2) \mathbin{\&} \max(\tau_1, \tau_2) \\
\max(\sigma_1 \otimes \tau_1, \sigma_2 \otimes \tau_2) &\triangleq \max(\sigma_1, \sigma_2) \otimes \max(\tau_1, \tau_2) \\
\max(\sigma_1 + \tau_1, \sigma_2 + \tau_2) &\triangleq \max(\sigma_1, \sigma_2) + \max(\tau_1, \tau_2) \\
\max(M_{s_1}\tau_1, M_{s_2}\tau_2) &\triangleq M_{\max(s_1,s_2)} \max(\tau_1, \tau_2) \\
\max(!_{s_1}\tau_1, !_{s_2}\tau_2) &\triangleq !_{\min(s_1,s_2)} \max(\tau_1, \tau_2) \\
\min(\sigma_1 \mathbin{\&} \tau_1, \sigma_2 \mathbin{\&} \tau_2) &\triangleq \min(\sigma_1, \sigma_2) \mathbin{\&} \min(\tau_1, \tau_2) \\
\min(\sigma_1 \otimes \tau_1, \sigma_2 \otimes \tau_2) &\triangleq \min(\sigma_1, \sigma_2) \otimes \min(\tau_1, \tau_2) \\
\min(\sigma_1 + \tau_1, \sigma_2 + \tau_2) &\triangleq \min(\sigma_1, \sigma_2) + \min(\tau_1, \tau_2) \\
\min(M_{s_1}\tau_1, M_{s_2}\tau_2) &\triangleq M_{\min(s_1,s_2)} \min(\tau_1, \tau_2) \\
\min(!_{s_1}\tau_1, !_{s_2}\tau_2) &\triangleq !_{\max(s_1,s_2)} \min(\tau_1, \tau_2) \\
\max(\sigma_1 \multimap \tau_1, \sigma_2 \multimap \tau_2) &\triangleq \min(\sigma_1, \sigma_2) \multimap \max(\tau_1, \tau_2) \\
\min(\sigma_1 \multimap \tau_1, \sigma_2 \multimap \tau_2) &\triangleq \max(\sigma_1, \sigma_2) \multimap \min(\tau_1, \tau_2)
\end{aligned}
$$

Figure 3.6: The max (supertype) relation on **NumFuzz** types, with $s_1, s_2 \in \mathbb{R}^{\geq 0} \cup \{\infty\}$.

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

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

$$
\frac{}{\sigma \sqsubseteq \sigma} \sqsubseteq\text{-refl}
$$

Figure 3.7: **NumFuzz** subtyping relation, with $s, s', u, u' \in \mathbb{R}^{\geq 0} \cup \{\infty\}$.

Proof. The proof follows by induction on the derivation $\Gamma \vdash e : \tau$. Most cases are immediate, but some require weakening (Lemma 2). We detail two examples here.

Case (Var). We are required to show $\Gamma, x :_s \tau, \Delta \vdash x : \tau'$ given $\tau \sqsubseteq \tau'$. We can conclude by weakening (Lemma 2) and the fact that the subenvironment relation is preserved by subtyping; i.e., $x :_s \tau' \sqsubseteq x :_s \tau$.

Case (! I). We are required to show $s * \Gamma \vdash [v] : !_{s'}\sigma'$ for some $s' \leq s$ and $\sigma' \sqsubseteq \sigma$. By the induction hypothesis we have $\Gamma \vdash v : \sigma'$, and by the box introduction rule (! I) we have $s' * \Gamma \vdash [v] : !_{s'}\sigma'$. Because $s * \Gamma \sqsubseteq s' * \Gamma$ for $s' \leq s$ we can conclude by weakening.

□

The algorithmic rules presented in Figure 3.5 define a *sound* type checking algorithm for **NumFuzz**:

Theorem 5 (Algorithmic Soundness)**.** If $\Gamma^{\bullet}; e \Rightarrow \Gamma; \sigma$ then there exists a derivation $\Gamma \vdash e : \sigma$.

Proof. By induction on the algorithmic derivations, we see that every step of the algorithm corresponds to a derivation in **NumFuzz**. Many cases are immediate, but those that use subtyping, supertyping, or subenvironments are not; we detail those here.

Case ($\multimap$ E). Applying subtyping (Theorem 4) to the induction hypothesis we have $\Delta \vdash w : \sigma$. We conclude by the ($\multimap$ E) rule.

Case (& I). We define the max relation on any two subenvironments Γ_1 and Γ_2 so that $\max(\Gamma_1, \Gamma_2) \sqsubseteq \Gamma_1$ and $\max(\Gamma_1, \Gamma_2) \sqsubseteq \Gamma_2$. Let us denote $\max(\Gamma_1, \Gamma_2)$ by the environment Δ. By the induction hypothesis and weakening (Lemma 2) we have $\Delta \vdash v : \sigma$ and $\Delta \vdash w : \tau$. We conclude by the (& I) rule.

Case ($\otimes$ E). Let us denote by Θ the environment $\Gamma + \max(s_1, s_2) * \Delta$ and by s the sensitivity $\max(s_1, s_2)$. By the induction hypothesis and weakening (Lemma 2), we have $\Gamma, x :_s \sigma, y :_s \tau \vdash e : \rho$ and we can conclude by the ($\otimes$ E) rule.

Case (+ E). The proof relies on the fact that, given a ρ such that $\rho = \max(\rho_1, \rho_2)$, both ρ_1 and ρ_2 are subtypes of ρ. Using this fact, and by subtyping (Theorem 4) and the induction hypothesis, we have $\Theta, x :_s \sigma \vdash e : \rho$ and $\Theta, y :_s \tau \vdash f : \rho$. If $s > 0$, we can can conclude directly by the (+ E) rule. Otherwise, we first apply weakening (Lemma 2) and then conclude by the (+ E) rule.

□

3.7.2 Evaluation

In order to serve as a practical tool, our type checker must infer useful error bounds within a reasonable amount of time. Our empirical evaluation therefore focuses on measuring two key properties: tightness of the inferred error bounds and performance. To this end, our evaluation includes a comparison in terms of relative error and performance to two popular tools that soundly and automatically bound relative error: FPTaylor (Solovyev et al., 2019) and Gappa (Marc Daumas and, 2010). Although Daisy (Eva Darulova and et al., 2018) and Rosa (Darulova and Kuncak, 2017) also compute relative error bounds, they do not compute error bounds for the directed rounding modes, and our instantiation of **NumFuzz** requires round towards $+\infty$ (see Section 3.6). For our comparison to Gappa and FPTaylor, we use benchmarks from FPBench (Damouche et al., 2017), which is the standard set of benchmarks used in the domain; we also include the Horner scheme discussed in Section 3.6. There are limitations, summarized below, to the arithmetic operations that the instantiation of **NumFuzz** used in our type checker can handle, so we are only able to evaluate a subset of the FPBench benchmarks. Even so, larger examples with more than 50 floating-point operations are intractable for most tools (Das et al., 2020), including FPTaylor and Gappa, and are not part of FPBench. Our evaluation therefore includes larger examples with well-known relative error bounds that we compare against. Finally, we used our type checker to analyze the rounding error of four floating-point conditionals.

Our experiments were performed on a MacBook with a 1.4 GHz processor and 8 GB of memory. Relative error bounds are derived from the relative precision computed by **NumFuzz** using Equation (2.11).

Limitations of NumFuzz

Soundness of the error bounds inferred by our type checker is guaranteed by Corollary 1 and the instantiation of **NumFuzz** described in Section 3.6. This instantiation imposes the following

Table 3.1: Comparison of **NumFuzz** to FPTaylor and Gappa. The Bound column gives upper bounds on relative error (smaller is better); the bounds for FPTaylor and Gappa assume all variables are in $[0.1, 1000]$. The Ratio column gives the ratio of **NumFuzz**'s relative error bound to the tightest (best) bound of the other two tools; values less than 1 indicate that **NumFuzz** provides a tighter bound. The Ops column gives the number of operations in each benchmark. Benchmarks from FPBench are marked with a (*).

Benchmark	Ops	Bound			Ratio	Timing (s)		
		NumFuzz	FPTaylor	Gappa		**NumFuzz**	FPTaylor	Gappa
hypot*	4	5.55e-16	5.17e-16	**4.46e-16**	1.3	0.002	3.55	0.069
x_by_xy*	3	4.44e-16	fail	**2.22e-16**	2	0.002	-	0.034
one_by_sqrtxx	3	5.55e-16	5.09e-13	**3.33e-16**	1.7	0.002	3.34	0.047
sqrt_add*	5	9.99e-16	6.66e-16	**5.54e-16**	1.5	0.003	3.28	0.092
test02_sum8*	8	1.55e-15	9.32e-14	1.55e-15	1	0.002	14.61	0.244
nonlin1*	2	4.44e-16	4.49e-16	**2.22e-16**	2	0.003	3.24	0.040
test05_nonlin1*	2	4.44e-16	4.46e-16	**2.22e-16**	2	0.008	3.27	0.042
verhulst*	4	8.88e-16	7.38e-16	**4.44e-16**	2	0.002	3.25	0.069
predatorPrey*	7	1.55e-15	4.21e-11	**8.88e-16**	1.7	0.002	3.28	0.114
test06_sums4_sum1*	4	6.66e-16	6.71e-16	6.66e-16	1	0.003	3.84	0.069
test06_sums4_sum2*	4	6.66e-16	1.78e-14	**4.44e-16**	1.5	0.002	11.02	0.055
i4*	4	4.44e-16	4.50e-16	4.44e-16	1	0.002	3.30	0.055
Horner2	4	4.44e-16	6.49e-11	4.44e-16	1	0.002	11.72	0.052
Horner2_with_error	4	1.55e-15	1.61e-10	**1.11e-15**	1.4	0.002	19.56	0.119
Horner5	10	1.11e-15	1.62e-01	1.11e-15	1	0.003	22.08	0.209
Horner10	20	2.22e-15	1.14e+13	2.22e-15	1	0.003	40.68	0.650
Horner20	40	4.44e-15	2.53e+43	4.44e-15	1	0.003	109.42	2.246

limitations on the benchmarks we can consider in our evaluation. First, only the operations $+$, $*$, $/$, and sqrt are supported by our instantiation, so we can't use benchmarks with subtraction or transcendental functions. Second, all constants and variables must be strictly positive numbers, and the rounding mode must be fixed as round towards $+\infty$. These limitations follow from the fact that the RP metric (Definition 3) is only well-defined for non-zero values of the same sign. We leave the exploration of tradeoffs between the choice of metric and the primitive operations that can be supported by the language to future work. Given these limitations, along with the fact that **NumFuzz** does not currently support programs with loops, we were able to include 13 of the 129 unique (at the time of writing) benchmarks from FPBench in our evaluation.

Small Benchmarks

The results for benchmarks with fewer than 50 floating-point operations are given in Table 3.1. Eleven of the seventeen benchmarks are taken from the FPBench benchmarks. Both FPTaylor and Gappa require user provided interval bounds on the input variables in order to compute the relative error; we used an interval of $[0.1, 1000]$ for each of the benchmarks. We used the default configuration for FPTaylor, and used Gappa without providing hints for interval subdivision. The floating-point format of each benchmark is binary64, and the rounding mode is set at round towards $+\infty$; the unit roundoff in this setting is 2^{-52} (approximately 2.22e-16). Only Horner2_with_error assumes error in the inputs.

Large Benchmarks

Table 3.2 shows the results for benchmarks with 100 or more floating-point operations. Five of the nine benchmarks are taken from SATIRE (Das et al., 2020), an *empirically sound* static analysis tool that computes absolute error bounds. Although SATIRE does not statically compute relative error bounds for the benchmarks listed in Table 3.2, most of these benchmarks have well-known worst case relative error bounds that we can compare against. These bounds are given in the Std. column in Table 3.2; the relevant references are as follows: Horner's scheme (cf. Higham, 2002, p. 95), summation (cf. Boldo et al., 2023, p.260), and matrix multiply (cf. Higham, 2002, p.63). For matrix multiplication, we report the max elementwise relative error bound produced by **NUMFUZZ**. When available, the Timing column in Table 3.2 lists the time reported for SATIRE to compute *absolute* error bounds (cf. Das et al., 2020, Table III).

Table 3.2: The performance of **NumFuzz** on benchmarks with 100 or more floating-point operations. The Std. column gives relative error bounds from the literature. Benchmarks from Satire are marked with with an (a); the Satire subcolumn gives timings for the computation of *absolute* error bounds as reported in (Das et al., 2020).

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

Table 3.3: The performance of **NumFuzz** on conditional benchmarks. Benchmarks from FPBench are marked with a (∗). Benchmarks from Dahlquist and Björck (cf. Dahlquist and Björck, 2008, p. 119) are marked with with a (b).

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

Conditional Benchmarks

Table 3.3 shows the results for conditional benchmarks. Two of the four benchmarks are taken from FPBench and the remaining benchmarks are examples from Dahlquist and Björck (cf. Dahlquist and Björck, 2008, p. 119). We were unable to compare the performance and computed relative error bounds shown in Table 3.3 against other tools. While Daisy, FPTaylor, and Gappa compute relative error bounds, they don't handle conditionals. And, while PRECiSA can handle conditionals, it doesn't compute relative error bounds. Only Rosa computes relative error bounds for floating-point conditionals, but Rosa doesn't compute bounds for the directed rounding modes.

Evaluation Summary

We draw three main conclusions from our evaluation.

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

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

Roundoff error bounds derived via type checking are strong. The relative error bounds produced by **NumFuzz** hold for all positive real inputs, assuming the absence of overflow and underflow. In comparison, the relative error bounds derived by FPTaylor and Gappa only hold for the user provided interval bounds on the input variables, which we took to be $[0.1, 1000]$. Increasing this interval range allows FPTaylor and Gappa to give stronger bounds, but can also lead to slower analysis. Furthermore, given that relative error is poorly behaved for values near zero, some tools are sensitive to the choice of interval. We see this in the results for the benchmark x_by_xy in Table 3.1, where we are tasked with calculating the roundoff error produced by the expression $x/(x + y)$, where x and y are binary64 floating-point numbers in the interval $[0.1, 1000]$. For these

parameters, the expression lies in the interval $[5.0\text{e-}05, 1.0]$ and the relative error should still be well defined. However, FPTaylor (used with its default configuration) fails to provide a bound, and issues a warning due to the value of the expression being too close to zero.

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

3.8 Related Work

Abstract Interpretation

The theory of abstract interpretation offers a generic framework for designing sound static analysis tools. At the heart of any abstract interpretation framework is the concept of an *abstract domain*, which provides a mathematical approximation of the program properties being analyzed. For floating-point programs, abstract interpretation frameworks use numerical abstract domains to soundly overapproximate the set of values that program variables can represent. Common numerical abstract domains for analyzing floating-point programs include interval arithmetic (Moore et al., 2009), affine arithmetic (de Figueiredo and Stolfi, 2004), and convex polyhedra (Chen et al., 2008).

Many tools based on abstract interpretation aim to derive sound and accurate bounds for floating-point variables but do not compute bounds or estimates on the rounding error in a floating-point result. These tools make it possible to validate numerical behaviors of systems without precisely tracking the rounding error associated with each floating-point operation, and are described in works by Rivera et al. (2024), Miné (2004), Chen et al. (2008, 2009, 2010), Jeannet and Miné (2009),

Chapoutot (2010), and Chapoutot and Martel (2009). In a somewhat orthogonal research direction, abstract interpretation frameworks have also been used in the development of satisfiability decision procedures for constraints over floating-point arithmetic (Haller et al., 2012).

Tools that use abstract domains to derive sound rounding error bounds include Gappa (Daumas and Melquiond, 2010), Rosa (Darulova and Kuncak, 2017), Daisy (Darulova et al., 2018), PRECiSA (Titolo et al., 2018, 2024), Fluctuat (Goubault and Putot, 2011), and Astrée (Cousot et al., 2005).

While abstract interpretation is flexible and can be generally applied to programs with conditionals and loops, it can significantly overestimate rounding error, and it is difficult to model the cancellation of errors. Unlike abstract interpretation, type-based approaches like **NumFuzz** provide a mechanism for defining valid programs, and can support features like foreign function interfaces (Ghica and Smith, 2014).

Type Systems for Representation Error

A type system due to Martel (2018) uses dependent types to track the occurrence and propagation of *representation errors*; i.e., error due to the fact that floating-point numbers do not exactly represent real numbers. In **NumFuzz**, both representation error and *roundoff error*—the error due to rounding the results of operations—are accounted for by the type system. A significant difference between **NumFuzz** and the type system proposed by Martel is error soundness. In Martel's system, the soundness result says that a semantic relation capturing the notion of accuracy between a floating-point expression and its ideal counterpart is preserved by a reduction relation. This is weaker than a standard type soundness guarantee. In particular, it is not shown that well-typed terms satisfy the semantic relation. In **NumFuzz**, the central novel property guaranteed by our type system is much stronger: well-typed programs of monadic type satisfy the error bound indicated by their type.

Optimization Techniques for Program Analysis

To provide more precise bounds, many methods rely on optimization. Conceptually, these methods bound the roundoff error by representing the error symbolically as a function of the program inputs and the error variables introduced during the computation, and then perform global optimization over all settings of the errors (Truong et al., 2014). Since the error expressions are typically complex, verification methods use approximations to simplify the error expression to make optimization more tractable, and mostly focus on straight-line programs. For instance, Real2Float (Magron et al., 2017) separates the error expression into a linear term and a non-linear term; the linear term is bounded using semidefinite programming, while the non-linear term is bounded using interval arithmetic. FPTaylor (Solovyev et al., 2019) was the first tool to use Taylor approximations of error expressions. Abbasi and Darulova (2023) describe a modular method for bounding the propagation of errors using Taylor approximations, and Rosa (Darulova and Kuncak, 2014, 2017) uses Taylor series to approximate the propagation of errors in possibly non-linear terms.

In contrast, our type system does not rely on global optimization and can be instantiated to different models of floating-point arithmetic with minimal changes. Our language supports a variety of datatypes and higher-order functions. While our language does not support recursive types and general recursion, similar languages support these features (Reed and Pierce, 2010; Azevedo de Amorim et al., 2017; Dal Lago and Gavazzo, 2022b) and we expect they should be possible in **NumFuzz**; however, the precision of the error bounds for programs using general recursion might be poor. Another limitation of our method is in typing conditionals: while **NumFuzz** can only derive error bounds when the ideal and floating-point executions follow the same branch, tools that use general-purpose solvers (e.g., PRECiSA and Rosa) can produce error bounds for programs where the ideal and floating-point executions take different branches.

Verification and Synthesis

Formal verification has a long history in the area of numerical computations, starting with the pioneering work of Harrison (Harrison, 1999, 1997a, 2000). Formalized specifications of floating-point arithmetic have been developed in the Coq (Boldo and Melquiond, 2011), Isabelle (Yu, 2013), and PVS (Paul S., 1995) proof assistants. These specifications have been used to develop sound tools for floating-point error analysis that generate proof certificates, such as VCFloat (Ramananandro et al., 2016; Appel and Kellison, 2024) and PRECiSA (Titolo et al., 2018, 2024). They have also been used to mechanize proofs of error bounds for specific numerical programs (e.g., (Kellison and Appel, 2022; Sylvie Boldo and et al., 2014; Tekriwal et al., 2023; Kellison et al., 2023; Moscato et al., 2019)). Work by Patrick Cousot and et al. (2005) has applied abstract interpretation to verify the absence of floating-point faults in flight-control software, which have caused real-world accidents. Finally, recent work uses *program synthesis*: Herbie (Panchekha et al., 2015) automatically rewrites numerical programs to reduce numerical error, while RLibm (Jay P. Lim and, 2022) automatically generates correctly-rounded math libraries.

Sensitivity Type Systems

NumFuzz belongs to a line of work on linear type systems for sensitivity analysis, starting with *Fuzz* (Reed and Pierce, 2010). We point out a few especially relevant works. Our syntax and typing rules are inspired by Dal Lago and Gavazzo (2022b), who propose a family of *Fuzz*-like languages and define various notions of operational equivalence; we are inspired by their syntax, but our case elimination rule (+ E) is different: we require s to be strictly positive when scaling the conclusion. This change is due to a subtle difference in how sums are treated.

In **NumFuzz**, as in *Fuzz*, the distance between left and right injections is ∞, whereas in the system by Dal Lago and Gavazzo (2022b), left and right injections are not related at any distance. Our approach allows non-trivial operations returning booleans to be typed as infinite sensitivity

functions, but the case rule must be adjusted: to preserve soundness, the conclusion must retain a dependence on the guard expression, even if the guard is not used in the branches.

de Amorim et al. (2021) added a graded monadic type to *Fuzz* to handle more complex variants of differential privacy; in their application, the grade does not interact with the sensitivity language. Finally, recent work by Wunder et al. (2023) proposes a variant of *Fuzz* with "bunched" (tree-shaped) contexts, with two methods of combining contexts. It could be interesting to develop a bunched version of **NumFuzz**—the metrics for $\otimes$ and $\&$ could be naturally accommodated in the contexts, possibly leading to more precise error analysis.

Other Approaches

The numerical analysis literature has explored other conceptual tools for static error analysis, such as stochastic error analysis (Connolly et al., 2021). Techniques for *dynamic* error analysis, which estimate the rounding error at runtime, have also been proposed (Higham, 2002).

It would be interesting to consider these techniques from a formal methods perspective, whether by connecting dynamic error analysis with ideas like runtime verification, or developing methods to verify the correctness of dynamic error analysis.

3.9 Conclusion

NumFuzz is a functional programming language designed to express quantitative bounds on forward rounding error. The rounding error analysis modeled by **NumFuzz** is standard: a sensitivity analysis is combined with a local rounding error analysis to derive a global rounding error bound. **NumFuzz** uses a linear type system and a graded comonad to perform a sensitivity analysis, and uses a novel a graded monad to track rounding error. A major benefit of our type-based approach is *soundness*:

NumFuzz programs are guaranteed to satisfy the error bounds assigned to them by the type system, which are overapproximations of the true rounding error. Another advantage is scalability: **NumFuzz** can infer tight error bounds for significantly larger programs than existing static analysis tools in a reasonable amount of time. Moreover, on well known benchmarks, **NumFuzz** infers error bounds that are competitive with those produced by existing tools, often with significantly faster performance. **NumFuzz** can be extended in various ways, and we conclude this chapter with a discussion of promising directions for future development.

Additional Language Features

Rounding error bounds are typically parametric in the length of the input data. For instance, the error bound for Horner's scheme (cf. Section 3.6) is usually expressed in terms of the polynomial's degree, which corresponds to the length of the vector of polynomial coefficients. Currently, **NumFuzz**, like *Fuzz*, only supports numeric annotations (grades), but these are insufficient for expressing how error bounds depend on properties of input data. To address this limitation, Gaboardi et al. (2013) introduced lightweight dependent types—sensitivity variables and quantifiers over these variables—to the types of *Fuzz*, enabling the expression of more general sensitivity properties. Given this prior work, we expect **NumFuzz**'s type system could similarly be extended to support lightweight dependent types.

Combining this extension with a bounded loop construct would further enhance **NumFuzz** by enabling the expression of more general error bounds and reducing the verbosity of programs. Compared to a bounded loop construct, it is less obvious that extending **NumFuzz** to support general recursive functions and types, even those with precise sensitivity as described by Azevedo de Amorim et al. (2017) for *Fuzz*, would be immediately useful for potential users. (Although it is clear that it would complicate the metatheory).

Probabilistic Rounding

Probabilistic models of rounding errors have been proposed for both deterministic computations (Higham and Mary, 2019; Ipsen and Zhou, 2020) and probabilistic computations (Constantinides et al., 2021). **NumFuzz** could be extended to track probabilistic rounding errors by incorporating techniques from probabilistic languages, such as those described by de Amorim et al. (2021) or Crubille and Dal Lago (2015). While Kahan (1996) and Ipsen and Zhou (2020) provide critical assessments of probabilistic rounding error analyses, Constantinides et al. (2021) argue that the probabilistic approach is necessary for analyzing rounding error in probabilistic computations.

Mixed-Precision Computations

It is possible to represent mixed-precision computations in **NumFuzz** by adding additional **rnd** constructs to the language, with each construct corresponding to a different supported precision. The challenge lies in accurately modeling the expected behavior when composing rounding operations. According to the IEEE standard (IEEE Computer Society, 2019), conversions between formats with the same radix but wider precision should always be exact. Evaluating the expression $\mathbf{let}_{\mathbf{M}}\ x = (\mathbf{rnd32}\ 3.0)\ \mathbf{in}\ (\mathbf{rnd64}\ x)$, where **rnd32** rounds to binary32 and **rnd64** rounds to binary64, should therefore produce only a single rounding effect, due to the evaluation of (**rnd32** 3.0). However, under the current monadic sequencing (MLet) rule, each operation introduces its own error, effectively modeling a scenario where both rounding steps contribute to the total error. While this is a sound overapproximation, it raises the question of whether **NumFuzz** can support a more precise, fine-grained analysis that distinguishes between scenarios where no additional error is introduced. One possible approach to achieving this finer-grained analysis is to use *graded monad transformers*, as described by Ivaskovic (2023), for combining two analyses of computations based on the same type of effectful operation.

Additional Error Measures

A natural follow up to our work on **NumFuzz** is to consider whether or not other error measures from the literature can be used in place of relative precision (Definition 3). Error measures that can uniformly represent floating-point error on both large and small values are the *units in the last place* (ULP) error, which measures the number of floating-point values between an approximate and exact value, and its logarithm, *bits of error* (Damouche et al., 2017):

$$er_{\text{ULP}}(x, \tilde{x}) = |\mathbb{F} \cap [\min(x, \tilde{x}), \max(x, \tilde{x})]| \quad \text{and} \quad er_{bits}(x, \tilde{x}) = \log_2 er_{\text{ULP}}(x, \tilde{x}). \tag{3.6}$$

While static analysis tools that provide sound, worst-case error bounds for floating-point programs compute relative or absolute error bounds (or both), the ULP error and its logarithm are often used in tools that optimize either the performance or accuracy of floating-point programs, like Herbie (Panchekha et al., 2015) and Stoke (Schkufza et al., 2014).

Generalizations of the relative precision metric have been proposed by Ziv (1982) and Pryce (1985, 1984), for analyzing the error of programs involving vectors and matrices. It would be interesting to explore whether **NumFuzz** could accommodate these metrics, though this would naturally require extending the type system to support types for matrices and vectors; *Fuzz*-like languages with matrix types have been described by Near et al. (2019) and Wunder et al. (2023).

Mechanization

It would be possible to mechanize several results about **NumFuzz** in a proof assistant like Coq. For instance, formalizing key aspects of the operational semantics, such as type-preservation and strong normalization (Theorem 2), as well as soundness of the type checking algorithm (Theorem 5), would be straightforward but valuable exercises. To our knowledge, there is currently no mechanization of sensitivity type systems like **NumFuzz** in a proof assistant, making this an interesting area for further exploration.

CHAPTER 4

A LANGUAGE FOR BACKWARD ERROR ANALYSIS

This chapter presents **Bean**, a programming language for **B**ackward **E**rror **AN**alysis. **Bean** features a type system that tracks how backward error flows through programs, and ensures that well-typed programs have bounded backward error.

4.1 Introduction

With **Bean**, our point of departure from other static analysis tools for floating-point rounding error analysis is our focus on deriving backward error bounds rather than forward error bounds. To facilitate our description of backward error, we will use the following notation: floating-point approximations to real-valued functions, as well as data with perturbations due to floating-point rounding error, will be denoted by a tilde. For instance, the floating-point approximation of a real-valued function f will be denoted by $\tilde{f}$, and data that are intended to represent slight perturbations of x will be denoted by $\tilde{x}$.

Backward Error and Backward Stability

Given a floating-point result $\tilde{y}$ approximating $y = f(x)$ with $f : \mathbb{R}^n \to \mathbb{R}^m$, a forward error analysis directly measures the accuracy of the floating-point result by bounding the distance between $\tilde{y}$ and y. In contrast, a backward error analysis identifies an input $\tilde{x}$ that would yield the floating-point result when provided as input to f; i.e., such that $\tilde{y} = f(\tilde{x})$. The backward error is a measure of the distance between the input x and the input $\tilde{x}$.

An illustration of the backward error is given in Figure 4.1. If the backward error is small for every possible input, then an implementation is said to be *backward stable*:

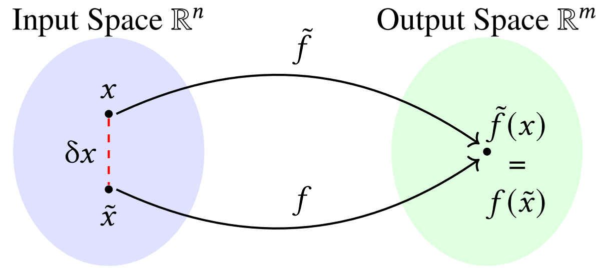


Figure 4.1: An illustration of backward error. The function $\tilde{f}$ represents a floating-point implementation of the function f. Given the points $\tilde{x} \in \mathbb{R}^n$ and $x \in \mathbb{F}^n \subset \mathbb{R}^n$ such that $\tilde{f}(x) = f(\tilde{x})$, the backward error is the distance δx between x and $\tilde{x}$.

Definition 27. (Backward Stability) A floating-point implementation $\tilde{f} : \mathbb{F}^n \longrightarrow \mathbb{F}^m$ of a real-valued function $f : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ is *backward stable* if, for every input $x \in \mathbb{F}^n$, there exists an input $\tilde{x} \in \mathbb{R}^n$ such that

$$f(\tilde{x}) = \tilde{f}(x) \text{ and } d(x, \tilde{x}) \leq \alpha u \tag{4.1}$$

where u is the *unit roundoff*—a value that depends on the precision of the floating-point format $\mathbb{F}$, α is a small constant, and $d : \mathbb{R}^n \times \mathbb{R}^n \longrightarrow \mathbb{R} \cup \{+\infty\}$ provides a measure of distance in $\mathbb{R}^n$.

In general, a large forward error can have two causes: the conditioning of the problem being solved or the stability of the program used to solve it. If the problem being solved is *ill-conditioned*, then it is highly sensitive to floating-point rounding errors, and can amplify these errors to produce arbitrarily large changes in the result. Conversely, if the problem is well-conditioned but the program is *unstable*, then inaccuracies in the result can be attributed to the way rounding errors accumulate during the computation. While forward error alone does not distinguish between these two sources of error, backward error provides a controlled way to separate them. The relationship between forward error and backward error is governed by the *condition number*, which provides a quantitative measure of the conditioning of a problem:

$$\text{forward error} \leq \text{condition number} \times \text{backward error}. \tag{4.2}$$

A more precise definition of the condition number is given in Definition 33.

By automatically deriving sound backward error bounds that indicate the backward stability of programs, **Bean** addresses a significant gap in current tools for automated error analysis. To quote Dianne P. O'Leary (O'Leary, 2009): "Life may toss us some ill-conditioned problems, but there is no good reason to settle for an unstable algorithm."

Backward Error Analysis by Example

A motivating example illustrating the importance of backward error is the dot product of two vectors. While the dot product can be computed in a backward stable way, if the vectors are orthogonal (i.e., when the dot product is zero) the floating-point result can have arbitrarily large relative forward error. This means that, for certain inputs, a forward error analysis can only provide trivial bounds on the accuracy of a floating-point dot product. In contrast, a backward error analysis can provide non-trivial bounds describing the quality of an implementation for all possible inputs.

To see how a backward error analysis works in practice, suppose we are given the vectors $x = (x_0, x_1) \in \mathbb{R}^2$ and $y = (y_0, y_1) \in \mathbb{R}^2$ with floating-point entries. The exact dot product simply computes the sum $s = x_0 \cdot y_0 + x_1 \cdot y_1$, while the floating-point dot product computes $\tilde{s} = (x_0 \odot_{\mathbb{F}} y_0) \oplus_{\mathbb{F}} (x_1 \odot_{\mathbb{F}} y_1)$, where $\oplus_{\mathbb{F}}$ and $\odot_{\mathbb{F}}$ represent floating-point addition and multiplication, respectively. A backward error bound for the computed result $\tilde{s}$ can be derived based on bounds for addition and multiplication. Following the error analysis proposed by Olver (Olver, 1978), and assuming no overflow and underflow, floating-point addition and multiplication behave like their exact arithmetic counterparts, with each input subject to small perturbations. Specifically, for any $a_1, a_2, b_1, b_2 \in \mathbb{R}$, we have:

$$a_1 \oplus_{\mathbb{F}} a_2 = a_1 e^{\delta} + a_2 e^{\delta} \tag{4.3}$$

$$= \tilde{a}_1 + \tilde{a}_2 \tag{4.4}$$

$$b_1 \odot_{\mathbb{F}} b_2 = b_1 e^{\delta'/2} \cdot b_2 e^{\delta'/2} \tag{4.5}$$

$$= \tilde{b}_1 \cdot \tilde{b}_2 \tag{4.6}$$

with $|\delta|, |\delta'| \leq u/(1-u)$, where u is the unit roundoff. For convenience, we use the notation $\varepsilon = u/(1-u)$. The basic intuition behind a perturbed input like $\tilde{a}_1 = a_1 e^{\delta}$ in Equation (4.4) is that $\tilde{a}$ is approximately equal to $a_1 + a_1\delta$ when the magnitude of δ is extremely small.

We can use Equation (4.4) and Equation (4.6) to perform a backward error analysis for the dot product: we can define the vectors $\tilde{x} = (\tilde{x}_0, \tilde{x}_1)$ and $\tilde{y} = (\tilde{y}_0, \tilde{y}_1)$ such that their dot product computed in exact arithmetic is equal to the floating-point result $\tilde{s}$:

$$\begin{aligned}
\tilde{s} = (x_0 \odot_{\mathbb{F}} y_0) \oplus_{\mathbb{F}} (x_1 \odot_{\mathbb{F}} y_1) &= (x_0 e^{\delta_0/2} \cdot y_0 e^{\delta_0/2}) \oplus_{\mathbb{F}} (x_1 e^{\delta_1/2} \cdot y_1 e^{\delta_1/2}) \\
&= (x_0 e^{\delta_0/2} \cdot y_0 e^{\delta_0/2}) e^{\delta_2} + (x_1 e^{\delta_1/2} \cdot y_1 e^{\delta_1/2}) e^{\delta_2} \\
&= (x_0 e^{\delta} \cdot y_0 e^{\delta}) + (x_1 e^{\delta'} \cdot y_1 e^{\delta'}) \\
&= (\tilde{x}_0 \cdot \tilde{y}_0) + (\tilde{x}_1 \cdot \tilde{y}_1) \qquad (4.7)
\end{aligned}$$

where $|\delta|, |\delta'| \leq 3\varepsilon/2$. Spelling this out, the above analysis says that the floating-point dot product of the vectors x and y is equal to an exact dot product of the slightly perturbed inputs $\tilde{x}$ and $\tilde{y}$. This means that, by Definition 27, the dot product can be implemented in a backward stable way, with the backward error of its two input vectors each bounded by $3\varepsilon/2$.

A subtle point is that the backward error for multiplication can be described in a slightly different way, while still maintaining the same backward error bound given in Equation (4.6). In particular, floating-point multiplication behaves like multiplication in exact arithmetic with a *single* input subject to small perturbations: for any $b_1, b_2 \in \mathbb{R}$, we have

$$b_1 \odot_{\mathbb{F}} b_2 = b_1 \cdot b_2 e^{\delta} = b_1 \cdot \tilde{b}_2 \qquad (4.8)$$

with $|\delta| \leq \varepsilon$. There are many other ways to assign backward error to multiplication as long as the exponents sum to δ; in general, a given program may satisfy a variety of different backward error bounds depending on how the backward error is allocated between the program inputs.

The cost of using the backward error analysis for multiplication described in Equation (4.8) instead of Equation (4.6) is that all of the rounding error in the result of a floating-point multiplication

is assigned to a single input, rather than distributing half of the error to each input. We will see in Section 4.1.1, the payoff is that, in some cases, it enables a backward error analysis of computations that share variables across subexpressions.

4.1.1 Backward Error Analysis in **Bean**: Motivating Examples

In order to reason about backward error as it has been described so far, the type system of **Bean** combines three ingredients: coeffects, distances, and linearity. To get a sense of the critical role each of these components plays in the type system, we first consider the following **Bean** program for computing the dot product of 2D-vectors x and y:

```
// Bean program for the dot product of vectors x and y
DotProd2 x y :=
let (x0, x1) = x in
let (y0, y1) = y in
let v = mul x0 y0 in
let w = mul x1 y1 in
add v w
```

Coeffects

The type system of **Bean** allows us to prove the following typing judgment:

$$\emptyset \mid \mathrm{x} :_{3\varepsilon/2} \mathbb{R}^2, \mathrm{y} :_{3\varepsilon/2} \mathbb{R}^2 \vdash \mathrm{DotProd2} : \mathbb{R} \tag{4.9}$$

The *coeffect* annotations $3\varepsilon/2$ in the context bindings $\mathrm{x} :_{3\varepsilon/2} \mathbb{R}^2$ and $\mathrm{y} :_{3\varepsilon/2} \mathbb{R}^2$ express per-variable relative backward error bounds for $\mathrm{DotProd2}$. Thus, the typing judgment for $\mathrm{DotProd2}$ captures the desired backward error bound in Equation (4.7).

Coeffect systems (Ghica and Smith, 2014; Petricek et al., 2014; Brunel et al., 2014; Tate, 2013) have traditionally been used in the design of programming languages that perform resource management, and provide a formalism for precisely tracking the usage of variables in programs. In *graded coeffect systems* (Gaboardi et al., 2016), bindings in a typing context Γ are of the form $x :_r \sigma$,

where the annotation r is some quantity controlling how x can be used by the program. In **Bean**, these annotations describe the amount of backward error that can be assigned to the variable. In more detail, a typing judgment $\emptyset \mid x :_r \sigma \vdash e : \tau$ ensures that the term e has at most r backward error with respect to the variable x.

In **Bean**, the coeffect system allows us to derive backward error bounds for larger programs from the known language primitives; the typing rules are used to track the backward error of increasingly large programs in a compositional way. For instance, the typing judgment given in Equation (4.9) for the program $\mathrm{DotProd2}$ is derived using primitive typing rules for addition and subtraction. These rules capture the backward error bounds described in Equation (4.4) and Equation (4.6):

$$\frac{}{\emptyset \mid \Gamma, x :_{\varepsilon} \mathbb{R}, y :_{\varepsilon} \mathbb{R} \vdash \text{add } x\ y : \mathbb{R}} \text{(Add)} \qquad \frac{}{\emptyset \mid \Gamma, x :_{\varepsilon/2} \mathbb{R}, y :_{\varepsilon/2} \mathbb{R} \vdash \text{mul } x\ y : \mathbb{R}} \text{(Mul)}$$

The following rule similarly captures the backward error bound described in Equation (4.8):

$$\frac{}{x : \mathbb{R} \mid \Gamma, y :_{\varepsilon} \mathbb{R} \vdash \text{dmul } x\ y : \mathbb{R}} \text{(DMul)}$$

Distances

In order to derive concrete backward error bounds, we require a notion of *distance* between points in an input space. To this end, each type σ in **Bean** is equipped with a distance function $d_\sigma : \sigma \times \sigma \to \mathbb{R}_{\geq 0} \cup \{+\infty\}$ describing how close pairs of values of type σ are to one another. For instance, for our numeric type $\mathbb{R}$, choosing the *relative precision* metric (Definition 3) proposed by Olver (1978) for the distance function $d_{\mathbb{R}}$ allows us to prove backward error bounds for a relative notion of error. This idea is reminiscent of type systems capturing function sensitivity (Reed and Pierce, 2010; Gaboardi et al., 2013; Kellison and Hsu, 2024); however, the **Bean** type system does

not capture function sensitivity since this concept does not play a central role in backward error analysis.

Linearity

The conditions under which composing backward stable programs yields another backward stable program are poorly understood. Our development of a static analysis framework for backward error analysis led to the following insight: the composition of two backward stable programs remains backward stable *as long as they do not assign backward error to shared variables*. Thus, to ensure that our programs satisfy a backward stability guarantee, **Bean** features a *linear* typing discipline to control the duplication of variables. While most coeffect type systems allow using a variable x in two subexpressions as long as the grades $x :_r \sigma$ and $x :_s \sigma$ are combined in the overall program, **Bean** requires a stricter condition: linear variables cannot be duplicated at all.

To understand why a type system for backward error analysis should disallow unrestricted duplication, consider the floating-point computation corresponding to the evaluation of the polynomial $h(x) = ax^2 + bx$. The variable x is used in each of the subexpressions $f(x) = ax^2$ and $g(x) = bx$. Using the backward error bound given in Equation (4.6) for multiplication, the backward stability of f is guaranteed by the existence of the perturbed coefficient $\tilde{a} = ae^{\delta_1}$ and the perturbed variable $\tilde{x}_f = xe^{\delta_2}$:

$$\tilde{f}(x) = ae^{\delta_1} \cdot (xe^{\delta_2})^2 = \tilde{a} \cdot \tilde{x}_f^2 \tag{4.10}$$

Similarly, the backward stability of g is guaranteed by the existence of the perturbed coefficient $\tilde{b} = be^{\delta_3/2}$ and the variable $\tilde{x}_g = xe^{\delta_3/2}$. However, there is no common variable $\tilde{x}$ that ensures the stability of f and g simultaneously. That is, there is no input $\tilde{x}$ such that $f(\tilde{x}) + g(\tilde{x}) = \tilde{f}(x) + \tilde{g}(x)$.

By requiring linearity, **Bean** ensures that we never need to reconcile multiple backward error requirements for the same variable. However, this restriction can be quite limiting, and rules out the backward error analysis of some programs that are backward stable—for instance, the polynomial

$h(x) = ax^2 + bx$ above *is* actually backward stable! To regain flexibility in **Bean**, we note that there is a special situation when a variable *can* be duplicated safely: when it doesn't need to be perturbed in order to provide a backward error guarantee. For our polynomial $h(x)$, we can obtain a backward error guarantee using Equation (4.8) to assign zero backward error to the variable x and non-zero backward error to the coefficients a and b. Since x does not need to be perturbed in order to provide an overall backward error guarantee for $h(x)$, it can be duplicated without violating backward stability.

To realize this idea in **Bean**, the type system distinguishes between linear, restricted-use data and non-linear, reusable data. Linear variables are those we can assign backward error to during an analysis, while non-linear variables are those we do not assign backward error to during an analysis. Technically, **Bean** uses a dual context judgment, reminiscent of work on linear/non-linear logic (Benton, 1994), to track the two kinds of variables. In more detail, a typing judgment of the form $y : \alpha \mid x :_r \sigma \vdash e : \tau$ ensures that the term e has at most r backward error with respect to the *linear* variable x, and has *no backward error* with respect to the *non-linear* variable y. (Note that the bindings in the nonlinear context do not carry an index, because no amount of backward error can be assigned to these variables.) The soundness theorem for **Bean**, which we introduce in Section 4.4, formalizes this result.

4.2 Type System

Bean is a simple first-order programming language, extended with a few constructs that are unique to a language for backward error analysis. The grammar of the language is presented in Figure 4.2, and the typing relation is presented in Figure 4.3.

$$
\begin{aligned}
\alpha \quad &::= \ \mathrm{dnum} \mid \alpha \otimes \alpha && \text{(discrete types)}\\
\sigma \quad &::= \ \mathrm{unit} \mid \alpha \mid \mathrm{num} \mid \sigma \otimes \sigma \mid \sigma + \sigma && \text{(linear types)}\\
\Gamma \quad &::= \ \emptyset \mid \Gamma, x :_r \sigma && \text{(linear typing contexts)}\\
\Phi \quad &::= \ \emptyset \mid \Phi, z : \alpha && \text{(discrete typing contexts)}\\
e, f \ &::= \ x \mid z \mid () \mid \ !e \mid (e, f) \mid \mathbf{inl}\ e \mid \mathbf{inr}\ e\\
&\quad \mid \mathrm{let}\ x = e\ \mathrm{in}\ f \mid \mathrm{let}\ (x, y) = e\ \mathrm{in}\ f \mid \mathrm{dlet}\ z = e\ \mathrm{in}\ f \mid \mathrm{dlet}\ (x, y) = e\ \mathrm{in}\ f\\
&\quad \mid \mathbf{case}\ e\ \mathbf{of}\ (\mathbf{inl}\ x.f \mid \mathbf{inr}\ x.f) \mid \mathrm{add}\ e\ f \mid \mathrm{sub}\ e\ f \mid \mathrm{mul}\ e\ f \mid \mathrm{dmul}\ e\ f \mid \mathrm{div}\ e\ f\\
& && \text{(expressions)}
\end{aligned}
$$

Figure 4.2: Grammar for **Bean** types and terms.

4.2.1 Types

We use *linear* and *discrete* types to distinguish between linear, restricted-use data that can have backward error, and non-linear, unrestricted-use data that cannot: linear types σ are used for linear data, and discrete types α are used for non-linear data. Both linear and discrete types include a base numeric type, denoted by num and dnum, respectively. Linear types also include a tensor product $\otimes$, a unit type unit, and a sum type $+$.

4.2.2 Typing Judgments

Terms are typed with judgments of the form $\Phi, z : \alpha \mid \Gamma, x :_r \sigma \vdash e : \tau$ where Γ is a linear typing context and σ is a linear type, Φ is a discrete typing context and α is a discrete type, and e is an expression. For linear typing contexts, variable assignments have the form $x :_r \sigma$, where the grade r is a member of a preordered monoid $\mathcal{M} = (\mathbb{R}^{\geq 0}, +, 0)$. Typing contexts, both linear and discrete, are defined inductively as shown in Figure 4.2.

Although linear typing contexts cannot be joined together with discrete typing contexts, linear typing contexts can be joined with other linear typing contexts as long as their domains are disjoint. We write Γ, Δ to denote the disjoint union of the linear contexts Γ and Δ.

While most graded coeffect systems support the composition of linear typing contexts Γ and Δ via a *sum* operation $\Gamma + \Delta$ (Dal Lago and Gavazzo, 2022b; Gaboardi et al., 2016), where the grades of shared variables in the contexts are added together, this operation is not supported in **Bean**. This is because the sum operation serves as a mechanism for the restricted duplication of variables, but **Bean**'s strict linearity requirement does not allow variables to be duplicated. However, **Bean**'s type system does support a sum operation that adds a given grade $q \in \mathcal{M}$ to the grades in a linear typing context:

$$q + \emptyset = \emptyset$$

$$q + (\Gamma, x :_r \sigma) = q + \Gamma, x :_{q+r} \sigma.$$

In **Bean**, a well-typed expression $\Phi \mid x :_r \sigma \vdash e : \tau$ is a program that has at most r backward error with respect to the *linear* variable x, and has *no backward error* with respect to the *discrete* variables in the context Φ. For more general programs of the form

$$\Phi \mid x_1 :_{r_1} \sigma_1, \ldots, x_i :_{r_i} \sigma_i \vdash e : \tau,$$

Bean guarantees that the program has at most r_i backward error with respect to each variable x_i, and has *no backward error* with respect to the discrete variables in the context Φ. This idea is formally expressed in our soundness theorem (Theorem 10).

4.2.3 Expressions

Bean expressions include linear variables x and discrete variables z, as well as a unit () value. Linear variables are bound in let-bindings of the form let $x = e$ in f, while discrete variables are bound in let-bindings of the form dlet $z = e$ in f. The !-constructor is a syntactic convenience for declaring that an expression can be duplicated. The pair constructor (e, f) corresponds to a tensor product, and can be composed of expressions of both linear and discrete type. Discrete

pairs are eliminated by pattern matching using the construct dlet $(x, y) = e$ in f, whereas linear pairs are eliminated by pattern matching using the construct let $(x, y) = e$ in f. The injections **inl** e and **inr** e correspond to a coproduct, and are eliminated by case analysis using the construct **case** e **of** (**inl** $x.f$ | **inr** $y.f$). Some of the primitive arithmetic operations of the language (add, mul, dmul) were already introduced in Section 4.1. **Bean** also supports division (div) and subtraction (sub).

4.2.4 Typing relation

The full type system for **Bean** is given in Figure 4.3. It is parametric with respect to the constant $\varepsilon = u/(1-u)$, where u represents the unit roundoff.

Let us now describe the rules in Figure 4.3, starting with those that employ the sum operation between grades and linear typing contexts: the linear let-binding rule (Let) and the elimination rules for sums (+ E) and linear pairs ($\otimes$ E_σ). Using the intuition that a grade describes the backward error bound of a variable with respect to an expression, we see that whenever we have an expression e that is well-typed in a context Γ and we want to use e in place of a variable that has a backward error bound of r with respect to another expression, then we must assign r backward error onto the variables in Γ using the sum operation $r + \Gamma$. That is, if an expression e has a backward error bound of q with respect to a variable x and the expression f has backward error bound of r with respect to a variable y, then $f[e/y]$ will have backward error bound of $r + q$ with respect to the variable x.

The action of the !-constructor is illustrated in the Disc rule, which promotes an expression of linear numeric type to discrete numeric type. The !-constructor allows an expression to be used without restriction, but there is a drawback: once an expression is promoted to discrete type it can no longer be assigned backward error. The discrete let-binding rule (DLet) allows us to bind variables of discrete type.

$$\frac{}{\Phi \mid \Gamma, x :_r \sigma \vdash x : \sigma}\ (\text{Var}) \qquad \frac{}{\Phi, z : \alpha \mid \Gamma \vdash z : \alpha}\ (\text{DVar})$$

$$\frac{\Phi \mid \Gamma \vdash e : \sigma \qquad \Phi \mid \Delta \vdash f : \tau}{\Phi \mid \Gamma, \Delta \vdash (e, f) : \sigma \otimes \tau}\ (\otimes\ \text{I}) \qquad \frac{}{\Phi \mid \Gamma \vdash () : \text{unit}}\ (\text{Unit})$$

$$\frac{\Phi \mid \Gamma \vdash e : \tau_1 \otimes \tau_2 \qquad \Phi \mid \Delta, x :_r \tau_1, y :_r \tau_2 \vdash f : \sigma}{\Phi \mid r + \Gamma, \Delta \vdash \text{let } (x, y) = e \text{ in } f : \sigma}\ (\otimes\ \text{E}_\sigma)$$

$$\frac{\Phi \mid \Gamma \vdash e : \alpha_1 \otimes \alpha_2 \qquad \Phi, z_1 : \alpha_1, z_2 : \alpha_2 \mid \Delta \vdash f : \sigma}{\Phi \mid \Gamma, \Delta \vdash \text{dlet } (z_1, z_2) = e \text{ in } f : \sigma}\ (\otimes\ \text{E}_\alpha)$$

$$\frac{\Phi \mid \Gamma \vdash e' : \sigma + \tau \qquad \Phi \mid \Delta, x :_q \sigma \vdash e : \rho \qquad \Phi \mid \Delta, y :_q \tau \vdash f : \rho}{\Phi \mid q + \Gamma, \Delta \vdash \text{case } e' \text{ of } (x.e \mid y.f) : \rho}\ (+\ \text{E})$$

$$\frac{\Phi \mid \Gamma \vdash e : \sigma}{\Phi \mid \Gamma \vdash \textbf{inl}\ \ e : \sigma + \tau}\ (+\ \text{I}_\text{L}) \qquad \frac{\Phi \mid \Gamma \vdash e : \tau}{\Phi \mid \Gamma \vdash \textbf{inr}\ \ e : \sigma + \tau}\ (+\ \text{I}_\text{R})$$

$$\frac{\Phi \mid \Gamma \vdash e : \tau \qquad \Phi \mid \Delta, x :_r \tau \vdash f : \sigma}{\Phi \mid r + \Gamma, \Delta \vdash \text{let } x = e \text{ in } f : \sigma}\ (\text{Let})$$

$$\frac{\Phi \mid \Gamma \vdash e : \text{num}}{\Phi \mid \Gamma \vdash\ !e : \text{dnum}}\ (\text{Disc}) \qquad \frac{\Phi \mid \Gamma \vdash e : \alpha \qquad \Phi, z : \alpha \mid \Delta \vdash f : \sigma}{\Phi \mid \Gamma, \Delta \vdash \text{dlet } z = e \text{ in } f : \sigma}\ (\text{DLet})$$

$$\frac{}{\Phi \mid \Gamma, x :_{\varepsilon + r_1} \text{num}, y :_{\varepsilon + r_2} \text{num} \vdash \{\text{add}, \text{sub}\}\ x\ y : \text{num}}\ (\text{Add, Sub})$$

$$\frac{}{\Phi \mid \Gamma, x :_{\varepsilon/2 + r_1} \text{num}, y :_{\varepsilon/2 + r_2} \text{num} \vdash \text{mul}\ x\ y : \text{num}}\ (\text{Mul})$$

$$\frac{}{\Phi \mid \Gamma, x :_{\varepsilon/2 + r_1} \text{num}, y :_{\varepsilon/2 + r_2} \text{num} \vdash \text{div}\ x\ y : \text{num} + \textbf{err}}\ (\text{Div})$$

$$\frac{}{\Phi, z : \text{dnum} \mid \Gamma, x :_{\varepsilon + r} \text{num} \vdash \text{dmul}\ z\ x : \text{num}}\ (\text{DMul})$$

Figure 4.3: Typing rules with $q, r, r_1, r_2 \in \mathbb{R}^{\geq 0}$ and fixed parameter $\varepsilon \in \mathbb{R}^{>0}$.

Aside from the rules discussed above, the only remaining rules in Figure 4.3 that are not mostly standard are the rules for primitive arithmetic operations: addition (Add), subtraction (Sub), multiplication between two linear variables (Mul), division (Div), and multiplication between a discrete and non-linear variable (DMul). While these rules are designed to mimic the relative backward error bounds for floating-point operations following analyses described in the numerical analysis literature (Olver, 1978; Higham, 2002; Corless and Fillion, 2013) and as briefly introduced in Section 4.1, they also allow weakening, or relaxing, the backward error guarantee. Intuitively,

if the backward error of an expression with respect to a variable is *bounded* by ε, then it is also *bounded* by $\varepsilon + r$ for some grade $r \in \mathcal{M}$. We also note that division is a partial operation, where the error result indicates a division by zero.

The following section is devoted to explaining how a novel categorical semantics, where **Bean** typing judgments are interpreted as morphisms in the category **Bel** of *backward error lenses*, supports the language features we have described so far.

4.3 Denotational Semantics

Now that we have seen the syntax of **Bean**, we turn to its semantics. We first introduce a novel category **Bel** of *backward error lenses*, where morphisms are functions that satisfy a backward error guarantee. We show that this category supports a variety of useful constructions, which we use to interpret **Bean** programs. We will assume knowledge of the basic category theory concepts (e.g., categories and functors) that are briefly described in Section 2.2.3, introducing less well-known constructions as we go.

4.3.1 Bel: The Category of Backward Error Lenses

The key semantic structure for **Bean** is the category **Bel** of *backward error lenses*. Each morphism in **Bel** corresponds to a pair of functions describing the continuous problem and its approximating function, along with a *backward map* that serves as a constructive mechanism for witnessing the existence of a backward error result. We view the category **Bel** as conceptually similar to categories of *lenses* (Hofmann et al., 2011; Johnson et al., 2010; Riley, 2018; Johnson et al., 2012). Lenses, first introduced by Foster et al. (2007), consist of pairs of transformations between a set of source structures and a set of target structures: a *forward* transformation produces a target from a source

and a *backward* transformation "puts back" a modified target onto a source according to some laws (Fischer et al., 2015; Ko and Hu, 2017). More concretely, if X is a set of source structures and Y is a set of target structures, then a lens is comprised of a pair of functions known as *get* of type $X \to Y$ (the forward transformation) and *put* of type $X \times Y \to Y$ (the backward transformation). The category **Lens** of lenses then has sets as objects and lenses as morphisms, and supplies a well defined notion of the composition of two lenses (Riley, 2018).

In contrast to the traditional definition of lenses, *backward error lenses* consist of a *triple* of transformations:

Definition 28 (Backward Error Lenses)**.** A *backward error lens* $L : X \to Y$ is a triple $(f, \tilde{f}, b)$ of set-maps between the *generalized distance spaces* (X, d_X) and (Y, d_Y), described by the following data:

- the forward map $f : X \to Y$
- the approximation map $\tilde{f} : X \to Y$, and
- the backward map $b : X \times Y \to X$ defined as

$$\forall x \in X.\ b(x, -) \in \{y \in Y \mid d_Y(\tilde{f}(x), y) \neq \infty\} \to X$$

satisfying the properties

Property 1. $\forall x \in X, y \in Y.\ d_X(x, b(x, y)) \leq d_Y(\tilde{f}(x), y)$

Property 2. $\forall x \in X, y \in Y.\ f(b(x, y)) = y$

under the assumption that $d_Y(\tilde{f}(x), y) \neq \infty$.

The backward map $b : X \times Y \to X$ for backward error lenses given in Definition 28 maps a point $x \in X$ in the input space and a point $y \in Y$ in the output space *that is at finite distance from*

x under the approximation map $\tilde{f}$ (i.e., such that $d_{\mathrm{Y}}(\tilde{f}(x), y) \neq +\infty$) to a point $\tilde{x} \in \mathrm{X}$ in the input space. By restricting the backward map to points that are at finite distance in the output space under the approximation map, we can guarantee that the backward map produces a point in the input space that is at finite distance from the original input.

Properties 1 and 2 of Definition 28 are closely related to the *lens laws* described in the literature: for the forward transformation *get* of type $\mathrm{X} \to \mathrm{Y}$ and the backward transformation *put* of type $\mathrm{X} \times \mathrm{Y} \to \mathrm{Y}$, every lens must obey the following laws:

$$\forall x \in \mathrm{X}.\ put\ x\ (get\ x) = x \tag{4.11}$$

$$\forall x \in \mathrm{X}, \forall y \in \mathrm{Y}.\ get\ (put\ x\ y) = y \tag{4.12}$$

Clearly, property 2 of Definition 28 and Equation (4.12) are closely related. For backward error lenses, property 2 requires that the backward map precisely captures the backward error. To see why, consider instantiating property 2 with a point $x \in \mathrm{X}$ and $\tilde{f}(x) \in \mathrm{Y}$: the backward map $b(x, \tilde{f}(x))$ produces a point $\tilde{x} \in \mathrm{X}$ and property 2 requires that the backward error result $f(\tilde{x}) = \tilde{f}(x)$ holds.

Looking closely at property 1, we can see that it corresponds to a generalized Equation (4.11), reframed as an inequality. Where Equation (4.11) requires *put* to exactly restore the original point in the source space under strict conditions on its arguments, property 1 requires that the distance between the point produced by the backward map and the original point in the source space is bounded. The bound in property 2, namely the distance $d_{\mathrm{Y}}(\tilde{f}(x), y)$, serves as an upper bound for the generalized notion of backward error. To see how, consider that property 2, instantiated on a point $x \in \mathrm{X}$ and the point $\tilde{f}(x) \in \mathrm{Y}$, requires the following inequality to hold:

$$d_{\mathrm{X}}(x, b(x, \tilde{f}(x))) \leq d_{\mathrm{Y}}(\tilde{f}(x), \tilde{f}(x)). \tag{4.13}$$

Observe that if d_{X} and d_{Y} were distance functions with zero self distance, as is usual for standard metric spaces, then Equation (4.13) forces the backward error, given as the distance $d_{\mathrm{X}}(x, b(x, \tilde{f}(x)))$, to zero.

However, we would also like our semantics to support maps with bounded, but *non-zero* backward error. It turns out that we can support these more maps by allowing the distance functions d_X to take on a wide range of values, ranging over $\mathbb{R} \cup \{\pm\infty\}$; we call such functions *generalized distances*. While it is not obvious what a negative distance represents, intuitively, we merely use this distances as technical devices to enable compositional reasoning about backward error. For applications, all backward error guarantees will involve maps to *standard* metric spaces, i.e., with non-negative distance function satisfying the usual metric axioms.

Definition 29 (The Category **Bel** of Backward Error Lenses)**.** The category **Bel** of *backward error lenses* is the category with the following data:

- Its objects are generalized distance spaces: $(M, d : M \times M \to \mathbb{R} \cup \{\pm\infty\})$, where the distance function has non-positive self-distance: $d(x, x) \leq 0$.
- Its morphisms from X to Y are backward error lenses from X to Y: triples of maps $(f, \tilde{f}, b)$, satisfying the two properties in Definition 28.
- The identity morphism on objects X is given by the triple (id_X, id_X, π_2).
- The composition

$$(f_2, \tilde{f}_2, b_2) \circ (f_1, \tilde{f}_1, b_1)$$

  of error lenses $(f_1, \tilde{f}_1, b_1) : X \to Y$ and $(f_2, \tilde{f}_2, b_2) : Y \to Z$ is the error lens $(f, \tilde{f}, b) : X \to Z$ defined by

  - the forward map

$$f : x \mapsto (f_1; f_2)\ x \tag{4.14}$$

  - the approximation map

$$\tilde{f} : x \mapsto (\tilde{f}_1; \tilde{f}_2)\ x \tag{4.15}$$

  - the backward map

$$b : (x, z) \mapsto b_1\big(x, b_2(\tilde{f}_1(x), z)\big) \tag{4.16}$$

The composition in Definition 29 is only well-defined if the domain of the backward map is well-defined, and if the error lens properties hold for the composition; we check these requirements in Appendix B.1.

4.3.2 Basic Constructions in Bel

We can now begin defining the lenses in **Bel** that are necessary for interpreting the language features in **Bean**. Following the description of **Bean** given in Section 4.2, we give the constructions below for lenses corresponding to a tensor product, coproducts, and *a graded comonad* (see Section 2.2.3) for interpreting linear typing contexts.

Initial and Final Objects

We start by introducing the initial and final objects of our category. Let $0 \in \mathbf{Bel}$ be the empty metric space $(\emptyset, d_\emptyset)$, and $1 \in \mathbf{Bel}$ be the singleton metric space $(\{\star\}, \underline{0})$ with a single element and a constant distance function $d_1(\star, \star) = 0$. Then for any object $X \in \mathbf{Bel}$, there is a unique morphism $0_X : 0 \to X$ where the forward, approximate, and backward maps are all the empty map, so 0 is an *initial object* for **Bel**.

Similarly, for every object $X \in \mathbf{Bel}$ is a morphism $!_X : X \to 1$ given by $f_! = \tilde{f}_! := x \mapsto \star$ and $b_! := (x, \star) \mapsto x$. To check that this is indeed a morphism in **Bel**—we must check the two backward error lens conditions in Definition 28. The first condition boils down to checking $d_X(x, x) \leq d_1(\star, \star) = 0$, but this holds since all objects in **Bel** have non-positive self distance. The second condition is clear, since there is only one element in 1. Finally, this morphism is clearly unique, so 1 is a *terminal object* for **Bel**.

Tensor Product

Next, we turn to products in **Bel**. Like most lens categories, **Bel** does not support a Cartesian product (Hofmann et al., 2011). In particular, it is not possible to define a diagonal morphism $\Delta_A : A \to A \times A$, where the space $A \times A$ consists of pairs of elements of A. The problem is the second lens condition in Definition 28: given an approximate map $\tilde{f} : A \to A \times A$ and a backward map $b : A \times (A \times A) \to A$, we need to satisfy

$$\tilde{f}(b(a_0, (a_1, a_2))) = (a_1, a_2)$$

for all $(a_0, a_1, a_2) \in A$. But it is not possible to satisfy this condition when $a_1 \neq a_2$: the backward map can only return one of a_0, a_1, or a_2. As a consequence, there is not enough information for the approximate map $\tilde{f}$ to recover (a_1, a_2). More conceptually, this is the technical realization of the problem described in Section 4.1: if we think of a_1 and a_2 as backward error witnesses for two subcomputations that both use A, we may not be able to reconcile these two witnesses into a single backward error witness.

Although a Cartesian product does not exist, **Bel** does support a weaker, monoidal product, which makes it a *symmetric monoidal category*. Specifically, given two objects X and Y in **Bel** we have the object $(X \times Y, d_{X \otimes Y})$ where the metric $d_{X \otimes Y}$ takes the componentwise max. Additionally, given any two morphisms $(f, \tilde{f}, b) : A \to X$ and $(g, \tilde{g}, b') : B \to Y$, we have the morphism

$$(f, \tilde{f}, b) \otimes (g, \tilde{g}, b') : A \otimes B \to X \otimes Y$$

defined by:

- the forward map
$$(a_1, a_2) \mapsto (f(a_1), g(a_2)) \tag{4.17}$$

- the approximation map
$$(a_1, a_2) \mapsto (\tilde{f}(a_1), \tilde{g}(a_2)) \tag{4.18}$$

- the backward map

$$((a_1, a_2), (x_1, x_2)) \mapsto (b(a_1, x_1), b'(a_2, x_2)) \tag{4.19}$$

We check that the tensor product given in Equations (4.17) to (4.19) is well-defined in Appendix B.2.1.

Lemma 14. The tensor product operation on lenses induces a bifunctor on **Bel**.

The proof of Lemma 14 requires checking conditions expressing preservation of composition and identities, and is given in Appendix B.2.1.

The bifunctor $\otimes$ on the category **Bel** gives rise to a symmetric monoidal category of error lenses. The unit object I is defined to be the terminal object $1 = (\{\star\}, 0)$ with a single element and a constant distance function $d_I(\star, \star) = 0$ along with natural isomorphisms for the associator ($\alpha_{X,Y,Z} : X \otimes (Y \otimes Z)$), and we can define the usual left-unitor ($\lambda_X : I \otimes X \to X$), right-unitor ($\rho_X : X \otimes I \to X$), and symmetry ($\gamma_{X,Y} : X \otimes Y \to Y \otimes X$) maps. These definitions are provided in Appendix B.2.1.

Projections

For any two spaces X and Y with the same self distance, i.e., with $d_X(x, x) = d_Y(y, y)$ for all $x \in X$ and $y \in Y$, we can define a projection map $\pi_1 : X \otimes Y \to X$ via:

- the forward map

$$f : (x, y) \mapsto x$$

- the approximation map

$$\tilde{f} : (x, y) \mapsto x$$

- the backward map

$$b : ((x, y), z) \mapsto (z, y)$$

The projection $\pi_2 : X \otimes Y \to Y$ is defined similarly.

Coproducts

For any two objects X and Y in **Bel** we have the object $(X + Y, d_{X+Y})$, where the metric d_{X+Y} is defined as

$$d_{X+Y}(x, y) \triangleq \begin{cases} d_X(x_0, y_0) & \text{if } x = inl\ x_0 \text{ and } y = inl\ y_0 \\ d_Y(x_0, y_0) & \text{if } x = inr\ x_0 \text{ and } y = inr\ y_0 \\ \infty & \text{otherwise.} \end{cases} \tag{4.20}$$

We define the morphism for the first projection $in_1 : X \to X + Y$ as the triple

$$f_{in_1}(x) = \tilde{f}_{in_1}(x) \triangleq inl\ x \tag{4.21}$$

$$b_{in_1}(x, z) \triangleq \begin{cases} x_0 & \text{if } z = inl\ x_0 \\ x & \text{otherwise.} \end{cases} \tag{4.22}$$

We check that the first projection is well-defined in Appendix B.2.2. The morphism $in_2 : Y \to X+Y$ for the second projection can be defined similarly.

Now, given any two morphisms

$$g : X \to C \triangleq (f_g, \tilde{f}_g, b_g) \tag{4.23}$$

$$h : Y \to C \triangleq (f_h, \tilde{f}_h, b_h) \tag{4.24}$$

we define the unique *copairing* morphism $[g, h] : X + Y \to C$ such that $[g, h] \circ in_1 = g$ and $[g, h] \circ in_2 = h$ by the following triple:

$$f_{[g,h]}(z) \triangleq \begin{cases} f_g(x) & \text{if } z = inl\ x \\ f_h(y) & \text{if } z = inr\ y \end{cases} \tag{4.25}$$

$$\tilde{f}_{[g,h]}(z) \triangleq \begin{cases} \tilde{f}_g(x) & \text{if } z = inl\ x \\ \tilde{f}_h(y) & \text{if } z = inr\ y \end{cases} \tag{4.26}$$

$$b_{[g,h]}(z, c) \triangleq \begin{cases} inl\ (b_g(x, c)) & \text{if } z = inl\ x \\ inr\ (b_h(y, c)) & \text{if } z = inr\ y. \end{cases} \tag{4.27}$$

We check that the morphism $[g, h] : X + Y \to C$ is well-defined in Appendix B.2.2.

A Graded Comonad

Next, we turn to the key construction in **Bel** that enables our semantics to capture morphisms with non-zero backward error. The rough idea is to use a graded comonad to shift the distance by a numeric constant; this change then introduces slack into the lens conditions in Definition 28 to support backward error.

More precisely, we construct a comonad graded by the real numbers. Let the pre-ordered monoid $\mathcal{R}$ be the non-negative real numbers $R^{\geq 0}$ with the usual order and addition. We define a graded comonad on **Bel** by the family of functors

$$\{D_r : \mathbf{Bel} \to \mathbf{Bel} \mid r \in \mathcal{R}\}$$

as follows.

- The object-map $D_r : \text{Bel} \to \text{Bel}$ takes (X, d_X) to $(X, d_X - r)$, where $\pm\infty - r$ is defined to be equal to $\pm\infty$.
- The arrow-map D_r takes an error lens $(f, \tilde{f}, b) : A \to X$ to an error lens

  $$(D_r f, D_r \tilde{f}, D_r b) : D_r A \to D_r X \tag{4.28}$$

  where

  $$(D_r g)x \triangleq g(x). \tag{4.29}$$

- The *counit* map $\varepsilon_X : D_0 X \to X$ is the identity lens.
- The *comultiplication* map $\delta_{q,r,X} : D_{q+r} X \to D_q(D_r X)$ is the identity lens.
- The *2-monoidality* map $m_{r,X,Y} : D_r X \otimes D_r Y \xrightarrow{\sim} D_r(X \otimes Y)$ is the identity lens.

- The map $m_{q \le r, X} : D_r X \to D_q X$ is the identity lens.

Unlike similar graded comonads considered in the literature on coeffect systems, our graded monad does not support a graded contraction map: there is no lens morphism $c_{r,s,A} : D_{r+s}A \to D_r A \otimes D_s A$. This is for the same reason that our category does not support diagonal maps: it is not possible to satisfy the second lens condition in Definition 28. Thus, we have a graded comonad, rather than a graded exponential comonad (Brunel et al., 2014).

Discrete Objects

While there is no morphism $A \to A \otimes A$ in general, graded or not, there is a special class of objects where we do have a diagonal map: the *discrete* spaces.

Definition 30 (Discrete space)**.** We say a generalized distance space (X, d_X) is *discrete* if its distance function satisfies $d_X(x_1, x_2) = \infty$ for all $x_1 \neq x_2$.

We write **Del** for category of the discrete spaces and backward error lenses; this forms a full subcategory of **Bel**. Discrete objects are closed under the monoidal product in **Bel**, and the unit object I is discrete.

There are two other key facts about discrete objects. First, it is possible to define a diagonal lens.

Lemma 15 (Discrete diagonal)**.** For any discrete object $X \in$ **Del** there is a lens morphism $t_X : X \to X \otimes X$ defined via

$$f_t = \tilde{f}_t \triangleq x \mapsto (x, x)$$

$$b_t \triangleq (x, (x_1, x_2)) \mapsto x.$$

The key reason this is a lens morphism is that according to the lens requirements in Definition 28, we only need to establish the lens properties for (x_1, x_2) at *finite* (i.e., not equal to $+\infty$) distance from

$f_t(x) = (x, x)$ under the distance on $X \otimes X$. Since this is a discrete space, we only need to consider pairs (x_1, x_2) that are *equal* to (x, x); thus, the lens conditions are obvious. More conceptually, we can think of a discrete object as a space that can't have any backward error pushed onto it. Thus, the backward error witnesses x_1 and x_2 are always equal to the input, and can always be reconciled.

Second, the graded comonad D_r restricts to a graded comonad on **Del**. In particular, if X is a discrete object, then $D_r X$ is also a discrete object.

4.3.3 Interpreting Bean

With the basic structure of **Bel** in place, we can now interpret the types and typing judgments of **Bean** as objects in **Bel** and morphisms in **Bel**, respectively. Given a type τ, we define a metric space $[\![\tau]\!]$ with the rules

$$
\begin{aligned}
[\![\mathrm{dnum}]\!] &\triangleq (\mathbb{R}, d_\alpha) \\
[\![\mathrm{num}]\!] &\triangleq (\mathbb{R}, d_\mathbb{R}) \\
[\![\sigma \otimes \tau]\!] &\triangleq [\![\sigma]\!] \otimes [\![\tau]\!] \\
[\![\sigma + \tau]\!] &\triangleq [\![\sigma]\!] + [\![\tau]\!] \\
[\![\mathbf{unit}]\!] &\triangleq (\{\star\}, \underline{0})
\end{aligned}
$$

where the distance function d_α is the discrete metric on $\mathbb{R}$ where self-distance is zero and the distance between two distinct point is $+\infty$, and $d_\mathbb{R}$ is the relative precision metric (Definition 3). By definition, if $d_\mathbb{R}$ is a standard distance function, then $[\![\tau]\!]$ is a standard metric space.

The interpretation of typing contexts is defined inductively as follows:

$$\llbracket \emptyset \mid \emptyset \rrbracket \triangleq \mathrm{I}$$

$$\llbracket \emptyset \mid \Gamma, x :_r \sigma \rrbracket \triangleq \llbracket \Gamma \rrbracket \otimes \mathrm{D}_r \llbracket \sigma \rrbracket$$

$$\llbracket \Phi, z : \alpha \mid \emptyset \rrbracket \triangleq \llbracket \Phi \rrbracket \otimes \llbracket \alpha \rrbracket$$

$$\llbracket \Phi, z : \alpha \mid \Gamma, x :_r \sigma \rrbracket \triangleq \llbracket \Phi \rrbracket \otimes \llbracket \alpha \rrbracket \otimes \llbracket \Gamma \rrbracket \otimes \mathrm{D}_r \llbracket \sigma \rrbracket$$

where the graded comonad D_r is used to interpret the linear variable assignment $x :_r \sigma$, and I is the monoidal unit $(\{\star\}, \underline{-\infty})$.

Given the above interpretations of types and typing environments, we can interpret **Bean** programs in **Bel**:

Definition 31. (Interpretation of **Bean** Terms.) We can interpret each well-typed term $\Phi \mid \Gamma \vdash e : \tau$ as an error lens $\llbracket e \rrbracket : \llbracket \Phi \rrbracket \otimes \llbracket \Gamma \rrbracket \to \llbracket \tau \rrbracket$ in **Bel**, by structural induction on the typing derivation.

The details of each construction for Definition 31 can be found in Appendix B.3. We provide the cases for the (Let), (Add), and (Mul) rules here as a demonstration.

Case (Let). Given the maps

$$h_1 = \llbracket \Phi \mid \Gamma \vdash e : \tau \rrbracket : \llbracket \Phi \rrbracket \otimes \llbracket \Gamma \rrbracket \to \llbracket \tau \rrbracket$$

$$h_2 = \llbracket \Phi \mid \Delta, x :_r \tau \vdash f : \sigma \rrbracket : \llbracket \Phi \rrbracket \otimes \llbracket \Delta \rrbracket \otimes \mathrm{D}_r \llbracket \tau \rrbracket \to \llbracket \sigma \rrbracket$$

we need to define a map $\llbracket \Phi \mid r + \Gamma, \Delta \vdash \mathrm{let}\ x = e\ \mathrm{in}\ f : \sigma \rrbracket$.

We first define a map $h : \mathrm{D}_r \llbracket \Phi \rrbracket \otimes \mathrm{D}_r \llbracket \Gamma \rrbracket \otimes \llbracket \Delta \rrbracket \to \llbracket \sigma \rrbracket$ as the following composition:

$$h_2 \circ (\mathrm{D}_r(h_1) \otimes (\varepsilon_{\llbracket \Phi \rrbracket} \circ m_{0 \le r, \llbracket \Phi \rrbracket}) \otimes id_{\llbracket \Delta \rrbracket}) \circ (m_{r, \llbracket \Phi \rrbracket, \llbracket \Gamma \rrbracket} \otimes id_{\mathrm{D}_r \llbracket \Phi \rrbracket} \otimes id_{\llbracket \Delta \rrbracket}) \circ (t_{\mathrm{D}_r \llbracket \Phi \rrbracket} \otimes id_{\mathrm{D}_r \llbracket \Gamma \rrbracket \otimes \llbracket \Delta \rrbracket}).$$

Here, the map ε is the counit of the graded comonad.

Since $[\![\sigma]\!]$ is a metric space, its distance is bounded below by 0. Since $[\![\Phi]\!]$ is a discrete space, we observe that forward, approximate, and backward maps in h are also a lens morphism between objects:

$$h : [\![\Phi]\!] \otimes \mathrm{D}_r[\![\Gamma]\!] \otimes [\![\Delta]\!] \to [\![\sigma]\!]$$

This is the desired map to interpret let-binding.

Case (Add). Suppose the contexts Φ and Γ have total length i. We define the map

$$[\![\Phi \mid \Gamma, x :_{\varepsilon+q} \text{num}, y :_{\varepsilon+r} \text{num} \vdash \text{add } x\ y : \text{num}]\!]$$

as the composition

$$\pi_i \circ (id_{[\![\Phi]\!]} \otimes (\overline{\varepsilon_{[\![\sigma_j]\!]}} \circ \overline{m_{0\leq q_j,[\![\sigma_j]\!]}}) \otimes id_{[\![\text{num}]\!]}) \circ (id_{[\![\Phi]\!]\otimes[\![\Gamma]\!]} \otimes \mathrm{L}_{add}) \circ (id_{[\![\Phi]\!]\otimes[\![\Gamma]\!]} \otimes m_{\varepsilon\leq\varepsilon+q,[\![\text{num}]\!]} \otimes m_{\varepsilon\leq\varepsilon+r,[\![\text{num}]\!]}),$$

where the map $\overline{\varepsilon_{[\![\sigma_j]\!]}}$ applies the counit map $\varepsilon_{\mathrm{X}} : \mathrm{D}_0\mathrm{X} \to \mathrm{X}$ to each object in the context $[\![\Gamma]\!]$, and the map $\overline{m_{0\leq q_j,[\![\sigma_j]\!]}}$ applies the map $m_{0\leq q,\mathrm{X}} : \mathrm{D}_q\mathrm{X} \to \mathrm{D}_0\mathrm{X}$ to each binding $[\![x :_q \sigma]\!]$ in the context $[\![\Gamma]\!]$.

The lens $\mathrm{L}_{add} : \mathrm{D}_\varepsilon(\mathbb{R}) \otimes \mathrm{D}_\varepsilon(\mathbb{R}) \to \mathbb{R}$ is given by the triple

$$\begin{aligned}
f_{add}(x_1, x_2) &\triangleq x_1 + x_2 \\
\tilde{f}_{add}(x_1, x_2) &\triangleq (x_1 + x_2)e^{\delta}; \quad |\delta| \leq \varepsilon \\
b_{add}((x_1, x_2), x_3) &\triangleq \left(\frac{x_3 x_1}{x_1 + x_2}, \frac{x_3 x_2}{x_1 + x_2}\right)
\end{aligned}$$

where $\varepsilon = u/(1-u)$ and u is the unit roundoff.

We now show $\mathrm{L}_{add} : \mathrm{D}_\varepsilon(\mathbb{R}) \otimes \mathrm{D}_\varepsilon(\mathbb{R}) \to \mathbb{R}$ is well-defined: it is clear that L_{add} satisfies property 2 of a backward error lens, and so we are left with checking property 1: Assuming, for any $x_1, x_2, x_3 \in \mathbb{R}$,

$$d_{\mathrm{R}}\left(\tilde{f}_{add}(x_1, x_2), x_3\right) \neq \infty, \tag{4.30}$$

we are required to show

$$d_{\mathbb{R}\otimes\mathbb{R}}\left((x_1, x_2), b_{add}((x_1, x_2), x_3)\right) - \varepsilon \leq d_{\mathbb{R}}\left(\tilde{f}_{add}(x_1, x_2), x_3\right) = d_{\mathbb{R}}\left((x_1 + x_2)e^{\delta}, x_3\right).$$

Note that Equation (4.30) implies $d_R((x_1+x_2)e^{\delta}, x_3) \neq \infty$: by Equation (2.7), we have that (x_1+x_2) and x_3 are either both zero, or are both non-zero and of the same sign. We can assume, without loss of generality,

$$d_{\mathbb{R}}\left(x_2, \frac{x_3 x_2}{x_1+x_2}\right) \leq d_{\mathbb{R}}\left(x_1, \frac{x_3 x_1}{x_1+x_2}\right). \tag{4.31}$$

Under this assumption, we have

$$d_{\mathbb{R}\otimes\mathbb{R}}\left((x_1,x_2), b_{add}((x_1,x_2),x_3)\right) = d_{\mathbb{R}}\left(x_1, \frac{x_3 x_1}{x_1+x_2}\right) \tag{4.32}$$

and we are then required to show

$$d_{\mathbb{R}}\left(x_1, \frac{x_3 x_1}{x_1+x_2}\right) \leq d_{\mathbb{R}}\left((x_1+x_2)e^{\delta}, x_3\right) + \varepsilon. \tag{4.33}$$

Using the distance function given in Equation (2.7), the inequality in Equation (4.33) becomes

$$\left|\ln\left(\frac{x_1+x_2}{x_3}\right)\right| \leq \left|\ln\left(\frac{x_1+x_2}{x_3}\right) + \delta\right| + \varepsilon, \tag{4.34}$$

which holds under the assumptions of $|\delta| \leq \varepsilon$ and $0 < \varepsilon$. Set $\alpha = |\ln((x_1+x_2)/x_3)|$, and assume, without loss of generality, that $\alpha < 0$. If $\alpha+\delta < 0$ then $|\alpha| = -\alpha$ and $|\alpha+\delta| = -(\alpha+\delta)$; the inequality in Equation (4.34) reduces to $\delta \leq \varepsilon$, which follows by assumption. Otherwise, if $0 \leq \alpha+\delta$, then $-\alpha \leq \delta \leq \varepsilon$ and it suffices to show that $\varepsilon \leq \alpha+\delta+\varepsilon$.

Case (Mul). We proceed the same as the case for (Add), with slightly different indices. We define a lens $\mathcal{L}_{mul} : \mathrm{D}_{\varepsilon/2}(\mathbb{R}) \otimes \mathrm{D}_{\varepsilon/2}(\mathbb{R}) \to \mathrm{R}$ given by the triple

$$\begin{aligned}
f_{mul}(x_1,x_2) &\triangleq x_1 x_2 \\
\tilde{f}_{mul}(x_1,x_2) &\triangleq x_1 x_2 e^{\delta}; \quad |\delta| \leq \varepsilon \\
b_{mul}((x_1,x_2),x_3) &\triangleq \left(x_1\sqrt{\frac{x_3}{x_1x_2}}, x_2\sqrt{\frac{x_3}{x_1x_2}}\right).
\end{aligned}$$

We check that $\mathcal{L}_{mul} : \mathrm{D}_{\varepsilon/2}(\mathrm{R}) \otimes \mathrm{D}_{\varepsilon/2}(\mathrm{R}) \to \mathrm{R}$ is well-defined.

For any $x_1, x_2, x_3 \in \mathrm{R}$ such that

$$d_{\mathrm{R}}\left(\tilde{f}_{mul}(x_1,x_2), x_3\right) \neq \infty. \tag{4.35}$$

holds, we need to check the that $\mathcal{L}_{mul}$ satisfies the properties of an error lens. We again take the distance function d_{R} as the metric given in Equation (2.7), so Equation (4.35) implies that $(x_1 x_2)$ and x_3 are either both zero or are both non-zero and of the same sign; this guarantees that the backward map (containing square roots) is indeed well defined.

Property 1. We are required to show

$$\begin{aligned} d_{\mathrm{R}\otimes\mathrm{R}}\left((x_1, x_2), b_{mul}((x_1, x_2), x_3)\right) - \varepsilon/2 &\leq d_{\mathrm{R}}\left(\tilde{f}_{mul}(x_1, x_2), x_3\right) \\ &\leq d_{\mathrm{R}}\left(x_1 x_2 e^{\delta}, x_3\right) \end{aligned}$$

Unfolding the definition of the distance function (Equation (2.7)), we have

$$\begin{aligned} d_{\mathrm{R}\otimes\mathrm{R}}\left((x_1, x_2), b_{mul}((x_1, x_2), x_3)\right) &= d_{\mathrm{R}}\left(x_1, x_1\sqrt{\frac{x_3}{x_1 x_2}}\right) \\ &= d_{\mathrm{R}}\left(x_2, x_2\sqrt{\frac{x_3}{x_1 x_2}}\right) \\ &= \frac{1}{2}\left|\ln\left(\frac{x_1 x_2}{x_3}\right)\right|, \end{aligned}$$

and so we are required to show

$$\frac{1}{2}\left|\ln\left(\frac{x_1 x_2}{x_3}\right)\right| \leq \left|\ln\left(\frac{x_1 x_2}{x_3}\right) + \delta\right| + \frac{1}{2}\varepsilon \tag{4.36}$$

which holds under the assumptions of $|\delta| \leq \varepsilon$ and $0 < \varepsilon$. Setting $\alpha = \ln(x_1 x_2/x_3)$, assume, without loss of generality, that $\alpha < 0$. If $\alpha + \delta < 0$ then $\alpha < -\delta$ and it suffices to show that $-\frac{1}{2}\delta \leq -\delta + \frac{1}{2}\varepsilon$, which follows by assumption. Otherwise, if $0 \leq \alpha + \delta$ then $-\alpha \leq \delta$ and it suffices to show that $\frac{1}{2}\delta \leq \alpha + \delta + \frac{1}{2}\varepsilon$, which follows by assumption.

Property 2.

$$f_{mul}\left(b_{mul}((x_1, x_2), x_3)\right) = f_{mul}\left(x_1\sqrt{\frac{x_3}{x_1 x_2}}, x_2\sqrt{\frac{x_3}{x_1 x_2}}\right) = x_3.$$

4.4 Backward Error Soundness

Recall the intuition behind the guarantee for **Bean**'s type system: a well-typed term of the form

$$\Phi \mid x_1 :_{r_1} \sigma_1, \ldots, x_i :_{r_i} \sigma_i \vdash e : \tau$$

is a program that has at most r_i backward error with respect to each variable x_i, and has no backward error with respect to the discrete variables in the context Φ. By interpreting **Bean** programs as morphisms in **Bel**, we are able to precisely describe how every **Bean** program captures the complex interaction between an ideal problem, its approximate program, and a map that constructs the backward error between them. Using these constructions **Bel**, we can clearly see a path towards a *backward error soundness theorem*: given the interpretation

$$[\![z_1 : \alpha_1, \ldots, z_j : \alpha_j \mid x_1 :_{r_1} \sigma_1, \ldots, x_i :_{r_i} \sigma_i \vdash e : \tau]\!] = (f, \tilde{f}, b) : [\![\alpha_1]\!] \otimes \cdots \otimes D_{r_i}[\![\sigma_i]\!] \to [\![\tau]\!]$$

if $\tilde{f}[u_1/z_1] \ldots [v_1/x_1]$ evaluates to a value w for the well-typed substitutions $(u)_{1 \le n \le j}$ and $(v)_{1 \le n \le i}$, then we can guarantee our desired backward error result if we can witness the existence of a well-typed substitution $(\tilde{v})_{1 \le n \le i}$ such that $f[u_1/z_1] \ldots [\tilde{v}_1/x_1]$ also evaluates to the value w, and $d_{\sigma_i}(v_i, \tilde{v}_i) \le r_i$; our backward map b can be used to construct the required witness.

Formalizing the above result requires explicit access to each transformation in the backward error lens individually. We achieve this by defining an intermediate language, which we call Λ_S, where programs denote morphisms in **Set**. We then define an ideal and approximate operational semantics for Λ_S, and relate these semantics to the backward error lens semantics of **Bean** via the **Set** semantics of Λ_S. As we will see, the semantic constructions for **Bean** can be transformed in a straightforward way to semantic constructions for Λ_S using the forgetful functors $U_{id} : \mathbf{Bel} \to \mathbf{Set}$ and $U_{ap} : \mathbf{Bel} \to \mathbf{Set}$; these functors associate each metric space with its underlying set, and associate each backward error lens with its underlying ideal (resp., approximate) function on sets. The actions of the forgetful functors U_{id} and U_{ap} on objects are both denoted by U for simplicity.

$$\frac{}{\Gamma, x : \sigma, \Delta \vdash x : \sigma}\ (\text{Var}) \qquad \frac{}{\Gamma \vdash () : \text{unit}}\ (\text{Unit}) \qquad \frac{k \in R}{\Gamma \vdash k : \textbf{num}}\ (\text{Const})$$

$$\frac{\Phi, \Gamma \vdash e : \sigma \qquad \Phi, \Delta \vdash f : \tau}{\Phi, \Gamma, \Delta \vdash (e, f) : \sigma \otimes \tau}\ (\otimes\ \text{I})$$

$$\frac{\Phi, \Gamma \vdash e : \tau_1 \otimes \tau_2 \qquad \Phi, \Delta, x : \tau_1, y : \tau_2 \vdash f : \sigma}{\Phi, \Gamma, \Delta \vdash \text{let } (x, y) = e \text{ in } f : \sigma}\ (\otimes\ \text{E})$$

$$\frac{\Phi, \Gamma \vdash e' : \sigma + \tau \qquad \Phi, \Delta, x : \sigma \vdash e : \rho \qquad \Phi, \Delta, y : \tau \vdash f : \rho}{\Phi, \Gamma, \Delta \vdash \text{case } e' \text{ of } (\textbf{inl}\ x.e \mid \textbf{inr}\ y.f) : \rho}\ (+\ \text{E})$$

$$\frac{\Phi, \Gamma \vdash e : \sigma}{\Phi, \Gamma \vdash \textbf{inl}\ \ e : \sigma + \tau}\ (+\ \text{I}_\text{L}) \qquad \frac{\Phi, \Gamma \vdash e : \tau}{\Phi, \Gamma \vdash \textbf{inr}\ \ e : \sigma + \tau}\ (+\ \text{I}_\text{R})$$

$$\frac{\Phi, \Gamma \vdash e : \tau \qquad \Phi, \Delta, x : \tau \vdash f : \sigma}{\Phi, \Gamma, \Delta \vdash \text{let } x = e \text{ in } f : \sigma}\ (\text{Let})$$

$$\frac{\Phi, \Gamma \vdash e : \text{num} \qquad \Phi, \Delta \vdash f : \text{num} \qquad \text{Op} \in \{\text{add}, \text{sub}, \text{mul}, \text{dmul}\}}{\Phi, \Gamma, \Delta \vdash \text{Op}\ e\ f : \text{num}}\ (\text{Op})$$

$$\frac{\Phi, \Gamma \vdash e : \text{num} \qquad \Phi, \Delta \vdash f : \text{num}}{\Phi, \Gamma, \Delta \vdash \text{div}\ e\ f : \text{num} + \textbf{err}}\ (\text{Div})$$

Figure 4.4: Full typing rules for Λ_S.

4.4.1 Λ_S: A Language for Projecting Bean into Set

A Type System for Λ_S

The type system of Λ_S corresponds closely to **Bean**'s. Terms are typed with judgments of the form $\Phi, \Gamma \vdash e : \tau$, where the typing context Γ corresponds to the linear typing contexts of **Bean** with all of the grade information erased, and the typing context Φ corresponds to the discrete typing contexts of **Bean**; we will denote the erasure of grade information from a linear typing environment Δ as Δ°. Under the erasure of grade information from a linear context Δ, the disjoint union of the contexts Φ, Δ° is well-defined.

In contrast to **Bean**, types in Λ_S are not categorized as linear and discrete:

$$\sigma \ ::= \ \text{num} \mid \text{unit} \mid \sigma \otimes \sigma \mid \sigma + \sigma \qquad (\Lambda_S \text{ types})$$

The grammar of terms in Λ_S is mostly unchanged from the grammar of **Bean**, except that Λ_S extends **Bean** to include primitive constants drawn from a signature R:

$$e, f \ ::= \ \cdots \mid k \in \mathrm{R} \qquad (\Lambda_S \text{ terms})$$

The typing relation of Λ_S is entirely standard for a first-order simply typed language; the full set of rules is given in Figure 4.4. The close correspondence between derivations in **Bean** and derivations in Λ_S is summarized in the following lemma.

Lemma 16. Let $\Phi \mid \Gamma \vdash e : \tau$ be a well-typed term in **Bean**. Then there is a derivation of $\Phi, \Gamma^\circ \vdash e : \tau$ in Λ_S.

The proof of Lemma 16 is given in Appendix B.5.

The proof of backward error soundness requires that Λ_S satisfies the basic properties of weakening and substitution:

Lemma 17 (Weakening)**.** Let $\Gamma \vdash e : \tau$ be a well-typed Λ_S term. Then for any typing environment Δ disjoint with Γ, there is a derivation of $\Gamma, \Delta \vdash e : \tau$.

In the following theorem statement, we write $e[v/x]$ for the capture avoiding substitution of the value v for all free occurrences of x in e. Given a typing environment $x_1 : \tau_1, \ldots, x_i : \tau_i = \Gamma$, we denote the simultaneous substitution of a vector of values $v_1, \ldots, v_i = \bar{v}$ for the variables in Γ as $e[\bar{v}/dom(\Gamma)]$. Additionally, for a vector $\gamma_1, \ldots, \gamma_i = \bar{\gamma}$ of well-typed closed values and a typing environment $x_1 : \tau_1, \ldots, x_i : \tau_i = \Gamma$ (note the assumption that γ and Γ have the same length) we write $\bar{\gamma} \vDash \Gamma$ to denote the following

$$\bar{\gamma} \vDash \Gamma \triangleq \forall x_i \in dom(\Gamma).\ \emptyset \vdash \gamma_i : \Gamma(x_i). \qquad (4.37)$$

Theorem 6 (Substitution)**.** Let $\Gamma \vdash e : \tau$ be a well-typed Λ_S term. Then for any well-typed substitution $\bar{\gamma} \vDash \Gamma$ of closed values, there is a derivation $\emptyset \vdash e[\bar{\gamma}/dom(\Gamma)] : \tau$.

Most cases for substitution are routine; we provide the details of the proof in Appendix B.5.

$$\frac{}{() \Downarrow ()} \qquad \frac{e \Downarrow u \qquad f \Downarrow v}{(e, f) \Downarrow (u, v)} \qquad \frac{e \Downarrow (u, v) \qquad f[u/x][v/y] \Downarrow w}{\text{let } (x, y) = e \text{ in } f \Downarrow w}$$

$$\frac{}{k \in \mathrm{R} \Downarrow k \in \mathrm{R}} \qquad \frac{e \Downarrow v}{\textbf{inl } e \Downarrow \textbf{inl } v} \qquad \frac{e \Downarrow v}{\textbf{inr } e \Downarrow \textbf{inr } v}$$

$$\frac{e \Downarrow u \qquad f[u/x] \Downarrow v}{\text{let } x = e \text{ in } f \Downarrow v}$$

$$\frac{e \Downarrow \textbf{inl } v \qquad e_1[v/x] \Downarrow w}{\text{case } e \text{ of } (x.e_1 \mid y.e_2) \Downarrow w} \qquad \frac{e \Downarrow \textbf{inr } v \qquad e_2[v/y] \Downarrow w}{\text{case } e \text{ of } (x.e_1 \mid y.e_2) \Downarrow w}$$

$$\frac{e_1 \Downarrow_{id} k_1 \qquad e_2 \Downarrow_{id} k_2 \qquad \text{Op} \in \{\text{Add, Sub, Mul, Div, LE}\}}{\text{Op } e_1\ e_2 \Downarrow_{id} f_{op}\ (k_1, k_2)}$$

$$\frac{e_1 \Downarrow_{ap} k_1 \qquad e_2 \Downarrow_{ap} k_2 \qquad \text{Op} \in \{\text{Add, Sub, Mul, Div, LE}\}}{\text{Op } e_1\ e_2 \Downarrow_{ap} \tilde{f}_{op}\ (k_1, k_2)}$$

Figure 4.5: Evaluation rules for Λ_S. A generic step relation ($\Downarrow$) is used when the rule is identical for both the ideal ($\Downarrow_{id}$) and approximate ($\Downarrow_{ap}$) step relations.

An Operational Semantics for Λ_S

Intuitively, an ideal problem and its approximating program can behave differently given the same input. Following this intuition, we allow programs in Λ_S to be executed under an ideal or approximate big-step operational semantics. The full set of evaluation rules is given in Figure 4.5. We write $e \Downarrow_{id} v$ (resp., $e \Downarrow_{ap} v$) to denote that a term e evaluates to value v under the ideal (resp., approximate) semantics. Values, the subset of terms that are allowed as results of evaluation, are defined as follows.

$$\text{Values } v \ ::= \ () \mid k \in \mathrm{R} \mid (v, v) \mid \textbf{inl } v \mid \textbf{inr } v$$

An important feature of Λ_S is that it is deterministic and strongly normalizing:

Theorem 7 (Strong Normalization)**.** If $\emptyset \vdash e : \tau$, then the well-typed closed values $\emptyset \vdash v, v' : \tau$ exist such that $e \Downarrow_{id} v$ and $e \Downarrow_{ap} v'$.

In our main result of backward error soundness, we will relate the ideal and approximate

operational semantics given above to the backward error lens semantics of **Bean** via an interpretation of programs in Λ_S as morphisms in the category **Set**.

4.4.2 Interpreting Λ_S

Our main backward error soundness theorem requires that we have explicit access to each transformation in a backward error lens. We achieve this by lifting the close syntactic correspondence between Λ_S and **Bean** to a close semantic correspondence using the forgetful functors $U_{id} : \mathbf{Bel} \to \mathbf{Set}$ and $U_{ap} : \mathbf{Bel} \to \mathbf{Set}$ to interpret Λ_S programs in **Set**.

We start with the interpretation of Λ_S types, defined as follows

$$
\begin{aligned}
(\!|\mathrm{num}|\!) &\triangleq \mathrm{U}[\![\mathrm{num}]\!] = \mathrm{U}[\![\mathrm{dnum}]\!] \\
(\!|\mathbf{unit}|\!) &\triangleq \mathrm{U}[\![(\{\star\}, \underline{0})]\!] \\
(\!|\sigma \otimes \tau|\!) &\triangleq \mathrm{U}[\![\sigma]\!] \times \mathrm{U}[\![\tau]\!] \\
[\![\sigma + \tau]\!] &\triangleq \mathrm{U}[\![\sigma]\!] + \mathrm{U}[\![\tau]\!]
\end{aligned}
$$

Given the above interpretation of types, the interpretation $(\!|\Gamma|\!)$ of a Λ_S typing context Γ is then defined as

$$
\begin{aligned}
(\!|\emptyset|\!) &\triangleq \mathrm{U}[\![(\{\star\}, \underline{0})]\!] \\
(\!|\Gamma, x : \sigma|\!) &\triangleq (\!|\Gamma|\!) \otimes (\!|\sigma|\!)
\end{aligned}
$$

Now, using the above definitions for the interpretations of Λ_S types and contexts, we can use the interpretation of **Bean** (Definition 31) terms along with the functors U_{id} and U_{ap} to define the interpretation of Λ_S programs as morphisms in **Set**:

Definition 32. (Interpretation of Λ_S terms.) Each typing derivation $\Gamma \vdash e : \tau$ in Λ_S yields the set maps $(\!|e|\!)_{id} : (\!|\Gamma|\!) \to (\!|\tau|\!)$ and $(\!|e|\!)_{ap} : (\!|\Gamma|\!) \to (\!|\tau|\!)$, by structural induction on the Λ_S typing derivation $\Gamma \vdash e : \tau$.

We give the detailed constructions for Definition 32 in Appendix B.4.

Given Definition 32, we can now show that Λ_S is semantically sound and compuationally adequate: a Λ_S program computes to a value if and only if their interpretations in **Set** are equal. Because Λ_S has an ideal and approximate operational semantics as well as an ideal and approximate denotational semantics, we have two version of the standard theorems for soundness and adequacy:

Theorem 8 (Soundness of $(\!|-|\!)$)**.** Let $\Gamma \vdash e : \tau$ be a well-typed Λ_S term. Then for any well-typed substitution of closed values $\bar{\gamma} \vDash \Gamma$, if $e[\bar{\gamma}/dom(\Gamma)] \Downarrow_{id} v$ for some value v, then $(\!|\Gamma \vdash e : \tau|\!)_{id}(\!|\bar{\gamma}|\!)_{id} = (\!|v|\!)_{id}$ (and similarly for $\Downarrow_{ap}$ and $(\!|-|\!)_{ap}$).

The proof of Theorem 8 is given in Appendix B.5.

Theorem 9 (Adequacy of $(\!|-|\!)$)**.** Let $\Gamma \vdash e : \tau$ be a well-typed Λ_S term. Then for any well-typed substitution of closed values $\bar{\gamma} \vDash \Gamma$, if $(\!|\Gamma \vdash e : \tau|\!)_{id}(\!|\bar{\gamma}|\!)_{id} = (\!|v|\!)_{id}$ for some value v, then $e[\bar{\gamma}/dom(\Gamma)] \Downarrow_{id} v$ (and similarly for $\Downarrow_{ap}$ and $(\!|-|\!)_{ap}$).

Details of the proof of Theorem 9 can be found in Appendix B.5.

Our main error backward error soundness theorem requires one final piece of information: we must know that the functors U_{id} and U_{ap} project directly from interpretations of **Bean** programs in **Bel** (Definition 31) to interpretations of Λ_S programs in **Set** (Definition 32):

Lemma 18 (Pairing)**.** Let $\Phi \mid \Gamma \vdash e : \sigma$ be a **Bean** program. Then we have

$$U_{id}[\![\Phi \mid \Gamma \vdash e : \sigma]\!] = (\!|\Phi, \Gamma^{\circ} \vdash e : \sigma|\!)_{id} \quad \text{and} \quad U_{ap}[\![\Phi \mid \Gamma \vdash e : \sigma]\!] = (\!|\Phi, \Gamma^{\circ} \vdash e : \sigma|\!)_{ap}.$$

A proof of Lemma 18 follows by induction on the structure of the **Bean** derivation $\Phi \mid \Gamma \vdash e : \sigma$; details of the proof can be found in Appendix B.5.

Theorem 10 (Backward Error Soundness)**.** Let

$$\Phi \mid x_1 :_{r_1} \sigma_1, \cdots, x_n :_{r_n} \sigma_n = \Gamma \vdash e : \sigma$$

be a well-typed **Bean** term. Then for any well-typed substitutions $\bar{p} \vDash \Phi$ and $\bar{k} \vDash \Gamma^\circ$, if

$$e[\bar{p}/dom(\Phi)][\bar{k}/dom(\Gamma)] \Downarrow_{ap} v$$

for some value v, then the well-typed substitution $\bar{l} \vDash \Gamma^\circ$ exists such that

$$e[\bar{p}/dom(\Phi)][\bar{l}/dom(\Gamma)] \Downarrow_{id} v,$$

and $d_{[\![\sigma_i]\!]}(k_i, l_i) \leq r_i$ for each $k_i \in \bar{k}$ and $l_i \in \bar{l}$.

Proof. We sketch the proof here; details are provided in Appendix B.6. The key idea is to use the backward map $b : ([\![\Phi]\!] \otimes [\![\Gamma]\!] \otimes [\![\sigma]\!]) \to ([\![\Phi]\!] \otimes [\![\Gamma]\!])$ to construct the well-typed substitutions $\bar{s}$ and $\bar{l}$ such that $(\bar{s}, \bar{l}) = b((\bar{p}, \bar{k}), v)$. From the second property of backward error lenses we then have

$$f(\!|(\bar{s}, \bar{l})|\!)_{id} = f(\!|b((\bar{p}, \bar{k}), v)|\!)_{id} = v.$$

We can use this result along with pairing (Lemma 18) and adequacy (Theorem 9) to show

$$e[\bar{s}/dom(\Phi)][\bar{l}/dom(\Gamma)] \Downarrow_{id} v.$$

By soundness (Theorem 8) we can then derive a backward error result:

$$\tilde{f}(\!|(\bar{p}, \bar{k})|\!)_{ap} = f(\!|(\bar{s}, \bar{l})|\!)_{id}.$$

Two things remain to be shown. First, we must show the values of discrete type carry no backward error, i.e., $\bar{s} = \bar{p}$. Second, we must show the values of linear type have bounded backward error. Both follow from the first property of error lenses: from the inequality

$$\max\left(d_{[\![\Phi]\!]}(\bar{p}, \bar{s}), d_{[\![\Gamma]\!]}(\bar{k}, \bar{l})\right) \leq d_{[\![\sigma]\!]}(v, v) \neq \infty$$

we can conclude $\bar{s} = \bar{p}$ and $d_{[\![\sigma_i]\!]}(k_i, l_i) \leq r_i$ for each $k_i \in \bar{k}$ and $l_i \in \bar{l}$. □

4.5 Example Bean Programs

We will present a range of case studies demonstrating how algorithms with well-known backward error bounds from the literature can be implemented in **Bean**. We begin by comparing two implementations of polynomial evaluation, a naive evaluation and Horner's scheme. Next, we write several programs which compose to perform generalized matrix-vector multiplication. Finally, we write a triangular linear solver.

To improve the readability of our examples, we adopt several conventions. First, matrices are assumed to be stored in row-major order. Second, following the convention used in the grammar for **Bean** in Section 4.2, we use x and y for linear variables and z for discrete variables. Finally, for types, we denote both discrete and linear numeric types by $\mathbb{R}$, and use a shorthand for type assignments of vectors and matrices. For instance: $\mathbb{R}^2 \equiv (\mathbb{R} \otimes \mathbb{R})$ and $\mathbb{R}^{3\times 2} \equiv (\mathbb{R} \otimes \mathbb{R}) \otimes (\mathbb{R} \otimes \mathbb{R}) \otimes (\mathbb{R} \otimes \mathbb{R})$.

Since **Bean** is a simple first-order language and currently does not support higher-order functions or variable-length tuples, programs can become verbose. To reduce code repetition, we use basic user-defined abbreviations in our examples.

Polynomial Evaluation

To illustrate how **Bean** can provide a fine-grained backward error analysis for numerical algorithms, we begin with simple programs for polynomial evaluation. The first program, $\mathrm{PolyVal}$, evaluates a polynomial by naively multiplying each coefficient by the variable multiple times and then summing the resulting terms. The second program, Horner, applies Horner's method, which iteratively adds the next coefficient and then multiplies the sum by the variable (Higham, 2002, p.94). We consider here **Bean** implementations of these algorithms for a second-order polynomial; in Section 4.6, we describe a prototype implementation of **Bean** and evaluate the backward error bounds it infers for higher-degree polynomials.

Given a tuple $\mathrm{a} : \mathbb{R}^3$ of coefficients and a discrete variable $\mathrm{z} : \mathbb{R}$, the **Bean** programs for evaluating a second-order polynomial $p(z) = a_0 + a_1 z + a_2 z^2$ using naive polynomial evaluation and Horner's method are shown below.

```
PolyVal a z :=
let (a0, a') = a  in
let (a1, a2) = a' in
let y1  = dmul z a1  in
let y2' = dmul z a2  in
let y2  = dmul z y2' in
let x   = add a0 y1  in
add x y2
```

```
Horner a z :=
let (a0, a') = a   in
let (a1, a2) = a'  in
let y1 = dmul z a2 in
let y2 = add a1 x  in
let y3 = dmul z y2 in
add a0 y3
```

Recall from Section 4.1 that the **dmul** operation assigns backward error onto its second argument; in the programs above, the operation indicates that backward error should not be assigned to the discrete variable z. Using **Bean**'s type system, the following typing judgments are valid:

$$\mathrm{z} : \mathbb{R} \mid \mathrm{a} :_{3\varepsilon} \mathbb{R}^3 \vdash \mathrm{PolyVal\ a\ z} : \mathbb{R} \qquad \mathrm{z} : \mathbb{R} \mid \mathrm{a} :_{4\varepsilon} \mathbb{R}^3 \vdash \mathrm{Horner\ a\ z} : \mathbb{R}$$

From these judgments, *backward error soundness* (Theorem 10) guarantees that PolyVal has backward error of at most 3ε with respect to each element in the tuple a, while Horner has backward error of at most 4ε with respect to each element in the tuple a.

Surprisingly, though Horner's scheme is considered more numerically stable as it minimizes the number of floating-point operations, we find it has potentially greater backward error with respect to the vector of coefficients. A closer look at each coefficient individually, however, reveals more information about the two implementations. By adjusting the implementations to take each coefficient as a separate input, we can derive the backward error bounds for each coefficient individually. Now, **Bean**'s type system derives the following valid judgments:

$$\mathrm{z} : \mathbb{R} \mid \mathrm{a0} :_{2\varepsilon} \mathbb{R}, \mathrm{a1} :_{3\varepsilon} \mathbb{R}, \mathrm{a2} :_{3\varepsilon} \mathbb{R} \vdash \mathrm{PolyVal'} : \mathbb{R} \qquad \mathrm{z} : \mathbb{R} \mid \mathrm{a0} :_{\varepsilon} \mathbb{R}, \mathrm{a1} :_{3\varepsilon} \mathbb{R}, \mathrm{a2} :_{4\varepsilon} \mathbb{R} \vdash \mathrm{Horner'} : \mathbb{R}$$

We see that Horner's scheme assigns more backward error onto the coefficients of higher-order terms than lower-order terms, while naive polynomial evaluation assigns the same error onto all but

the lowest-order coefficient. In this way, **Bean** can be used to investigate the numerical stability of different polynomial evaluation schemes by providing a fine-grained error analysis.

Matrix-Vector Multiplication

A key feature of **Bean**'s type and effect system is its ability to precisely track backward error across increasingly large programs. Here, we demonstrate this process with several programs that gradually build up to a scaled matrix-vector multiplication.

Given a matrix $\mathrm{M} \in \mathbb{R}^{m \times n}$, vectors $v \in \mathbb{R}^n$ and $u \in \mathbb{R}^m$, and constants $a, b \in \mathbb{R}$, a scaled matrix-vector operation computes $a \cdot (\mathrm{M} \cdot v) + b \cdot u$. Since **Bean** does not currently support variable-length tuples, we present the details of a **Bean** implementation for a 2×2 matrix.

We first define the program $\mathrm{SVecAdd}$, which computes a scalar-vector product using $\mathrm{ScaleVec}$ and then adds the result to another vector. Given a discrete variable $\mathrm{a} : \mathbb{R}$, along with linear variables $\mathrm{x} : \mathbb{R}^2$ and $\mathrm{y} : \mathbb{R}^2$, we implement these programs as follows:

```
ScaleVec a x :=
let (x0, x1) = x  in
let u = dmul a x0 in
let v = dmul a x1 in
(u, v)
```

```
SVecAdd a x y :=
let (x0, x1) = ScaleVec a x in
let (y0, y1) = y  in
let u = add x0 y0 in
let v = add x1 y1 in
(u, v)
```

These programs have the following valid typing judgments:

$$\mathrm{a} : \mathbb{R} \mid \mathrm{x} :_{\varepsilon} \mathbb{R}^2 \vdash \mathrm{ScaleVec\ a\ x} : \mathbb{R} \qquad \mathrm{a} : \mathbb{R} \mid \mathrm{x} :_{2\varepsilon} \mathbb{R}^2, \mathrm{y} :_{\varepsilon} \mathbb{R}^2 \vdash \mathrm{SVecAdd\ a\ x\ y} : \mathbb{R}$$

In the typing judgment for $\mathrm{SVecAdd}$, we observe that the linear variable x has a backward error bound of 2ε, while the linear variable y has backward error bound of only ε. This difference arises because x accumulates ε backward error from $\mathrm{ScaleVec}$ and an additional ε backward error from the vector addition with the linear variable y.

Now, given discrete variables $\mathrm{a} : \mathbb{R}$ and $\mathrm{b} : \mathbb{R}$, and $\mathrm{v} : \mathbb{R}^2$, along with the linear variables $\mathrm{M} : \mathbb{R}^{2\times 2}$ and $\mathrm{u} : \mathbb{R}^2$, we can compute a matrix-vector product of M and v with $\mathrm{MatVecMul}$, and use the result in the scaled matrix-vector product, $\mathrm{SMatVecMul}$:

```
MatVecMul M v :=
let (m0, m1) = M in
let u0 = InnerProduct m0 v in
let u1 = InnerProduct m1 v in
(u0, u1)
```

```
SMatVecMul M v u a b :=
let x = MatVecMul M v in
let y = ScaleVec b u  in
SVecAdd a x y
```

For $\mathrm{MatVecMul}$, we rely on a program $\mathrm{InnerProduct}$, which computes the dot product of two 2×2 vectors. Notably, $\mathrm{InnerProduct}$ differs from the $\mathrm{DotProd2}$ program described in Section 4.1 because it assigns backward error only onto the first vector. The type of this program is:

$$\mathrm{v} : \mathbb{R}^2 \mid \mathrm{u} :_{2\varepsilon} \mathbb{R}^2 \vdash \mathrm{InnerProduct\ u\ v} : \mathbb{R}$$

The **Bean** programs $\mathrm{MatVecMul}$ and $\mathrm{SMatVecMul}$ have the following valid typing judgments:

$$\mathrm{v} : \mathbb{R}^2 \mid \mathrm{M} :_{2\varepsilon} \mathbb{R}^{2\times 2} \vdash \mathrm{MatVecMul\ M\ v} : \mathbb{R}^2$$

$$\mathrm{a} : \mathbb{R}, \mathrm{b} : \mathbb{R}, \mathrm{v} : \mathbb{R}^2 \mid \mathrm{M} :_{4\varepsilon} \mathbb{R}^{2\times 2}, \mathrm{u} :_{2\varepsilon} \mathbb{R}^2 \vdash \mathrm{SMatVecMul\ M\ v\ u\ a\ b} : \mathbb{R}^2$$

By error soundness, these judgments say that the computation $\mathrm{SMatVecMul}$ produces at most 2ε backward error with respect to the vector u and at most 4ε backward error with respect to the matrix M. The backward error bound for M can be understood as follows: the computation $\mathrm{MatVecMul\ M\ v}$ assigns at most 2ε backward error to M, and the computation $\mathrm{SVecAdd\ a\ x\ y}$ assigns an additional 2ε backward error to M, resulting in a backward error bound of 4ε. Similarly, the backward error bound of 2ε for the variable u arises from the computation $\mathrm{ScaleVec\ b\ u}$, which assigns at most ε backward error to u, and $\mathrm{SVecAdd\ a\ x\ y}$, which assigns at most an additional ε backward error to u, leading to a total backward error bound of 2ε. In Section 4.6.2, we will see that the backward error bounds for matrix-vector multiplication derived by **Bean** match the worst-case theoretical backward error bounds given in the literature.

Overall, these examples highlight the compositional nature of **Bean**'s analysis: like all type systems, the type of a **Bean** program is derived from the types of its subprograms. While the numerical analysis literature is unclear on whether (and when) backward error analysis can be performed compositionally (e.g., (Bornemann, 2007)), **Bean** demonstrates that this is in fact possible.

Triangular Linear Solver

One of the benefits of integrating error analysis with a type system is the ability to weave common programming language features, such as conditionals (if-statements) and error-trapping, into the analysis. We demonstrate these features in our final, and most complex example: a linear solver for triangular matrices. Given a lower triangular matrix $A \in \mathbb{R}^{2\times2}$ and a vector $b \in \mathbb{R}^2$, the linear solver should compute return a vector x satisfying $Ax = b$ if there is a unique solution.

We comment briefly on the program $\mathrm{LinSolve}$, shown below. The matrix and vector are given as inputs $((\mathrm{a00}, \mathrm{a01}), (\mathrm{a10}, \mathrm{a11})) : \mathbb{R}^{2\times2}$ where $\mathrm{a01}$ is assumed to be 0, and $(\mathrm{b0}, \mathrm{b1}) : \mathbb{R}^2$. The program either returns the solution x as a vector, or returns error if the linear system does not have a unique solution. The div operator has return type $\mathbb{R} + \mathbf{err}$, where **err** represents division by zero. Ensuing computations can check if the division succeeded using case expressions. This example also uses the !-constructor to convert a linear variable into a discrete one; this is required since the later entries in the vector x depend on—i.e., require duplicating—earlier entries in the vector.

```
LinSolve ((a00, a01), (a10, a11)) (b0, b1) :=
let x0_or_err = div b0 a00 in // solve for x0 = b0 / a00
case x0_or_err of
  inl (x0) => // if div succeeded
    dlet d_x0 = !x0 in // make x0 discrete for reuse
    let s0 = dmul d_x0 a10 in // s0 = x0 * a10
    let s1 = sub b1 s0 in // s1 = b1 − x0 * a10
    let x1_or_err = div s1 a11 in // solve for x1 = (b1 − x0 * a10) / a11
    case x1_or_err of
      inl (x1) => inl (d_x0, x1) // return (x0, x1)
    | inr (err) => inr err // division by 0
| inr (err) => inr err // division by 0
```

The type of $\mathrm{LinSolve}$ is $\mathrm{A} :_{5\varepsilon/2} \mathbb{R}^{2\times 2}, b :_{3\varepsilon/2} \mathbb{R}^2 \vdash \mathrm{LinSolve}\ \mathrm{A}\ \mathrm{b} : \mathbb{R}^2 + \mathbf{err}$. Hence, $\mathrm{LinSolve}$ has a guaranteed backward error bound of at most $5\varepsilon/2$ with respect to the matrix M and at most $3\varepsilon/2$ with respect to the vector b. If either of the division operations fail, the program returns **err**. This example demonstrates how various features in **Bean** combine to establish backward error guarantees for programs involving control flow and duplication, via careful control of how to assign and accumulate backward error through the program.

4.6 Implementation and Evaluation

4.6.1 Implementation

We implemented a type checking and coeffect inference algorithm for **Bean** in OCaml. It is based on the sensitivity inference algorithm introduced by de Amorim et al. (2014), which is used in implementations of *Fuzz*-like languages Gaboardi et al. (2013); Kellison and Hsu (2024). Given a **Bean** program without any error bound annotations in the context, the type checker ensures the program is well-formed, outputs its type, and infers the tightest possible backward error bound on each input variable. Using the type checker, users can write large **Bean** programs and automatically infer backward error with respect to each variable.

More precisely, let $\Gamma^\bullet$ denote a context *skeleton*, a linear typing context with no coeffect annotations. If Γ is a linear context, let $\overline{\Gamma}$ denote its skeleton. Next, we say Γ_1 is a *subcontext* of Γ_2, $\Gamma_1 \sqsubseteq \Gamma_2$, if $\mathrm{dom}\,\Gamma_1 \subseteq \mathrm{dom}\,\Gamma_2$ and for all $x :_r \sigma \in \Gamma_1$, we have $x :_q \sigma \in \Gamma_2$ where $r \leq q$. In other words, x has a tighter backward error bound in the subcontext. Now, we can say the input to the type checking algorithm is a typing context skeleton $\Phi \mid \Gamma^\bullet$ and a **Bean** program, e. The output is the type of the program σ and a linear context Γ such that $\Phi \mid \Gamma \vdash e : \sigma$ and $\overline{\Gamma} \sqsubseteq \Gamma^\bullet$. Calls to the algorithm are written as $\Phi \mid \Gamma^\bullet; e \Rightarrow \Gamma; \sigma$. The algorithm uses a recursive, bottom-up approach to build the final context.

Table 4.1: A comparison of **Bean** to Fu et al. (2015) on polynomial approximations of sin and cos. The **Bean** implementation matches the programs evaluated by Fu et al. (2015) for the given range of input values.

Benchmark	Range	Backward Bound		Timing (ms)	
		Bean	Fu et al. (2015)	**Bean**	Fu et al. (2015)
cos	[0.0001, 0.01]	1.33e-15	5.43e-09	1	1310
sin	[0.0001, 0.01]	1.44e-15	1.10e-16	1	1280

For example, to type the **Bean** program (e, f), where e and f are themselves programs, we use the algorithm rule

$$\frac{\Phi \mid \Gamma^\bullet; e \Rightarrow \Gamma_1; \sigma \qquad \Phi \mid \Gamma^\bullet; f \Rightarrow \Gamma_2; \tau \qquad \operatorname{dom}\Gamma_1 \cap \operatorname{dom}\Gamma_2 = \emptyset}{\Phi \mid \Gamma^\bullet; (e, f) \Rightarrow \Gamma_1, \Gamma_2; \sigma \otimes \tau} (\otimes \text{ I})$$

In practice, this means recursively calling the algorithm on e and f then combining their outputted contexts. The output contexts discard unused variables from the input skeletons; thus, the requirement $\operatorname{dom}\Gamma_1 \cap \operatorname{dom}\Gamma_2 = \emptyset$ ensures the strict linearity requirement is met.

The type checking algorithm is sound and complete, meaning that it agrees exactly with **Bean**'s typing rules. Precisely:

Theorem 11 (Algorithmic Soundness)**.** If $\Phi \mid \Gamma^\bullet; e \Rightarrow \Gamma; \sigma$, then $\bar{\Gamma} \sqsubseteq \Gamma^\bullet$ and the derivation $\Phi \mid \Gamma \vdash e : \sigma$ exists.

Theorem 12 (Algorithmic Completeness)**.** If $\Phi \mid \Gamma \vdash e : \sigma$ is a valid derivation in **Bean**, then there exists a context $\Delta \sqsubseteq \Gamma$ such that $\Phi \mid \bar{\Gamma}; e \Rightarrow \Delta; \sigma$.

The full algorithm and proofs of its correctness are given in Appendix B.7. The **Bean** implementation is parametrized only by unit roundoff, which is dependent on the floating-point format and rounding mode and is fixed for a given analysis.

Table 4.2: The performance of **Bean** benchmarks with known backward error bounds from the literature. The Input Size column gives the length of the input vector or dimensions of the input matrix; the Ops column gives the total number of floating-point operations. The Backward Bound column reports the bounds inferred by **Bean** and well as the standard bounds (Std.) from the literature. The Timing column reports the time in seconds for **Bean** to infer the backward error bound.

Benchmark	Input Size	Ops	Backward Bound		Timing (s)
			Bean	Std.	
DotProd	20	39	2.22e-15	2.22e-15	0.004
	50	99	5.55e-15	5.55e-15	0.04
	100	199	1.11e-14	1.11e-14	0.3
	500	999	5.55e-14	5.55e-14	30
Horner	20	40	4.44e-15	4.44e-15	0.002
	50	100	1.11e-14	1.11e-14	0.02
	100	200	2.22e-14	2.22e-14	0.1
	500	1000	1.11e-13	1.11e-13	10
PolyVal	10	65	1.22e-15	1.22e-15	0.004
	20	230	2.33e-15	2.33e-15	0.06
	50	1325	5.66e-15	5.66e-15	5
	100	5150	1.12e-14	1.12e-14	200
MatVecMul	5×5	45	5.55e-16	5.55e-16	0.003
	10×10	190	1.11e-15	1.11e-15	0.1
	20×20	780	2.22e-15	2.22e-15	6
	50×50	4950	5.55e-15	5.55e-15	1000
Sum	50	49	5.44e-15	5.44e-15	0.008
	100	100	1.10e-14	1.10e-14	0.04
	500	499	5.54e-14	5.54e-14	4
	1000	999	1.11e-13	1.11e-13	30

Table 4.3: A comparison of forward bounds derived from **Bean**'s backward error bounds to those of NumFuzz and Gappa. For Gappa, we assume all variables are in the interval $[0.1, 1000]$.

Benchmark	Input Size	Ops	Forward Bound		
			Bean	NumFuzz	Gappa
Sum	500	499	1.11e-13	1.11e-13	1.11e-13
DotProd	500	999	1.11e-13	1.11e-13	1.11e-13
Horner	500	1000	2.22e-13	2.22e-13	2.22e-13
PolyVal	100	5150	2.24e-14	2.24e-14	2.24e-14

4.6.2 Evaluation

In this section, we report results from an empirical evaluation of our **Bean** implementation, focusing primarily on the quality of the inferred bounds. Since **Bean** is the first tool to statically derive *sound* backward error bounds, a direct comparison with existing tools is challenging. We therefore evaluate the inferred bounds using three complementary methods.

First, we compare our results to those from a dynamic analysis tool for automated backward error analysis introduced by Fu et al. (2015). To our knowledge, the results reported by Fu et al. (2015) provide the only automatically derived quantitative bounds on backward error available for comparison; these results serve as a useful baseline for assessing the tightness of the bounds inferred by **Bean**. However, the experimental results reported by Fu et al. (2015) are limited to transcendental functions, while **Bean** is designed to handle larger programs oriented towards linear algebra primitives. Therefore, we also include an evaluation against theoretical worst-case backward error bounds described in the literature. This allows us to benchmark **Bean**'s bounds in relation to established theoretical limits, providing a measure of how closely **Bean**'s inferred bounds approach these worst-case values. Finally, we evaluate the quality of the backward error bounds derived by **Bean** using forward error as a proxy. Specifically, using known values of the relative componentwise condition number (Definition 33), we compute forward error bounds from our backward error bounds. This approach enables a comparison to existing tools focused on forward error analysis. We compare our derived forward error bounds to those produced by two tools that soundly and automatically bound relative forward error: NumFuzz Kellison and Hsu (2024) and Gappa Daumas and Melquiond (2010). Both tools are capable of scaling to larger benchmarks involving over 100 floating-point operations, making them suitable tools for comparison with **Bean**. All of our experiments were performed on a MacBook Pro with an Apple M3 processor and 16 GB of memory.

Comparison to Dynamic Analysis

The results for the comparison of **Bean** to the optimization based tool for automated backward error analysis due to Fu et al. (2015) is given in Table 4.1. The benchmarks are polynomial approximations of sin and cos implemented using Taylor series expansions following the GNU C Library (glibc) version 2.21 implementations. Our **Bean** implementations match the benchmarks from Table 4.1 on the input range $[0.0001, 0.01]$. Specifically, the Taylor series expansions implemented in **Bean** only match the glibc implementations for inputs in this range. Since the glibc implementations analyzed by Fu et al. (2015) use double-precision and round-to-nearest, we instantiated **Bean** with a unit roundoff of $u = 2^{-53}$. Although we include timing information for reference, the implementation described by Fu et al. is neither publicly available nor maintained, preventing direct runtime comparisons; thus, all values are taken from Table 6 of Fu et al. (2015).

Evaluation Against Theoretical Worst-Case Bounds

Table 4.2 presents results for several benchmark problems with known backward error bounds from the literature. Each benchmark was run on inputs of increasing size (given in Input Size), with the total number of floating-point operations listed in the Ops column. The Std. column provides the worst-case theoretical backward error bound reported in the literature assuming double-precision and round-to-nearest; the relevant references are (Higham, 2002, p.63, p.94, p.82). For simplicity, the **Bean** programs are written with a single linear variable, while the remaining inputs are treated as discrete variables. The maximum elementwise backward bound is computed with respect to the linear input. The **Bean** programs emulate the following analyses for input size N:

- $\mathrm{DotProd}$ computes the dot product of two vectors in $\mathbb{R}^N$, assigning backward error to a single vector.
- Horner evaluates an N-degree polynomial using Horner's scheme, assigning backward error

onto the vector of coefficients.

- $\mathrm{PolyVal}$ naively evaluates an N-degree polynomial, assigning backward error onto the vector of coefficients.
- $\mathrm{MatVecMul}$ computes the product of a matrix in $\mathbb{R}^{N \times N}$ and a vector in $\mathbb{R}^{N}$, assigning backward error onto the matrix.
- Sum sums the elements of a vector in $\mathbb{R}^{N}$, assigning backward error onto the vector.

Since we report the backward error bounds from the literature under the assumption of double-precision and round-to-nearest, we instantiated **Bean** with a unit roundoff of $u = 2^{-53}$.

Using Forward Error as a Proxy

We can compare the quality of the backward error bounds derived by **Bean** to existing tools using forward error as a proxy. Specifically, by using known values of the relative componentwise condition number κ_{rel}, we can compute relative forward error bounds from relative backward error bounds (Hohmann and Deuflhard, 2003, Definition 2.12):

Definition 33 (Relative Componentwise Condition Number)**.** The *relative componentwise condition number* of a scalar function $f : \mathbb{R}^n \to \mathbb{R}$ is the smallest number $\kappa_{rel} \geq 0$ such that, for all $x \in \mathbb{R}^n$,

$$d_{\mathbb{R}}(f(x), \tilde{f}(x)) \leq \kappa_{rel} \max_i d_{\mathbb{R}}(x_i, \tilde{x}_i) \tag{4.38}$$

where $\tilde{f}$ is the approximating program and $\tilde{x}$ is the perturbed input witnessing $f(\tilde{x}) = \tilde{f}(x)$. In Equation (4.38), $d_{\mathbb{R}}(f(x), \tilde{f}(x))$ is the *relative forward error*, and $\max_i d_{\mathbb{R}}(x_i, \tilde{x}_i)$ is the maximum *relative backward error*. Thus, for problems where the relative condition number is known, we can compute relative forward error bounds from the relative backward error bounds inferred by **Bean**.

Table 4.3 presents the results for several benchmark problems with $\kappa_{rel} = 1$. For these problems, according to Equation (4.38), the maximum relative backward error serves as an upper bound on the

relative forward error. As an example, the problem of summing n values $(a_i)_{1\leq i\leq n}$ has a relative condition number $\kappa_{rel} = \sum_{i=1}^{n} |a_i| / |\sum_{i=1}^{n} a_i|$ Muller et al. (2018), which clearly reduces to $\kappa_{rel} = 1$ when all $a_i > 0$. In fact, for each of the benchmarks listed in Table 4.3, $\kappa_{rel} = 1$ is only guaranteed for strictly positive inputs. This assumption is already required for NumFuzz in order to guarantee the soundness of its forward error bounds. To enforce this in Gappa, we used an interval of $[0.1, 1000]$ for each input. Since NumFuzz assumes double-precision and round towards positive infinity, we instantiated **Bean** and Gappa with a unit roundoff of $u = 2^{-52}$.

Evaluation Summary

The main conclusions from our evaluation results are as follows. ***Bean's backward error bounds are useful***: In all of our experiments, **Bean** produced competitive error bounds. Compared to the backward error bounds reported by Fu et al. (2015) for their dynamic backward error analysis tool, **Bean** was able to derive *sound* backward error bounds that were close to or better than those produced by the dynamic tool. Furthermore, **Bean**'s sound bounds precisely match the worst-case theoretical backward error bounds from the literature, demonstrating that our approach guarantees soundness without being overly conservative. Finally, when using forward error as a proxy to assess the quality of **Bean**'s backward error bounds, we find that **Bean**'s bounds again precisely match the bounds produced by NumFuzz and Gappa. ***Bean performs well on large programs***: In our comparison to worst-case theoretical error bounds, we find that **Bean** takes under a minute to infer backward error bounds on benchmarks with fewer than 1000 floating-point operations. Overall, **Bean**'s performance scales linearly with the number of floating-point operations in a benchmark.

4.7 Related Work

Automated Backward Error Analysis

Existing automated methods for backward error analysis are based on automatic differentiation and optimization techniques. Unlike **Bean**, existing methods do not provide a soundness guarantee and are based on heuristics. Miller's algorithm (Miller and Spooner, 1978) first appeared in a FORTRAN package and used automatic differentiation to compute partial derivatives of function outputs with respect to function inputs as a proxy for backward error. The algorithm was later augmented to handle a broader range of program features (loops and conditional expressions) in a MATLAB implementation (Gáti, 2012).

The first optimization based tool for automated backward error analysis was introduced by Fu et al. (2015). The key idea of the approach is to separate the analysis into a *local* error analysis and a *global* error analysis. Given a program and specific inputs, the local error simulates the ideal, continuous problem by lifting the program to a higher-precision version. Then, a generic minimizer is used to derive a backward error function that associates the input and output of the original program to an input of the higher-precision program that hits the same output. The global error analysis uses the backward error function as a black-box function to heuristically estimate the maximal backward error for a range of inputs by Markov Chain Monte Carlo techniques.

Bean is similar to the optimization technique due to Fu et al. (2015) by virtue of the direct construction of the backward function: in order to perform a backward error analysis, both **Bean** and the optimization technique require an ideal function, an approximating function, and an explicit backward function. However, unlike **Bean**, the existing optimization method must perform a sometimes costly analysis to construct the ideal and backward functions for every program. In **Bean**, it is not necessary to construct these functions for typechecking since they are built into our semantic model.

Residual Based Methods

When a backward error bound does exist, it is possible to compute *a posteriori* estimates of the error dynamically using *residual-based methods* as described by Corless and Fillion (2013). These methods require constructing a *defining function*, which depends on both the input and the output of the program—similar to the backward maps in **Bean**. Whereas residual-based methods require the manual construction of a defining function, in **Bean**, the backward maps of complex programs are composed from their individual components, enabling more automated reasoning. Moreover, residual-based estimates are not *bounds* and, unlike **Bean**, do not provide a guaranteed sound overapproximation of the true error. In situations where soundness is not required, residual-based methods can be manually incorporated into programs. Unfortunately, in some instances, computing these estimates can be more computationally expensive than solving the original problem.

Type Systems and Formal Methods

A diverse set of tools for reasoning about forward rounding error bounds have been proposed in the formal methods literature; these tools are discussed in Section 3.8. In comparison, for backward error analysis, the formal methods literature is sparse. The LAProof library due to Kellison et al. (2023) provides formal proofs of backward error bounds for basic linear algebra subprograms in the Coq proof assistant, and gives an example of how these proofs can be used to verify real C-programs. These proofs are parametric in both the floating-point format and the size of the underlying data structures. In **Bean**, as with other type-based approaches, we trade some of the expressivity offered by proof assistant-based methods for a more lightweight and potentially more automated system. While less expressive, valid typing derivations in **Bean** correspond to formal proofs that a given program satisfies the backward error bound the type system assigns it. This guarantee is rigorously established by our backward error soundness theorem, Theorem 10.

The only other type-based approach to rounding error analysis is **NumFuzz**, which, like **Bean**,

also uses a linear type system and coeffects. However, **NumFuzz** is specifically designed for forward error analysis, whereas **Bean** focuses on backward error analysis. While the syntactic similarities between **NumFuzz** and **Bean** may suggest that **Bean** is simply a derivative of **NumFuzz** modified for backward error analysis, this is not the case. We designed **Bean** by first developing the category **Bel** of backward error lenses, and then developing the language described in Section 4.2 to fit this category. Indeed, the semantics of the two systems are entirely different:

- The primary semantic novelty in **NumFuzz** is the neighborhood monad, which tracks forward error but cannot be adapted for backward error. **Bean** does not use the neighborhood monad, or any monad at all.
- While both **NumFuzz** and **Bean** use a graded comonad, their interpretations are different and are used for different purposes. The graded comonad in **NumFuzz** scales the metric to track function sensitivity, while the graded comonad in **Bean** shifts (translates) the metric to track backward error.
- Similar to other *Fuzz*-like languages, **NumFuzz** interprets programs in the category of metric spaces, which lacks the necessary structure for reasoning about backward error. To address this, we introduced the novel category of backward error lenses, offering a completely new semantic foundation that distinguishes **Bean** from all languages in the *Fuzz* family.

Linear Type Systems and Coeffects

After **NumFuzz**, which we have already discussed, **Bean** is most closely related to coeffect-based type systems, like those described by Petricek et al. (2014); references can be found in the brief description of coeffects given in Section 2.2.2. As discussed before, a notable difference between **Bean**'s type system and other coeffect systems is that our model does not support contraction, and so our type system enforces strict linearity for variables with coeffect annotations, with a separate context for variables that can be reused. Our dual context approach is similar to the

Linear/Non-Linear (LNL) calculus described by Benton (1994).

Lenses and Bidirectional Programming Languages

Our semantic model is inspired by work on lenses, proposed by Foster et al. (2007), as a tool to address the view-update problem in databases. Basically, a lens is a pair of a forward transformation *get* and a backward transformation *put* which are used to synchronize related data. In general, lenses satisfy several *lens laws*, which can be framed as equations that specify the relationship between the lens transformations and the data they operate on; the equations defining *well-behaved lenses* are given in Equation (4.11) and Equation (4.12). These equations correspond closely to the properties of backward error lenses (Definition 28). The concept of a lens has been rediscovered multiple times in different contexts, ranging from categorical proof theory and Gödel's Dialectica translation (de Paiva, 1991) to more recent work on open games (Ghani et al., 2018), and supervised learning (Fong et al., 2019); the interested reader can see Hedges (2018) for a good summary. While the formal similarity between our backward error lenses and existing work on lenses is undeniable, we are not aware of any existing notion of lens that includes ours.

4.8 Conclusion

Bean is a typed first-order programming language that guarantees backward error bounds. Its type system is based on the combination of three elements: a notion of distances for types, a coeffect system for tracking backward error, and a linear type system for controlling how backward error can flow though programs. Although the backward error analysis modeled by **Bean** is more general than the standard approach, we can capture the standard definition as a special case, as shown by our main theorem of backward error soundness (Theorem 10). A major benefit of our proposed approach is that it is structured around the idea of composition: when backward error bounds exist,

the backward error bounds of complex programs are composed from the backward error bounds of their subprograms. The linear type system of bean correctly rejects programs that do not have bounded backward error, and is also flexible enough to capture the backward error analysis of well-known algorithms from the literature. **Bean** is the first demonstration of a static analysis framework for reasoning about backward error, and can be extended in various ways. We conclude this chapter with a discussion of promising directions for future development.

Implementation

The linear fragment of **Bean** resembles a first-order version of **NumFuzz**, and fully automated type checking and grade inference algorithms developed for the *Fuzz* family of languages, such as those described by Kellison and Hsu (2024),D'Antoni et al. (2013), and de Amorim et al. (2014) can be adapted for **Bean**. The main difference between the inference algorithms designed for *Fuzz*-like languages and **Bean** is strict linearity and dual contexts, which introduces a small but manageable complication to existing implementations. Similar to the inference algorithm described in Section 3.7, the general idea is that given three inputs—a **Bean** term e, a linear context $x_1 : \sigma_1, \ldots, x_i : \sigma_i = \Gamma^\circ$ with all grade annotations erased, and a discrete context Φ—the algorithm infers a type τ of e and the backward error bounds δ_i such that $\Phi \mid x_1 :_{\delta_1} \sigma_1, \ldots, x_i :_{\delta_i} \sigma_i \vdash e : \tau$. While using **Bean** in practice would require users to understand a linear type system and select operations based on whether a variable is linear or discrete (i.e., the choice of using `dmul` or `mul`), this is standard practice in languages with linear types. From a numerical standpoint, it is also often known which variables will have backward error assigned to them during the analysis, and the primary concern is computing a backward error bound for compositions of programs; **Bean** is well-suited to this task.

Additional Language Features

Bean does not support higher-order functions, limiting code reuse. Technically, we do not know if the **Bel** category supports linear exponentials, which would be needed to interpret function types. While most lens categories do not support higher-order functions, there are some notable situations where the lens category is symmetric monoidal closed (e.g., de Paiva (1991)). Connecting our work to these lens categories could suggest how to support higher-order functions in our framework.

BIBLIOGRAPHY

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

S. Abramsky and N. Tzevelekos. 2011. Introduction to Categories and Categorical Logic. In *New Structures for Physics*, Bob Coecke (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 3–94. https://doi.org/10.1007/978-3-642-12821-9_1

Mehrdad Aliasgari, Marina Blanton, Yihua Zhang, and Aaron Steele. 2013. Secure Computation on Floating Point Numbers. In *20th Annual Network and Distributed System Security Symposium, NDSS* (2013, San Diego, California, USA, February 24-27, 2013). The Internet Society.

E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. 1999. *LAPACK Users' Guide* (third ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA.

Andrew W. Appel and Ariel E. Kellison. 2024. VCFloat2: Floating-Point Error Analysis in Coq. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs* (London, UK) *(CPP 2024)*. Association for Computing Machinery, New York, NY, USA, 14–29. https://doi.org/10.1145/3636501.3636953

Robert Atkey. 2009. Parameterised Notions of Computation. *Journal of Functional Programming* 19, 3-4 (2009), 335–376. https://doi.org/10.1017/S095679680900728X

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

Arthur Azevedo de Amorim, Marco Gaboardi, Justin Hsu, Shin-ya Katsumata, and Ikram Cherigui. 2017. A semantic account of metric preservation. *SIGPLAN Not.* 52, 1 (Jan 2017), 545–556. https://doi.org/10.1145/3093333.3009890

P. N. Benton. 1994. A Mixed Linear and Non-Linear Logic: Proofs, Terms and Models (Extended Abstract). In *Selected Papers from the 8th International Workshop on Computer Science Logic (CSL '94)*. Springer-Verlag, Berlin, Heidelberg, 121–135.

L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. 2002. An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Software* 28, 2 (2002), 135–151.

Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. 2008. Boomerang: resourceful lenses for string data. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '08)*. Association for Computing Machinery, New York, NY, USA, 407–419. https://doi.org/10.1145/1328438.1328487

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

Sylvie Boldo and Guillaume Melquiond. 2011. Flocq: A Unified Library for Proving Floating-Point Algorithms in Coq. In *Proceedings of the 2011 IEEE 20th Symposium on Computer Arithmetic (ARITH '11)*. IEEE Computer Society, USA, 243–252. https://doi.org/10.1109/ARITH.2011.40

Sylvie Boldo and Guillaume Melquiond. 2017. *Computer Arithmetic and Formal Proofs*. ISTE Press - Elsevier. 326 pages. https://inria.hal.science/hal-01632617

Folkmar Bornemann. 2007. A Model for Understanding Numerical Stability. *IMA J. Numer. Anal.* 27, 2 (04 2007), 219–231. https://doi.org/10.1093/imanum/drl037

Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. 2014. A Core Quantitative Coeffect Calculus. In *Programming Languages and Systems*, Zhong Shao (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 351–370.

Alexandre Chapoutot. 2010. Interval Slopes as a Numerical Abstract Domain for Floating-Point Variables. In *Static Analysis*, Radhia Cousot and Matthieu Martel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 184–200.

Alexandre Chapoutot and Matthieu Martel. 2009. Abstract Simulation: A Static Analysis of Simulink Models. In *2009 International Conference on Embedded Software and Systems*. 83–92. https://doi.org/10.1109/ICESS.2009.80

Konstantinos Chatzikokolakis, Daniel Gebler, Catuscia Palamidessi, and Lili Xu. 2014. Generalized Bisimulation Metrics. In *CONCUR 2014 – Concurrency Theory*, Paolo Baldan and Daniele Gorla (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 32–46.

Liqian Chen, Antoine Miné, and Patrick Cousot. 2008. A Sound Floating-Point Polyhedra Abstract Domain. In *Programming Languages and Systems*, G. Ramalingam (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 3–18.

Liqian Chen, Antoine Miné, Ji Wang, and Patrick Cousot. 2009. Interval Polyhedra: An Abstract Domain to Infer Interval Linear Relationships. In *Proceedings of the 16th International Symposium on Static Analysis* (Los Angeles, CA) *(SAS '09)*. Springer-Verlag, Berlin, Heidelberg, 309–325. https://doi.org/10.1007/978-3-642-03237-0_21

Liqian Chen, Antoine Miné, Ji Wang, and Patrick Cousot. 2010. An Abstract Domain to Discover Interval Linear Equalities. In *11th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'10) (LNCS, Vol. 5944)*. Springer, Spain, 112–128. https://hal.science/hal-00531563

Yuanfeng Chen, Gaofeng Huang, Junjie Shi, Xiang Xie, and Yilin Yan. 2020. Rosetta: A Privacy-Preserving Framework Based on TensorFlow. https://github.com/LatticeX-Foundation/Rosetta.

Michael P. Connolly, Nicholas J. Higham, and Theo Mary. 2021. Stochastic Rounding and Its

Probabilistic Backward Error Analysis. *SIAM Journal on Scientific Computing* 43, 1 (2021), A566–A585. https://doi.org/10.1137/20M1334796

George Constantinides, Fredrik Dahlqvist, Zvonimir Rakamarić, and Rocco Salvia. 2021. Rigorous Roundoff Error Analysis of Probabilistic Floating-Point Computations. In *Computer Aided Verification*, Alexandra Silva and K. Rustan M. Leino (Eds.). Cham: Springer International Publishing, Cham, 626–650.

Robert M. Corless and Nicolas Fillion. 2013. *A Graduate Introduction to Numerical Methods*. Springer New York, New York, NY, USA. https://doi.org/10.1007/978-1-4614-8453-0

Patrick Cousot, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2005. The ASTREÉ Analyzer. In *Programming Languages and Systems*, Mooly Sagiv (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 21–30.

Raphaelle Crubille and Ugo Dal Lago. 2015. Metric reasoning about λ-terms: The affine case. In *Proceedings of the 2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '15)*. IEEE Computer Society, USA, 633–644. https://doi.org/10.1109/LICS.2015.64

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

Ugo Dal Lago and Francesco Gavazzo. 2022a. Effectful program distancing. *Proc. ACM Program. Lang.* 6, POPL, Article 19 (Jan 2022), 30 pages. https://doi.org/10.1145/3498680

Ugo Dal Lago and Francesco Gavazzo. 2022b. A Relational Theory of Effects and Coeffects. *Proc. ACM Program. Lang.* 6, POPL, Article 31 (Jan 2022), 28 pages. https://doi.org/10.1145/3498692

Nasrine Damouche, Matthieu Martel, Pavel Panchekha, Chen Qiu, Alexander Sanchez-Stern, and Zachary Tatlock. 2017. Toward a Standard Benchmark Format and Suite for Floating-Point

Analysis. In *Proceedings of the 10th International Workshop on Numerical Software Verification (NSV 2017)*. Springer, 63–77.

Loris D'Antoni, Marco Gaboardi, Emilio Jesús Gallego Arias, Andreas Haeberlen, and Benjamin Pierce. 2013. Sensitivity Analysis Using Type-Based Constraints. In *Proceedings of the 1st Annual Workshop on Functional Programming Concepts in Domain-Specific Languages* (Boston, Massachusetts, USA) *(FPCDSL '13)*. Association for Computing Machinery, New York, NY, USA, 43–50. https://doi.org/10.1145/2505351.2505353

Eva Darulova, Anastasiia Izycheva, Fariha Nasir, Fabian Ritter, Heiko Becker, and Robert Bastian. 2018. Daisy - Framework for Analysis and Optimization of Numerical Programs (Tool Paper). In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*. https://api.semanticscholar.org/CorpusID:4800709

Eva Darulova and Viktor Kuncak. 2014. Sound Compilation of Reals. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 235–248. https://doi.org/10.1145/2535838.2535874

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

Arnab Das, Ian Briggs, Ganesh Gopalakrishnan, Sriram Krishnamoorthy, and Pavel Panchekha. 2020. Scalable yet rigorous floating-point error analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20)*. IEEE Press, New York, NY, USA, Article 51, 14 pages.

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

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

Arthur Azevedo de Amorim, Marco Gaboardi, Justin Hsu, and Shin-ya Katsumata. 2021. Probabilistic relational reasoning via metrics. In *Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science* (Vancouver, Canada) *(LICS '19)*. IEEE Press, New York, NY, USA, Article 40, 19 pages.

Luiz Henrique de Figueiredo and Jorge Stolfi. 2004. Affine Arithmetic: Concepts and Applications. *Numerical Algorithms* 37, 1 (2004), 147–158. https://doi.org/10.1023/B:NUMA.0000049462.70970.b6

Valeria de Paiva. 1991. *The Dialectica Categories*. Ph. D. Dissertation. University of Cambridge. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-213.pdf Computer Laboratory Technical Report 213.

Anastasiia Izycheva Eva Darulova and, Fariha Nasir, Fabian Ritter, Heiko Becker, and Robert Bastian. 2018. Daisy - Framework for Analysis and Optimization of Numerical Programs, (Tool Paper). In *tacas18 (Lecture Notes in Computer Science, Vol. 10805)*. Springer, 270–287. https://doi.org/10.1007/978-3-319-89960-2_15

Sebastian Fischer, ZhenJiang Hu, and Hugo Pacheco. 2015. The Essence of Bidirectional Programming. *Science China Information Sciences* 58, 5 (2015), 1–21. https://doi.org/10.1007/s11432-015-5316-8

Brendan Fong, David I. Spivak, and Rémy Tuyéras. 2019. Backprop as Functor: A Compositional Perspective on Supervised Learning. In *34th Annual ACM/IEEE Symposium on Logic in Computer*

Science, LICS 2019, Vancouver, BC, Canada, June 24-27, 2019*. IEEE, 1–13. https://doi.org/10.1109/LICS.2019.8785665

John Nathan Foster. 2009. *Bidirectional Programming Languages*. Ph. D. Dissertation. University of Pennsylvania.

J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 2007. Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-Update Problem. *ACM Trans. Program. Lang. Syst.* 29, 3 (May 2007), 17–es. https://doi.org/10.1145/1232420.1232424

Nate Foster, Kazutaka Matsuda, and Janis Voigtländer. 2012. Three Complementary Approaches to Bidirectional Programming. *Generic and Indexed Programming: International Spring School, SSGIP 2010, Oxford, UK, March 22-26, 2010, Revised Lectures* (2012), 1–46.

Martin Franz and Stefan Katzenbeisser. 2011. Processing Encrypted Floating Point Signals. In *Proceedings of the Thirteenth ACM Multimedia , Workshop on Multimedia and Security* (Buffalo, New York, USA) *(MM & Sec '11)*. Association for Computing Machinery, New York, NY, USA, 103–108. https://doi.org/10.1145/2037252.2037271

Zhoulai Fu, Zhaojun Bai, and Zhendong Su. 2015. Automated Backward Error Analysis for Numerical Code. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 639–654. https://doi.org/10.1145/2814270.2814317

S. Fujii, S. Katsumata, and P.-A. Melliés. 2016. Towards a Formal Theory of Graded Monads. In *Foundations of Software Science and Computation Structures (FOSSACS)*. Springer, 513–530. https://doi.org/10.1007/978-3-662-49630-5_30

Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C. Pierce. 2013. Linear Dependent Types for Differential Privacy. In *Proceedings of the 40th Annual ACM

SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Rome, Italy) *(POPL '13, Vol. 48)*. Association for Computing Machinery, New York, NY, USA, 357–370. https://doi.org/10.1145/2480359.2429113

Marco Gaboardi, Shin-ya Katsumata, Dominic Orchard, Flavien Breuvart, and Tarmo Uustalu. 2016. Combining effects and coeffects via grading. *SIGPLAN Not.* 51, 9 (Sep 2016), 476–489. https://doi.org/10.1145/3022670.2951939

Sanjam Garg, Abhishek Jain, Zhengzhong Jin, and Yinuo Zhang. 2022. Succinct Zero Knowledge for Floating Point Computations. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 1203–1216. https://doi.org/10.1145/3548606.3560653

Attila Gáti. 2012. Miller Analyzer for Matlab: A Matlab Package for Automatic Roundoff Analysis. *Computing and Informatics* 31, 4 (Oct. 2012), 713–726. https://www.cai.sk/ojs/index.php/cai/article/view/1101

Francesco Gavazzo. 2018. Quantitative Behavioural Reasoning for Higher-order Effectful Programs: Applicative Distances. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science* (Oxford, United Kingdom) *(LICS '18)*. Association for Computing Machinery, New York, NY, USA, 452–461. https://doi.org/10.1145/3209108.3209149

Francesco Gavazzo. 2019. *Coinductive Equivalences and Metrics for Higher-order Languages with Algebraic Effects*. Theses. Alma Mater Studiorum Università di Bologna. https://inria.hal.science/tel-02386201

Neil Ghani, Jules Hedges, Viktor Winschel, and Philipp Zahn. 2018. Compositional Game Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in , Computer Science* (Oxford, United Kingdom) *(LICS '18)*. Association for Computing Machinery, New York, NY, USA, 472–481. https://doi.org/10.1145/3209108.3209165

D. R. Ghica and A. I. Smith. 2014. Bounded Linear Types in a Resource Semiring. In *Proceedings of the 23rd European Symposium on Programming (ESOP 2014) (Lecture Notes in Computer Science, Vol. 8410)*. Springer, 331–350.

David K. Gifford and John M. Lucassen. 1986. Integrating functional and imperative programming. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming* (Cambridge, Massachusetts, USA) *(LFP '86)*. Association for Computing Machinery, New York, NY, USA, 28–38. https://doi.org/10.1145/319838.319848

Jean-Yves Girard. 1987. Linear Logic. *Theoretical Computer Science* 50 (1987), 1–102. https://doi.org/10.1016/0304-3975(87)90045-4

Jean-Yves Girard, Yves Lafont, and Paul Taylor. 1992. Bounded Linear Logic: A Modular Approach to Polynomial-Time Computability. In *Proceedings of the 7th Annual IEEE Symposium on Logic in Computer Science (LICS '92)*. IEEE Computer Society, 15–25. https://doi.org/10.1109/LICS.1992.185515

Eric Goubault and Sylvie Putot. 2011. Static Analysis of Finite Precision Computations. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, Ranjit Jhala and David Schmidt (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 232–247. https://doi.org/10.1007/978-3-642-18275-4_17

Leopold Haller, Alberto Griggio, Martin Brain, and Daniel Kroening. 2012. Deciding Floating-Point Logic with Systematic Abstraction. In *Formal Methods in Computer-Aided Design, FMCAD 2012, Cambridge, UK, October 22-25, 2012*, Gianpiero Cabodi and Satnam Singh (Eds.). IEEE, 131–140.

John Harrison. 1997a. Floating Point Verification in HOL Light: The Exponential Function. In *International Conference on Algebraic Methodology and Software, Technology (AMAST), Sydney, Australia (Lecture Notes in Computer Science, Vol. 1349)*. Springer, 246–260. https://doi.org/10.1007/BFb0000475

John Harrison. 1997b. Floating-Point Verification using Theorem Proving. In *14th International Conference on Theorem Proving in Higher Order Logics (TPHOLs) (LNCS, Vol. 1275)*. Springer, 3–17. https://doi.org/10.1007/BFb0028391

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

John Harrison. 2000. Formal Verification of Floating Point Trigonometric Functions. In *fmcad00 (Lecture Notes in Computer Science, Vol. 1954)*. Springer, 217–233. https://doi.org/10.1007/3-540-40922-X_14

Jules Hedges. 2018. Lenses for philosophers. https://julesh.com/2018/08/16/lenses-for-philosophers/

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

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

Martin Hofmann, Benjamin Pierce, and Daniel Wagner. 2011. Symmetric lenses. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 371–384. https://doi.org/10.1145/1926385.1926428

Andreas Hohmann and Peter Deuflhard. 2003. *Numerical Analysis in Modern Scientific Computing: An Introduction*. Vol. 43. Springer Science & Business Media.

Chun-Yi Hu, Nicholas M. Patrikalakis, and Xiuzi Ye. 1996. Robust interval solid modelling Part I: representations. *Computer-Aided Design* 28, 10 (1996), 807–817. https://doi.org/10.1016/0010-4485(96)00013-9

IEEE Computer Society. 2019. *IEEE Standard for Floating-Point Arithmetic*. Standard IEEE Std 754-2019. Institute of Electrical and Electronics Engineers. https://doi.org/10.1109/IEEESTD.2019.8766229

Ilse C. F. Ipsen and Hua Zhou. 2020. Probabilistic Error Analysis for Inner Products. *SIAM J. Matrix Anal. Appl.* 41, 4 (2020), 1726–1741. https://doi.org/10.1137/19M1270434 arXiv:https://doi.org/10.1137/19M1270434

Andrej Ivaskovic. 2023. *Programming and static analysis with graded monads*. Ph. D. Dissertation.

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

Bertrand Jeannet and Antoine Miné. 2009. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *Proceedings of the 21st International Conference on Computer Aided Verification* (Grenoble, France) *(CAV '09)*. Springer-Verlag, Berlin, Heidelberg, 661–667. https://doi.org/10.1007/978-3-642-02658-4_52

Michael Johnson, Robert Rosebrugh, and Richard Wood. 2010. Algebras and Update Strategies. *JUCS - Journal of Universal Computer Science* 16, 5 (2010), 729–748. https://doi.org/10.3217/jucs-016-05-0729 arXiv:https://doi.org/10.3217/jucs-016-05-0729

Michael Johnson, Robert Rosebrugh, and R. J. Wood. 2012. Lenses, fibrations and universal translations. *Mathematical Structures in Computer Science* 22, 1 (2012), 25–42. https://doi.org/10.1017/S0960129511000442

William Kahan. 1996. The Improbability of Probabilistic Error Analyses for Numerical Computations. (1996). Unpublished manuscript.

Liina Kamm and Jan Willemson. 2015. Secure floating point arithmetic and private satellite collision , analysis. *International Journal of Information Security* 14, 6 (2015), 531–548.

Shin-ya Katsumata. 2014. Parametric Effect Monads and Semantics of Effect Systems. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL 2014)*. Association for Computing Machinery, New York, NY, USA, 633–646. https://doi.org/10.1145/2535838.2535846

Shin-ya Katsumata. 2018. A Double Category Theoretic Analysis of Graded Linear Exponential Comonads. In *Foundations of Software Science and Computation Structures*, Christel Baier and Ugo Dal Lago (Eds.). Springer International Publishing, Cham, 110–127.

Ariel E. Kellison and Andrew W. Appel. 2022. Verified Numerical Methods for Ordinary Differential Equations. In *Software Verification and Formal Methods for ML-Enabled Autonomous Systems: 5th International Workshop, FoMLAS 2022, and 15th International Workshop, NSV 2022* (Haifa, Israel). Springer-Verlag, Berlin, Heidelberg, 147–163. https://doi.org/10.1007/978-3-031-21222-2_9

Ariel E. Kellison, Andrew W. Appel, Mohit Tekriwal, and David Bindel. 2023. LAProof: A Library of Formal Proofs of Accuracy and Correctness for Linear Algebra Programs. In *2023 IEEE 30th Symposium on Computer Arithmetic (ARITH)*. IEEE Computer Society, Los Alamitos, CA, USA, 36–43. https://doi.org/10.1109/ARITH58626.2023.00021

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

Hsiang-Shang Ko and Zhenjiang Hu. 2017. An axiomatic basis for bidirectional programming. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–29.

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

PaulBlain Levy, John Power, and Hayo Thielecke. 2003. Modelling environments in call-by-value programming languages. *Information and Computation* 185, 2 (2003), 182–210. https://doi.org/10.1016/S0890-5401(03)00088-9

Xiaoye S. Li, James W. Demmel, David H. Bailey, Greg Henry, Yozo Hida, Jimmy Iskandar, William Kahan, Suh Y. Kang, Anil Kapur, Michael C. Martin, Brandon J. Thompson, Teresa Tung, and Daniel J. Yoo. 2002. Design, implementation and testing of extended and mixed precision BLAS. *ACM Trans. Math. Softw.* 28, 2 (Jun 2002), 152–205. https://doi.org/10.1145/567806.567808

Joannes M Lucassen. 1987. *Types and effects: Towards the integration of functional and imperative programming*. Technical Report. MIT Laboratory for Computer Science.

J. M. Lucassen and D. K. Gifford. 1988. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '88)*. Association for Computing Machinery, New York, NY, USA, 47–57. https://doi.org/10.1145/73560.73564

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

Guillaume Melquiond Marc Daumas and. 2010. Certification of Bounds on Expressions Involving Rounded Operators. *ACM Trans. Math. Softw.* 37, 1 (2010), 2:1–2:20. https://doi.org/10.1145/1644001.1644003

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

Webb Miller and David Spooner. 1978. Algorithm 532: software for roundoff analysis [Z]. *ACM Trans. Math. Softw.* 4, 4 (dec 1978), 388–390. https://doi.org/10.1145/356502.356497

Antoine Miné. 2004. Relational abstract domains for the detection of floating-point run-time errors. In *European Symposium on Programming*. Springer, 3–17.

E. Moggi. 1989. Computational lambda-calculus and monads. In *Proceedings. Fourth Annual Symposium on Logic in Computer Science*. 14–23. https://doi.org/10.1109/LICS.1989.39155

Eugenio Moggi. 1991. Notions of computation and monads. *Information and Computation* 93, 1 (1991), 55–92. https://doi.org/10.1016/0890-5401(91)90052-4 Selections from 1989 IEEE Symposium on Logic in Computer Science.

Ramon E Moore, R Baker Kearfott, and Michael J Cloud. 2009. *Introduction to interval analysis*. SIAM.

Mariano M. Moscato, Laura Titolo, Marco A. Feliú, and César A. Muñoz. 2019. Provably Correct Floating-Point Implementation of a Point-in-Polygon Algorithm. In *Formal Methods – The Next 30 Years*, Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira (Eds.). Springer International Publishing, Cham, 21–37.

Jean-Michel Muller. 2016. *Elementary Functions: Algorithms and Implementation* (3rd ed.). Birkhäuser Basel. https://doi.org/10.1007/978-1-4899-7983-4

Jean-Michel Muller, Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, Mioara Joldes, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres. 2018. *Handbook of Floating-Point Arithmetic, 2nd edition*. Birkhäuser Boston. 632 pages. ACM G.1.0; G.1.2; G.4; B.2.0; B.2.4; F.2.1., ISBN 978-3-319-76525-9.

Alan Mycroft, Dominic Orchard, and Tomas Petricek. 2015. Effect Systems Revisited–Control-Flow Algebra and Semantics. In *Essays Dedicated to Hanne Riis Nielson and , Flemming Nielson*

on the Occasion of Their 60th Birthdays on Semantics, Logics, and Calculi - Volume 9560. Springer-Verlag, Berlin, Heidelberg, 1–32. https://doi.org/10.1007/978-3-319-27810-0_1

Joseph P. Near, David Darais, Chike Abuah, Tim Stevens, Pranav Gaddamadugu, Lun Wang, Neel Somani, Mu Zhang, Nikhil Sharma, Alex Shan, and Dawn Song. 2019. Duet: an expressive higher-order language and linear type system for statically enforcing differential privacy. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 172 (Oct. 2019), 30 pages. https://doi.org/10.1145/3360598

Flemming Nielson and Hanne Riis Nielson. 1999. Type and Effect Systems. In *Correct System Design: Recent Insights and Advances*, Ernst-Rüdiger Olderog and Bernhard Steffen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 114–136. https://doi.org/10.1007/3-540-48092-7_6

Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. 1999. *Principles of Program Analysis*. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-662-03811-6

Dianne P. O'Leary. 2009. *Scientific Computing with Case Studies*. Society for Industrial and Applied Mathematics, Philadelphia, PA. https://doi.org/10.1137/9780898717723

FWJ Olver. 1982. Further developments of rp and ap error analysis. *IMA J. Numer. Anal.* 2, 3 (1982), 249–274.

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

F. W. J. Olver and J. H. Wilkinson. 1982. A Posteriori Error Bounds for Gaussian Elimination. *IMA J. Numer. Anal.* 2, 4 (10 1982), 377–406. https://doi.org/10.1093/imanum/2.4.377

Dominic Orchard, Tomas Petricek, and Alan Mycroft. 2014. The Semantic Marriage of Monads and Effects. *CoRR* abs/1401.5391 (2014). http://arxiv.org/abs/1401.5391

Dominic A. Orchard, Philip Wadler, and Harley D. Eades. 2020. Unifying Graded and Parameterised

Monads. In *Proceedings of the 2020 Workshop on Mathematically , Structured Functional Programming (MSFP 2020) at ETAPS*. https://api.semanticscholar.org/CorpusID:210932591

Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. 2015. Automatically improving accuracy for floating point expressions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/2737924.2737959

Radhia Cousot Patrick Cousot and, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2005. The ASTREÉ Analyzer. In *esop05 (Lecture Notes in Computer Science, Vol. 3444)*. Springer, 21–30. https://doi.org/10.1007/978-3-540-31987-0_3

Miner Paul S. 1995. *Defining the IEEE-854 Floating-Point Standard in PVS*. Technical Report. NASA Langley.

Tomas Petricek. 2016. *Context-aware programming languages*. Ph. D. Dissertation.

Tomas Petricek, Dominic Orchard, and Alan Mycroft. 2013. Coeffects: Unified static analysis of context-dependence. In *Automata, Languages, and Programming: 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part II 40*. Springer, 385–397.

T. Petricek, D. Orchard, and A. Mycroft. 2014. Coeffects: A Calculus of Context-Dependent Computation. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP 2014)*. ACM, 123–135.

J. D. Pryce. 1984. A New Measure of Relative Error for Vectors. *SIAM J. Numer. Anal.* 21, 1 (1984), 202–215. http://www.jstor.org/stable/2157057

J. D. Pryce. 1985. Multiplicative Error Analysis of Matrix Transformation Algorithms. *IMA J. Numer. Anal.* 5, 4 (10 1985), 437–445. https://doi.org/10.1093/imanum/5.4.437

Tahina Ramananandro, Paul Mountcastle, Benoît Meister, and Richard Lethin. 2016. A unified Coq framework for verifying C programs with floating-point computations. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs* (St. Petersburg, FL, USA) *(CPP 2016)*. Association for Computing Machinery, New York, NY, USA, 15–26. https://doi.org/10.1145/2854065.2854066

Jason Reed and Benjamin C. Pierce. 2010. Distance Makes the Types Grow Stronger: A Calculus for Differential, Privacy. In *Proceedings of the 15th ACM SIGPLAN International Conference on , Functional Programming (ICFP 2010)*. ACM, 157–168. https://doi.org/10.1145/1863543.1863568

Mitchell Riley. 2018. Categories of Optics. arXiv:1809.00738 [math.CT]

Joao Rivera, Franz Franchetti, and Markus Püschel. 2024. Floating-Point TVPI Abstract Domain. *Proc. ACM Program. Lang.* 8, PLDI, Article 165 (jun 2024), 25 pages. https://doi.org/10.1145/3656395

Eric Schkufza, Rahul Sharma, and Alex Aiken. 2014. Stochastic optimization of floating-point programs with tunable precision. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 53–64. https://doi.org/10.1145/2594291.2594302

Benjamin Sherman, Jesse Michel, and Michael Carbin. 2019. Sound and robust solid modeling via exact real arithmetic and , continuity. *Proc. ACM Program. Lang.* 3, ICFP, Article 99 (jul 2019), 29 pages. https://doi.org/10.1145/3341703

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

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

Jean-Pierre Talpin. 1993. *Theoretical and Practical Aspects of Type and Effect Inference*. Ph. D. Dissertation. Ecole des Mines de Paris and University Paris VI.

Jean-Pierre Talpin and Pascal Jouvelot. 1992. Polymorphic Type, Region, and Effect Inference. *Journal of Functional Programming* 2, 3 (July 1992), 245–271. https://doi.org/10.1017/S0956796800000410

Jean-Pierre Talpin and Pascal Jouvelot. 1994. The Type and Effect Discipline. *Information and Computation* 111, 2 (1994), 245–296. https://doi.org/10.1006/inco.1994.1041

R. Tate. 2013. The Sequential Semantics of Producer Effect Systems. In *Proceedings of the Symposium on Principles of Programming Languages, POPL '13*. ACM, New York, NY, USA, 15–26.

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

Laura Titolo, Marco A. Feliú, Mariano M. Moscato, and César A. Muñoz. 2018. An Abstract Interpretation Framework for the Round-Off Error Analysis, of Floating-Point Programs. In *vmcai18 (Lecture Notes in Computer Science, Vol. 10747)*. Springer, 516–537. https://doi.org/10.1007/978-3-319-73721-8_24

Laura Titolo, Mariano Moscato, Marco A. Feliu, Paolo Masci, and César A. Muñoz. 2024. Rigorous Floating-Point Round-Off Error Analysis in PRECiSA 4.0. In *Formal Methods: 26th International*

Symposium, FM 2024, Milan, Italy, September 9-13, 2024, Proceedings, Part II (Milan, Italy). Springer-Verlag, Berlin, Heidelberg, 20–38. https://doi.org/10.1007/978-3-031-71177-0_2

Cassia Torczon, Emmanuel Suárez Acevedo, Shubh Agrawal, Joey Velez-Ginorio, and Stephanie Weirich. 2024. Effects and Coeffects in Call-by-Push-Value. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 310 (Oct 2024), 27 pages. https://doi.org/10.1145/3689750

Anh-Hoang Truong, Huy-Vu Tran, and Bao-Ngoc Nguyen. 2014. Finding Round-Off Error Using Symbolic Execution. In *Knowledge and Systems Engineering*, Van Nam Huynh, Thierry Denoeux, Dang Hung Tran, Anh Cuong Le, and Son Bao Pham (Eds.). Springer International Publishing, Cham, 415–428.

Philip Wadler. 1990. Linear Types Can Change the World!

Philip Wadler and Peter Thiemann. 2003. The marriage of effects and monads. *ACM Trans. Comput. Logic* 4, 1 (Jan 2003), 1–32. https://doi.org/10.1145/601775.601776

Chenkai Weng, Kang Yang, Xiang Xie, Jonathan Katz, and Xiao Wang. 2021. Mystique: Efficient Conversions for Zero-Knowledge Proofs with Applications to Machine Learning. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 501–518. https://www.usenix.org/conference/usenixsecurity21/presentation/weng

James Wood and Robert Atkey. 2022. A Framework for Substructural Type Systems. In *Programming Languages and Systems: 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2–7, 2022, Proceedings* (Munich, Germany). Springer-Verlag, Berlin, Heidelberg, 376–402. https://doi.org/10.1007/978-3-030-99336-8_14

June Wunder, Arthur Azevedo de Amorim, Patrick Baillot, and Marco Gaboardi. 2023. Bunched Fuzz: Sensitivity for Vector Metrics. In *Proceedings of the 32nd European Symposium on Programming (ESOP 2023)*. Springer, 451–478.

Lei Yu. 2013. A Formal Model of IEEE Floating Point Arithmetic. https://isa-afp.org/entries/IEEE_Floating_Point.html

A. Ziv. 1982. Relative Distance–An Error Measure in Round-Off Error Analysis. *Math. Comp.* 39, 160 (1982), 563–569. https://doi.org/10.2307/2007334

APPENDIX A

APPENDIX FOR NUMFUZZ

A.1 Termination (Strong Normalization)

This appendix gives the proof of Theorem 2, which verifies that **NumFuzz** is strongly normalizing using a logical relations argument. In addition to Lemma 4 and Lemma 5, the proof relies on the following lemma.

Lemma 19. Let $\Gamma \vdash e : M_u \tau$ be a well-typed term, and let $\vec{v}_1, \vec{v}_2$ be well-typed substitutions of closed values such that $\vec{v}_1, \vec{v}_2 \vDash \Gamma$. If $e[\vec{v}_1/dom(\Gamma)] \in \mathcal{VR}^m_{M_u \tau}$ for some $m \in \mathbb{N}$ and $e[\vec{v}_2/dom(\Gamma)] \in \mathcal{R}_{M_u \tau}$, then $e[\vec{v}_2/dom(\Gamma)] \in \mathcal{VR}^m_{M_u \tau}$.

Theorem 2. *If* $\emptyset \vdash e : \tau$ *then there exists* $v \in \mathrm{CV}(\tau)$ *such that* $e \mapsto^* v$*.*

Proof. We first prove the following stronger statement. Let $\Gamma \triangleq x_1 :_{s_1} \sigma_1, \ldots, x_i :_{s_i} \sigma_i$ be a typing environment and let $\vec{w}$ denote the values $\vec{w} \triangleq w_1, \ldots, w_i$. If $\Gamma \vdash e : \tau$ and $w_i \in \mathcal{R}_{\Gamma(x_i)}$ for every $x_i \in dom(\Gamma)$, then $e[\vec{w}/dom(\Gamma)] \in \mathcal{R}_\tau$. The proof follows by induction on the derivation $\Gamma \vdash e : \tau$. We consider the monadic cases, as the non-monadic cases are standard. The base cases (Const), (Ret), and (Rnd) follow by definition, and (MSub) follows by Lemma 5. The case for (MLet) requires some detail. The rule is

$$\text{(MLet)}\quad \frac{\Gamma \vdash v : M_r \sigma \qquad \Theta, x :_s \sigma \vdash f : M_q \tau}{s \cdot \Gamma + \Theta \vdash \textbf{let}_\textbf{M}\ x = v\ \textbf{in}\ f : M_{s \cdot r + q} \tau}$$

and so we are required to show $(\textbf{let}_\textbf{M}\ x = v\ \textbf{in}\ f) \in \mathcal{R}_{M_{s \cdot r+q}\tau}$. We proceed by cases on v.

Subcase: $v \equiv$ **rnd** k **with** $k \in \mathrm{R}$**.** Let $\Delta = s \cdot \Gamma + \Phi$. By Lemma 3,

$$(\textbf{let}_\textbf{M}\ x = \textbf{rnd}\ k\ \textbf{in}\ f)[\vec{w}/dom(\Delta)] \in \mathrm{CV}(M_{s \cdot r+q}\tau)$$

and it remains to be shown that

$$(\textsf{\textbf{let}}_{\textsf{\textbf{M}}}\ x = \textsf{\textbf{rnd}}\ k\ \textsf{\textbf{in}}\ f)[\vec{w}/dom(\Delta)] \in \mathcal{VR}_{\mathrm{M}_{s\cdot r+q}\tau}.$$

By definition of the logical relation, **rnd** $k \in \mathcal{VR}^0_{\mathrm{M}_s\sigma}$. Given $\vec{w} \in \mathcal{R}_{\Delta(x_i)}$ for every $x_i \in \Delta$, we have the vector of values $\vec{w}'$ such that $w'_i \in \mathcal{R}_{\Phi(y_i)}$ for every $y_i \in dom(\Phi)$. By the induction hypothesis and Lemma 4 we have

$$f[\vec{w}'/dom(\Phi)][k/x] \in \mathcal{VR}^n_{\mathrm{M}_q\tau}$$

for some $n \in \mathbb{N}$. If $n = 0$, then the conclusion follows trivially. Otherwise, $n > 0$, and we need to show that, given some $t \in \mathcal{VR}_{\mathrm{num}}$,

$$f[\vec{w}'/dom(\Phi)][t/x] \in \mathcal{VR}^n_{\mathrm{M}_q\tau}$$

This follows by the induction hypothesis and Lemma 19.

Subcase: $v \equiv$ **ret** v' **with** $v' \in \mathcal{VR}_\sigma$**.** The conclusion follows by applying reasoning identical to that of the previous subcase.

Subcase: $v \equiv$ $\textsf{\textbf{let}}_{\textsf{\textbf{M}}}$ y **=** $\textsf{\textbf{rnd}}$ k $\textsf{\textbf{in}}$ g**.** From the reduction rules we have

$$(\textsf{\textbf{let}}_{\textsf{\textbf{M}}}\ x = v\ \textsf{\textbf{in}}\ f)[\vec{w}/dom(\Delta)] \mapsto (\textsf{\textbf{let}}_{\textsf{\textbf{M}}}\ y = \textsf{\textbf{rnd}}\ k\ \textsf{\textbf{in}}\ (\textsf{\textbf{let}}_{\textsf{\textbf{M}}}\ x = g\ \textsf{\textbf{in}}\ f))[\vec{w}/dom(\Delta)]$$

with $y \notin \mathrm{FV}(f)$. By Lemma 3 we have

$$\emptyset \vdash (\textsf{\textbf{let}}_{\textsf{\textbf{M}}}\ x = (\textsf{\textbf{let}}_{\textsf{\textbf{M}}}\ y = \textsf{\textbf{rnd}}\ k\ \textsf{\textbf{in}}\ g)\ \textsf{\textbf{in}}\ f)[\vec{w}/dom(\Delta)] : \mathrm{M}_{s\cdot r+q}\tau$$

and, by Lemma 4, it is therefore sufficient to show

$$(\textsf{\textbf{let}}_{\textsf{\textbf{M}}}\ y = \textsf{\textbf{rnd}}\ k\ \textsf{\textbf{in}}\ (\textsf{\textbf{let}}_{\textsf{\textbf{M}}}\ x = g\ \textsf{\textbf{in}}\ f))[\vec{w}/dom(\Delta)] \in \mathcal{R}_{\mathrm{M}_{s\cdot r+q}}\tau,$$

which follows by applying reasoning identical to that of the previous two subcases.

□

Lemma 20. If $\emptyset \vdash e : \tau$ then $e \in \mathcal{R}_\tau$.

The following lemma follows directly from the definition of the reducibility predicate.

Lemma 21. If $e \in \mathcal{R}_\tau$ then there exists a $v \in \mathrm{CV}(\tau)$ such that $e \mapsto^* v$.

The proof of termination (Theorem 2) then follows from Lemma 21 and Lemma 20.

A.2 The Neighborhood Monad

This appendix provides the details verifying that the neighborhood monad defined in Section 3.4.3 forms an $\mathcal{R}$-strong graded monad on Met.

Lemma 6. *Let $q, r \in \mathcal{R}$. For any metric space A, the maps $(q \leq r)_\mathrm{A}$, η_A, and $\mu_{q,r,\mathrm{A}}$ are non-expansive maps and natural in A.*

Proof. Non-expansiveness and naturality for the subeffecting maps $(q \leq r)_\mathrm{A} : \mathrm{T}_q\mathrm{A} \to \mathrm{T}_r\mathrm{A}$ and the unit maps $\eta_\mathrm{A} : \mathrm{A} \to \mathrm{T}_0\mathrm{A}$ are straightforward. We describe the checks for the multiplication map $\mu_{q,r,\mathrm{A}}$.

First, we check that the multiplication map has the claimed domain and codomain. Note that $d_\mathrm{A}(x, x') = d_{\mathrm{T}_r\mathrm{A}}((x, y), (x', y')) \leq q$ because of the definition of T_q, and $d_\mathrm{A}(x', y') \leq r$ because of the definition of T_r, so via the triangle inequality we have $d_\mathrm{A}(x, y') \leq r + q$ as claimed.

Second, we check non-expansiveness. Let $((x, y), (x', y'))$ and $((w, z), (w', z'))$ be two elements

of $T_q(T_r A)$. Then:

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

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

Lemma 8. *The neighborhood monad (Definition 22) together with the* tensorial strength maps $st_{r,A,B} : A \otimes T_r B \to T_r(A \otimes B)$ *defined as*

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

for every $r \in \mathcal{R}$ form a $\mathcal{R}$-strong graded monad on Met.

Proof. We must verify the non-expansiveness (Definition 19) and naturality (Definition 12) of the tensorial strength map. We check the non-expansiveness with respect to the tensor product $\otimes$, although it is also easy to show non-expansiveness with respect to the product &: For $(a,(b,b'))$ and $(c,(d,d'))$ in $A \otimes T_r B$, we have:

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

Finally, checking the naturality of the strength follows directly by unfolding definitions. □

Lemma 9. *Let $s \in \mathcal{S}$ and $r \in \mathcal{R}$ be grades, and let* A *be a metric space. Then identity map on the carrier set* $|\mathrm{A}| \times |\mathrm{A}|$ *is a non-expansive map*

$$\lambda_{s,r,\mathrm{A}} : \mathrm{D}_s(\mathrm{T}_r\mathrm{A}) \to \mathrm{T}_{s \cdot r}(\mathrm{D}_s\mathrm{A})$$

Moreover, these maps are natural in A.

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

$$\begin{aligned}
d_{\mathrm{T}_{s\cdot r}(\mathrm{D}_s\mathrm{A})}((x, y), (x', y')) &\triangleq d_{\mathrm{D}_s\mathrm{A}}(x, x') && (\text{def. } \mathrm{T}_{s\cdot r}) \\
&\triangleq s \cdot d_{\mathrm{A}}(x, x') && (\text{def. } \mathrm{D}_s) \\
&\triangleq s \cdot d_{\mathrm{T}_r\mathrm{A}}((x, y), (x', y')) && (\text{def. } \mathrm{T}_r) \\
&\triangleq d_{\mathrm{D}_s(\mathrm{T}_r\mathrm{A})}((x, y), (x', y')) && (\text{def. } \mathrm{D}_s)
\end{aligned}$$

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

A.3 Interpreting NumFuzz Terms

This appendix provides the constructions of the interpretation of **NumFuzz** terms for Definition 24 that were not included in Section 3.4.

To reduce notation, we elide the the unitors $\lambda_{\mathrm{A}} : \mathrm{I} \otimes \mathrm{A} \to \mathrm{A}$ and $\rho_{\mathrm{A}} : \mathrm{A} \otimes \mathrm{I} \to \mathrm{I}$; the associators $\alpha_{\mathrm{A,B,C}} : (\mathrm{A} \otimes \mathrm{B}) \otimes \mathrm{C} \to \mathrm{A} \otimes (\mathrm{B} \otimes \mathrm{C})$; and the symmetries $\sigma_{\mathrm{A,B}} : \mathrm{A} \otimes \mathrm{B} \to \mathrm{B} \otimes \mathrm{A}$.

(Unit). Define $[\![\Gamma \vdash () : \mathrm{unit}]\!]$ as the map that sends all points in Γ to $\star \in [\![\mathrm{unit}]\!]$.

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

(Abs). Let $f = \llbracket \Gamma, x :_1 \sigma \vdash e : \tau \rrbracket : \llbracket \Gamma \rrbracket \otimes D_1 \llbracket \sigma \rrbracket \to \llbracket \tau \rrbracket$. Define

$$\llbracket \Gamma \vdash \lambda x.e : \sigma \multimap \tau \rrbracket \triangleq \lambda(f)$$

The map $\lambda(f) : \llbracket \Gamma \rrbracket \to (\llbracket \sigma \rrbracket \multimap \llbracket \tau \rrbracket)$ is guaranteed by the closed symmetric monoidal structure of Met (Theorem 3). The equality $D_1 \llbracket \sigma \rrbracket = \llbracket \sigma \rrbracket$ follows by definition of the comonad.

(App). Let $f = \llbracket \Gamma \vdash v : \sigma \multimap \tau \rrbracket$ and $g = \llbracket \Theta \vdash w : \sigma \rrbracket$. Define

$$\llbracket \Gamma + \Theta \vdash vw : \tau \rrbracket \triangleq c_{\llbracket \Gamma \rrbracket, \llbracket \Theta \rrbracket}; (f \otimes g); ev : \llbracket \Gamma + \Theta \rrbracket \to \llbracket \tau \rrbracket$$

The map $ev : (A \multimap B) \otimes A \to B$ is guaranteed by the closed symmetric monoidal structure of Met (Theorem 3).

(& I). Let $f = \llbracket \Gamma \vdash v : \sigma \rrbracket$ and $g = \llbracket \Gamma \vdash w : \tau \rrbracket$. Then, define:

$$\llbracket \Gamma \vdash \langle v, w \rangle : \sigma \,\&\, \tau \rrbracket \triangleq \langle f, g \rangle$$

(& E). Let $f = \llbracket \Gamma \vdash v : \tau_1 \,\&\, t_2 \rrbracket$ be the denotation of the premise. Define

$$\llbracket \Gamma \vdash \pi_i \, v : \tau_i \rrbracket \triangleq f; \pi_i$$

(⊗ I). Let $f = \llbracket \Gamma \vdash v : \sigma \rrbracket$ and $g = \llbracket \Theta \vdash w : \tau \rrbracket$ be the denotations of the premises. Then, define:

$$\llbracket \Gamma + \Theta \vdash (v, w) : \sigma \otimes \tau \rrbracket \triangleq c_{\llbracket \Gamma \rrbracket, \llbracket \Theta \rrbracket}; (f \otimes g) : \llbracket \Gamma + \Theta \rrbracket \to \llbracket \sigma \otimes \tau \rrbracket$$

(⊗ E). Let the denotations for the premises be:

$$f = \llbracket \Gamma \vdash v : \sigma \otimes \tau \rrbracket : \llbracket \Gamma \rrbracket \to \llbracket \sigma \rrbracket \otimes \llbracket \tau \rrbracket$$
$$g = \llbracket \Theta, x :_s \sigma, y :_s \tau \vdash e : \rho \rrbracket : \llbracket \Theta \rrbracket \otimes D_s \llbracket \sigma \rrbracket \otimes D_s \llbracket \tau \rrbracket \to \llbracket \rho \rrbracket$$

Then, define:

$$\llbracket s \cdot \Gamma + \Theta \vdash \textbf{let } (x, y) = v \textbf{ in } e : \rho \rrbracket \triangleq c_{\llbracket s \cdot \Gamma \rrbracket, \llbracket \Theta \rrbracket}; h; g$$

The map $h : [\![s \cdot \Gamma]\!] \otimes [\![\Theta]\!] \to [\![\Theta]\!] \otimes D_s[\![\sigma]\!] \otimes D_s[\![\tau]\!]$ is constructed as follows. Applying the functor D_s to f and pre-composing with m yields

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

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

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

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

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

(+ I$_\mathrm{L}$). Let $f = [\![\Gamma \vdash v : \sigma]\!]$ be the denotation of the premise. Then, define:

$$[\![\Gamma \vdash \mathbf{inl}\, v : \sigma + \tau]\!] \triangleq f; \iota_1$$

where ι_1 is the first injection into the coproduct.

(+ I$_\mathrm{R}$). Let $f = [\![\Gamma \vdash v : \tau]\!]$ be the denotation of the premise. Then, define:

$$[\![\Gamma \vdash \mathbf{inr}\, v : \sigma + \tau]\!] \triangleq f; \iota_r$$

where ι_2 is the second injection into the coproduct.

(+ E). This particular case requires as few additional facts about the structures in our category. First, when $s > 0$, there is an isomorphism

$$\mathrm{dist}^{\mathrm{D}}_s : D_s(A + B) \cong D_s A + D_s B$$

Second, there is a map

$$\mathrm{dist}_{A,B,C} : A \times (B + C) \to A \times B + A \times C$$

that pushes the first component into the disjoint union. This map is non-expansive, and in fact Met is a distributive category.

Now, let $f = [\![\Gamma \vdash v : \sigma + \tau]\!]$ and $g_i = [\![\Theta, x_i :_s \sigma \vdash e_i : \rho]\!]$ for $i = 1, 2$ and $s > 0$. Then, define

$$[\![s \cdot \Gamma + \Theta \vdash \textbf{case } v \textbf{ of } (\textbf{inl } x.e \mid \textbf{inr } y.f) : \rho]\!] \triangleq c_{[\![s \cdot \Gamma]\!],[\![\Theta]\!]}; h; [g_1, g_2]$$

The map h is constructed as follows. Since $s > 0$, dist_s is an isomorphism. Using the functor D_s on f, composing in parallel with $id_{[\![\Theta]\!]}$ and distributing, we have:

$$h = (id_{[\![\Theta]\!]} \otimes (m; D_s f; \text{dist}_s^D)); \text{dist}_{[\![\Theta]\!],[\![\sigma]\!],[\![\tau]\!]} : [\![\Theta]\!] \otimes [\![s \cdot \Gamma]\!] \to [\![\Theta]\!] \otimes D_s[\![\sigma]\!] + [\![\Theta]\!] \otimes D_s[\![\tau]\!]$$

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

(! I). Let $f = [\![\Gamma \vdash v : \sigma]\!]$ be the denotation of the premise. Then, define:

$$[\![s \cdot \Gamma \vdash [v] : !_s : \sigma]\!] \triangleq m; D_s f$$

(! E). Let $f = [\![\Gamma \vdash v : !_s\sigma]\!]$ and $g = [\![\Theta, x :_{m \cdot n} \sigma \vdash e : \tau]\!]$. Then, define:

$$[\![t \cdot \Gamma + \Theta \vdash \textbf{let } [x] = v \textbf{ in } e : \tau]\!] \triangleq m; D_m f; \delta^{-1}_{m,n,[\![\sigma]\!]} : [\![m \cdot \Gamma]\!] \to D_{m \cdot n}[\![\sigma]\!]$$

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

(Let). Let $f = \Gamma \vdash e : \tau$ and let $g = [\![\Theta, x :_s \tau \vdash f : \sigma]\!]$. Then, define:

$$[\![s \cdot \Gamma + \Theta \vdash \textbf{let } x = e \textbf{ in } f : \sigma]\!] \triangleq c_{[\![s \cdot \Gamma]\!],[\![\Theta]\!]}; h; g$$

Similar to the other elimination cases, the map $h : [\![s \cdot \Gamma]\!] \otimes [\![\Theta]\!] \to [\![\Theta]\!] \otimes D_s[\![\tau]\!]$ is constructed as follows. Applying the functor D_s to f, pre-composing with m, and composing in parallel with $\text{id}_{[\![\Theta]\!]}$ yields

$$h = (m; D_s f) \otimes \text{id}_{[\![\Theta]\!]} : [\![\Theta]\!] \otimes [\![s \cdot \Gamma]\!] \to D_s([\![tau]\!]) \otimes [\![\Theta]\!]$$

A.4 Denotational Semantics

This appendix provides basic lemmas about the denotational semantics: weakening (Lemma 22), subsumption (Lemma 23), and substitution (Lemma 24). It also includes a computational soundness lemma (Lemma 25) showing that our metric interpretation of **NumFuzz** terms respect the operational semantics given in Figure 3.3; these are the semantics defined prior to the refinement into a ideal and floating-point step relations.

Lemma 22 (Weakening)**.** Let $\Gamma, \Gamma' \vdash e : \tau$ be a well-typed term. Then for any context, there is a derivation of $\Gamma, \Delta, \Gamma' \vdash e : \tau$ with semantics $[\![\Gamma, \Gamma' \vdash e : \tau]\!] \circ \pi$, where $\pi : [\![\Gamma, \Delta, \Gamma']\!] \to [\![\Gamma, \Gamma']\!]$ projects the components in Γ and Γ'.

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

Lemma 23 (Subsumption)**.** Let $\Gamma \vdash e : M_r\tau$ be a well-typed program of monadic type, where the typing derivation concludes with the subsumption rule. Then either e is of the form **ret**v or **rnd** k, or there is a derivation of $\Gamma \vdash e : M_r\tau$ with the same semantics that does not conclude with the subsumption rule.

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

Lemma 24 (Substitution)**.** Let $\Gamma, \Delta, \Gamma' \vdash e : \tau$ be a well-typed term, and let $\vec{v} : \Delta$ be a well-typed substitution of closed values, i.e., we have derivations $\emptyset \vdash v_x : \Delta(x)$. Then there is a derivation of

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

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

Proof. By induction on the typing derivation of $\Gamma, \Delta, \Gamma' \vdash e : \tau$. The base cases Unit and Const are obvious. The other base case Var follows by unfolding the definition of the semantics. Most of the rest of the cases follow from the substitution lemma for *Fuzz* (Azevedo de Amorim et al., 2017, Lemma 3.3). We show the cases for **Rnd**, **Ret**, and **MLet**, which differ from *Fuzz*. We omit the bookkeeping morphisms.

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

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

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

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

Case MLet. Suppose that $\Gamma = \Gamma_1, \Delta_1, \Gamma_2$ and $\Theta = \Theta_1, \Delta_2, \Theta_2$ such that $\Delta = s \cdot \Delta_1 + \Delta_2$. By combining Lemma 10 and Lemma 11, there is a natural transformation $\sigma : [\![s \cdot \Gamma + \Theta]\!] \to [\![\Theta]\!] \otimes D_s[\![\Gamma]\!]$. Let $g_1 = [\![\Gamma_1, \Delta_1, \Gamma_2 \vdash w : M_r \sigma]\!]$ and $g_2 = [\![\Theta_1 1, \Delta_2, \Theta_2, x :_s \sigma \vdash f : M_{r'} \tau]\!]$. By induction, we have:

$$\tilde{g_1} = [\![\Gamma_1, \Gamma_2 \vdash w[\vec{v}/\Delta] : M_r \sigma]\!] = (id_{[\![\Gamma_1]\!]} \otimes [\![\emptyset \vdash \vec{v} : \Delta]\!] \otimes id_{[\![\Gamma_2]\!]}); g_1$$

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

Thus we have a derivation of the judgment $s \cdot (\Gamma_1, \Gamma_2) + (\Theta_1, \Theta_2) \vdash \mathsf{let}_{\mathsf{M}}\ x = w\ \mathsf{in}\ f[\vec{v}/\Delta] : M_{s \cdot r + r'} \tau$, and by the definition of the semantics of **MLet**, its semantics is:

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

From here, we can conclude by showing that the first morphisms in $\tilde{g_1}$ and $\tilde{g_2}$ can be pulled out to the front. For instance,

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

by functoriality. By naturality of $split$, the first morphism can be pulled out in front of $split$.

Similarly, for $\tilde{g}_2$, we have:

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

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

□

Lemma 25 (Computational Soundness (Metric Semantics))**.** Let $\emptyset \vdash e : \tau$ be a well-typed closed term, and suppose $e \mapsto e'$. Then there is a derivation of $\emptyset \vdash e' : \tau$, and the semantics of both derivations are equal:

$$[\![\vdash e : \tau]\!] = [\![\vdash e' : \tau]\!].$$

Proof. By case analysis on the step rule, using the fact that e is well-typed. For the beta-reduction steps for programs of non-monadic type, preservation follows by the soundness theorem; these cases are exactly the same as in *Fuzz* (Azevedo de Amorim et al., 2017).

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

MLet q**.** Suppose that $e = \mathbf{let_M}\ x = \mathbf{ret}\,v\ \mathbf{in}\ f$ is a well-typed program with type $M_{s\cdot r+q}\tau$. Since subsumption is admissible (Lemma 23), we may assume that the last rule is **MLet** and we have derivations $\emptyset \vdash \mathbf{ret}\,v : M_r\sigma$ and $x :_s \sigma \vdash f : M_q\tau$. By definition, the semantics of $\emptyset \vdash e : M_{s\cdot r+q}\tau$ is given by the composition:

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

By substitution (Lemma 24), we have a derivation of $\cdot \vdash f[v/x] : M_q\tau$. By applying the subsumption rule, we have a derivation of $\emptyset \vdash f[v/x] : M_{s\cdot r+q}\tau$ with semantics:

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

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

MLet Assoc. Suppose that $e = \textbf{let}_\textbf{M}\ y = \textbf{let}_\textbf{M}\ x = \textbf{rnd}\ k\ \textbf{in}\ f\ \textbf{in}\ g$ is a well-typed program with type $M_{s\cdot r+q}\tau$. Since subsumption is admissible (Lemma 23), we have derivations:

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

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

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

And by applying **MLet** again, we have:

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

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

APPENDIX B

APPENDIX FOR BEAN

B.1 The Category of Backward Error Lenses

This appendix verifies that the composition in the category **Bel** of backward error lenses (Definition 29) is well-defined. We first restate the definition: the composition

$$(f_2, \tilde{f}_2, b_2) \circ (f_1, \tilde{f}_1, b_1)$$

of error lenses $(f_1, \tilde{f}_1, b_1) : \mathrm{X} \to \mathrm{Y}$ and $(f_2, \tilde{f}_2, b_2) : \mathrm{Y} \to \mathrm{Z}$ is the error lens $(f, \tilde{f}, b) : \mathrm{X} \to \mathrm{Z}$ defined by

- the forward map

$$f : x \mapsto (f_1 ; f_2)\ x() \qquad \text{Equation (4.14)}$$

- the approximation map

$$\tilde{f} : x \mapsto (\tilde{f}_1 ; \tilde{f}_2)\ x() \qquad \text{Equation (4.15)}$$

- the backward map

$$b : (x, z) \mapsto b_1\big(x, b_2(\tilde{f}_1(x), z)\big)() \qquad \text{Equation (4.16)}$$

The diagram for the backward map for the composition of error lenses is given in Figure B.1.

$$\begin{array}{ccc} \mathrm{X} \times \mathrm{Y} \times \mathrm{Z} & \xrightarrow{id_{\mathrm{X}} \times\, b_2} & \mathrm{X} \times \mathrm{Y} \\ \uparrow {\scriptstyle \langle id_{\mathrm{X}}, \tilde{f}_1 \rangle \times id_{\mathrm{Z}}} & & \downarrow {\scriptstyle b_1} \\ \mathrm{X} \times \mathrm{Z} & \xrightarrow[b]{} & \mathrm{X} \end{array}$$

Figure B.1: The backward map b for the composition $(f_2, \tilde{f}_2, b_2) \circ (f_1, \tilde{f}_1, b_1)$.

Let $L_1 = (f_1, \tilde{f}_1, b_1)$ and let $L_2 = (f_2, \tilde{f}_2, b_2)$. We first check the domain: for all $x \in X$ and $z \in Z$, and assuming $d_Z\left(\tilde{f}(x), z\right) \neq \infty$, we must show

$$d_Y\left(\tilde{f}_1(x), b_2(\tilde{f}_1(x), z)\right) \neq \infty.$$

This follows from Property 1 for L_2 and the assumption:

$$d_Y\left(\tilde{f}_1(x), b_2(\tilde{f}_1(x), z)\right) \leq d_Z\left(\tilde{f}_2(\tilde{f}_1(x)), z\right) \tag{B.1}$$

$$= d_Z\left(\tilde{f}(x), z\right) \neq \infty \tag{B.2}$$

Now, given that $d_Y\left(\tilde{f}_1(x), b_2(\tilde{f}_1(x), z)\right) \neq \infty$ holds for all $x \in X$ and $z \in Z$ under the assumption of $d_Z\left(\tilde{f}(x), z\right) \neq \infty$, we can freely use Properties 1 and 2 of the lens L_1 to show that the lens properties hold for the composition:

Property 1.

$$\begin{aligned}
d_X(x, b(x,z)) &= d_X\left(x, b_1\left(x, b_2(\tilde{f}_1(x), z)\right)\right) && \text{Eq. (4.16)} \\
&\leq d_Y\left(\tilde{f}_1(x), b_2(\tilde{f}_1(x), z)\right) && \text{Property 1 for } L_1 \\
&\leq d_Z\left(\tilde{f}_2(\tilde{f}_1(x)), z\right) && \text{Property 1 for } L_2 \\
&= d_Z(\tilde{f}(x), z) && \text{Eq. (4.15)}
\end{aligned}$$

Property 2.

$$\begin{aligned}
f(b(x,z)) &= f_2\left(f_1\left(b_1\left(x, b_2(\tilde{f}_1(x), z)\right)\right)\right) && \text{Eq. (4.16) \& Eq. (4.14)} \\
&= f_2\left(b_2(\tilde{f}_1(x), z)\right) && \text{Property 2 for } L_1 \\
&= z && \text{Property 2 for } L_2
\end{aligned}$$

B.2 Basic Constructions in Bel

This appendix verifies that the basic constructions in **Bel** from Section 4.3.2 are well-defined.

B.2.1 Tensor Product

The tensor product given in Equations (4.17) to (4.19) is only well-defined if the domain of the backward map is well-defined, and if the error lens properties hold. We check these properties below, and restate the definition of the tensor product lens here for convenience:

Given any two morphisms $(f, \tilde{f}, b) : \mathrm{A} \to \mathrm{X}$ and $(g, \tilde{g}, b') : \mathrm{B} \to \mathrm{Y}$, we have the morphism

$$(f, \tilde{f}, b) \otimes (g, \tilde{g}, b') : \mathrm{A} \otimes \mathrm{B} \to \mathrm{X} \otimes \mathrm{Y}$$

defined by

- the forward map

$$(a_1, a_2) \mapsto (f(a_1), g(a_2)) \qquad \text{Equation (4.17)}$$

- the approximation map

$$(a_1, a_2) \mapsto (\tilde{f}(a_1), \tilde{g}(a_2)) \qquad \text{Equation (4.18)}$$

- the backward map

$$((a_1, a_2), (x_1, x_2)) \mapsto (b(a_1, x_1), b'(a_2, x_2)) \qquad \text{Equation (4.19)}$$

We first check the domain: for all $(a_1, a_2) \in \mathrm{A} \otimes \mathrm{B}$ and $(x_1, x_2) \in \mathrm{X} \otimes \mathrm{Y}$, we assume

$$d_{\mathrm{X}\otimes\mathrm{Y}}(\tilde{f}_{\otimes}(a_1, a_2), (x_1, x_2)) \neq \infty \tag{B.3}$$

and we are required to show

$$d_{\mathrm{X}}(\tilde{f}(a_1), x_1) \neq \infty \text{ and } d_{\mathrm{Y}}(\tilde{g}(a_2), x_2) \neq \infty \tag{B.4}$$

which follows directly by assumption.

Given that Equation (B.4) holds for all $(a_1, a_2) \in \mathrm{A} \otimes \mathrm{B}$ and $(x_1, x_2) \in \mathrm{X} \otimes \mathrm{Y}$ under the assumption given in Equation (B.3), we can freely use Properties 1 and 2 of the lenses $(f, \tilde{f}, b)$ and $(g, \tilde{g}, b')$ to show that the lens properties hold for the product:

Property 1.

$$d_{A\otimes B}\big((a_1,a_2), b_\otimes((a_1,a_2),(x_1,x_2))\big) = \max\big(d_A(a_1, b(a_1,x_1)), d_B(a_2, b'(a_2,x_2))\big) \quad \text{Eq. (4.19)}$$
$$\leq \max\big(d_X(\tilde{f}(a_1),x_1), d_Y(\tilde{g}(a_2),x_2)\big)$$
$$\text{(Property 1 of } (f,\tilde{f},b) \text{ \& } (g,\tilde{g},b'))$$

Property 2. As above, the property follows directly from Property 2 of the component $(f, \tilde{f}, b)$ and $(g, \tilde{g}, b')$.

Tensor product as bifunctor

Lemma 14. *The tensor product operation on lenses induces a bifunctor on* ***Bel****.*

Proof. The functoriality of the triple given in Equations (4.17) to (4.19) follows by checking conditions expressing preservation of composition and identities. Specifically, for any error lenses $h : A \to B$ $h' : A' \to B'$ $g : B \to C$ and $g' : B' \to C'$ we must show

$$(g \otimes g') \circ (h \otimes h') = (g \circ h) \otimes (g' \circ h')$$

We check the backward map:

Given any $(a_1, a_2) \in A \otimes A'$ and $(c_1, c_2) \in C \otimes C'$ we have

$$b_{(g\otimes g')\circ(h\otimes h')}((a_1,a_2),(c_1,c_2)) = b_{h\otimes h'}\big((a_1,a_2), b_{g\otimes g'}\big(\tilde{f}_{h\otimes h'}(a_1,a_2),(c_1,c_2)\big)\big) \tag{B.5}$$
$$= (b_h(a_1, b_g(\tilde{f}_h(a_1),c_1)), b_{h'}(a_2, b_{g'}(\tilde{f}_{h'}(a_2),c_2))) \tag{B.6}$$
$$= b_{(g\circ h)\otimes(g'\circ h')}((a_1,a_2),(c_1,c_2)) \tag{B.7}$$

Moreover, for any objects X and Y in **Bel**, the identity lenses id_X and id_Y clearly satisfy

$$id_X \otimes id_Y = id_{X\otimes Y}$$

□

Associator

We define the associator $\alpha_{X,Y,Z} : X \otimes (Y \otimes Z) \to (X \otimes Y) \otimes Z$ as the following triple:

$$f_\alpha(x, (y, z)) \triangleq ((x, y), z) \tag{B.8}$$

$$\tilde{f}_\alpha(x, (y, z)) \triangleq ((x, y), z) \tag{B.9}$$

$$b_\alpha((x, (y, z)), ((a, b), c)) \triangleq (a, (b, c)). \tag{B.10}$$

It is straightforward to check that $\alpha_{X,Y,Z}$ is an error lens satisfying Properties 1 and 2. To check that the associator is an isomorphism, we are required to show the existence of the lens

$$\alpha' : (X \otimes Y) \otimes Z \to X \otimes (Y \otimes Z)$$

satisfying

$$\alpha' \circ \alpha_{X,Y,Z} = id_{X \otimes (Y \otimes Z)}$$

and

$$\alpha_{X,Y,Z} \circ \alpha' = id_{(X \otimes Y) \otimes Z}$$

where id is the identity lens (see Definition 29). Defining the forward and approximation maps for α' is straightforward; for the forward map we have

$$f_{\alpha'}((x, y), z) \triangleq (x, (y, z))$$

and the approximation map is defined identically. For the backward map we have

$$b_{\alpha'}\big(((x, y), z), (a, (b, c))\big) \triangleq ((a, b), c)$$

It is straightforward to check that α' satisfies Properties 1 and 2 of an error lens.

The naturality of the associator follows by checking that the following diagram commutes.

$$\begin{array}{ccc} X \otimes (Y \otimes Z) & \xrightarrow{g_X \otimes (g_Y \otimes g_Z)} & X \otimes (Y \otimes Z) \\ \downarrow{\scriptstyle \alpha_{X,Y,Z}} & & \downarrow{\scriptstyle \alpha_{X,Y,Z}} \\ (X \otimes Y) \otimes Z & \xrightarrow[(g_X \otimes g_Y) \otimes g_Z]{} & (X \otimes Y) \otimes Z \end{array}$$

That is, we check that

$$((g_X \otimes g_Y) \otimes g_Z) \circ \alpha_{X,Y,Z} = \alpha_{X,Y,Z} \circ (g_X \otimes (g_Y \otimes g_Z))$$

for the error lenses

$$g_X : X \to X \triangleq (f_X, \tilde{f}_X, b_X)$$
$$g_Y : Y \to Y \triangleq (f_Y, \tilde{f}_Y, b_Y)$$
$$g_Z : Z \to Z \triangleq (f_Z, \tilde{f}_Z, b_Z)$$

This follows from the definitions of lens composition (Definition 28) and the tensor product on lenses (Eqs. (4.17) to (4.19)). We detail here the case of the backward map.

Using the notation b_g (resp. $\tilde{f}_g$) to refer to both of the backward maps (resp. approximation maps) of the tensor product lenses of the lenses g_X, g_Y, and g_Z, we are required to show that

$$b_\alpha\left(xyz, b_g\left(\tilde{f}_\alpha(xyz), x'y'z'\right)\right) = b_g\left(xyz, b_\alpha\left(\tilde{f}_g(xyz), x'y'z'\right)\right) \tag{B.11}$$

for any $xyz \in X \otimes (Y \otimes Z)$ and $x'y'z' \in (X \otimes Y) \otimes Z$:

$$b_\alpha\left(xyz, b_g\left(\tilde{f}_\alpha(xyz), x'y'z'\right)\right) = b_g\left(xyz, b_\alpha\left(\tilde{f}_g(xyz), x'y'z'\right)\right)$$

$$b_\alpha\left(xyz, b_g\left(\tilde{f}_\alpha(xyz), x'y'z'\right)\right) = b_g\left(xyz, (x', (y', z'))\right) \quad \text{by Equation (B.10)}$$

$$b_\alpha\left(xyz, b_g\left(((x, y), z), x'y'z'\right)\right) = b_g\left(xyz, (x', (y', z'))\right) \quad \text{by Equation (B.9)}$$

$$b_\alpha\left(xyz, ((b_X(x, x'), b_Y(y, y')), b_Z(z, z'))\right) = (b_X(x, x'), (b_Y(y, y'), b_Z(z, z'))) \quad \text{by Equation (4.19)}$$

$$(b_X(x, x'), (b_Y(y, y'), b_Z(z, z'))) = (b_X(x, x'), (b_Y(y, y'), b_Z(z, z'))) \quad \text{by Equation (B.10)}$$

Unitors

We define the left-unitor $\lambda_X : I \otimes X \to X$ as

$$f_\lambda(\star, x) \triangleq x$$

$$\tilde{f}_\lambda(\star, x) \triangleq x$$

$$b_\lambda((\star, x), x') \triangleq (\star, x')$$

The right-unitor is similarly defined.

The fact that $d_I(\star, \star) = -\infty$ is essential in order for λ_X to satisfy the first property of an error lens:

$$d_{I\otimes X}(x, b_\lambda((\star, x), x')) \leq d_X(\tilde{f}_\lambda(\star, x), x')$$

$$\max(-\infty, d_X(x, x')) \leq d_X(x, x')$$

Checking the naturality of λ_X amounts to checking that the following diagram commutes for all error lenses $g : X \to Y$.

$$\begin{array}{ccc} I \otimes X & \xrightarrow{id_I \otimes g} & I \otimes Y \\ \downarrow{\scriptstyle \lambda_X} & & \downarrow{\scriptstyle \lambda_Y} \\ X & \xrightarrow[g]{} & Y \end{array}$$

Symmetry

We define the symmetry map $\gamma_{X,Y} : X \otimes Y \to Y \otimes X$ as the following triple:

$$f_\gamma(x, y) \triangleq (y, x)$$

$$\tilde{f}_\gamma(x, y) \triangleq (y, x)$$

$$b_\gamma((x, y), (y', x')) \triangleq (x', y')$$

It is straightforward to check that $\gamma_{X,Y}$ is an error lens. Checking the naturality of $\gamma_{X,Y}$ amounts to checking that the following diagram commutes for any error lenses $g_1 : X \to Y$ and $g_2 : Y \to X$.

$$\begin{array}{ccc} X \otimes Y & \xrightarrow{g_1 \otimes g_2} & Y \otimes X \\ {\scriptstyle \gamma_{X,Y}}\downarrow & & \downarrow{\scriptstyle \gamma_{Y,X}} \\ Y \otimes X & \xrightarrow[g_2 \otimes g_1]{} & X \otimes Y \end{array}$$

B.2.2 Coproducts

Property 1 For any $x \in X$ and $z \in X + Y$, $d_X(x, b_{in_1}(x, z)) \leq d_{X+Y}(\tilde{f}_{in_1}(x), z)$ supposing

$$d_{X+Y}(\tilde{f}_{in_1}(x), z) = d_{X+Y}(inl\ x, z) \neq \infty.$$

From $d_{X+Y}(inl\ x, z) \neq \infty$ and Equation (4.20), we know $z = inl\ x_0$ for some $x_0 \in X$, and so we must show

$$d_X(x, x_0) \leq d_{X+Y}(inl\ x, inl\ x_0)$$

which follows from Equation (4.20) and reflexivity.

Property 2 For any $x \in X$ and $z \in X + Y$, $f_{in_1}(b_{in_1}(x, z)) = z$ supposing

$$d_{X+Y}(\tilde{f}_{in_1}(x), z) = d_{X+Y}(inl\ x, z) \neq \infty$$

From $d_{X+Y}(inl\ x, z) \neq \infty$ and Equation (4.20), we know $z = inl\ x_0$ for some $x_0 \in X$, and so we must show

$$f_{in_1}(x_0) = inl\ x_0$$

which follows from Equation (4.21).

Property 1 For all $z \in X + Y$ and $c \in C$, $d_{X+Y}\left(z, b_{[g,h]}(z, c)\right) \leq d_C\left(\tilde{f}_{[g,h]}(z), c\right)$ supposing

$$d_C\left(\tilde{f}_{[g,h]}(z), c\right) \neq \infty.$$

This follows directly given that g and h are error lenses:

If $z = inl\ x$ for some $x \in \mathrm{X}$ then $d_{\mathrm{C}}\left(\tilde{f}_g(x), c\right) \neq \infty$ and we use Property 1 for g to satisfy the desired conclusion: $d_{\mathrm{X}}(x, (b_g(x, c))) \leq d_{\mathrm{C}}(\tilde{f}_g(x), c)$. Otherwise, $z = inr\ y$ for some $y \in \mathrm{Y}$ then $d_{\mathrm{C}}\left(\tilde{f}_h(y), c\right) \neq \infty$ and we use Property 1 for h.

Property 2 For all $z \in \mathrm{X} + \mathrm{Y}$ and $c \in \mathrm{C}$, $f_{[g,h]}\left(b_{[g,h]}(z, c)\right) = c$

supposing $d_{\mathrm{C}}\left(\tilde{f}_{[g,h]}(z), c\right) \neq \infty$. If $z = inl\ x$ for some $x \in \mathrm{X}$ then $d_{\mathrm{C}}\left(\tilde{f}_g(x), c\right) \neq \infty$ and we use Property 2 for g. Otherwise, $z = inr\ y$ for some $y \in \mathrm{Y}$ then $d_{\mathrm{C}}\left(\tilde{f}_h(y), c\right) \neq \infty$ and we use Property 2 for h.

To show that $[g, h] \circ in_1 = g$ (resp. $[g, h] \circ in_2 = h$), we observe that the following diagrams, by definition, commute.

$$\begin{array}{ccc} \mathrm{X} + \mathrm{Y} & \xrightarrow{f_{[g,h]}} & \mathrm{C} \\ \uparrow{\scriptstyle f_{in_1}} & \nearrow{\scriptstyle f_g} & \\ \mathrm{X} & & \end{array} \qquad \begin{array}{ccc} \mathrm{X} + \mathrm{Y} & \xrightarrow{\tilde{f}_{[g,h]}} & \mathrm{C} \\ \uparrow{\scriptstyle \tilde{f}_{in_1}} & \nearrow{\scriptstyle \tilde{f}_g} & \\ \mathrm{X} & & \end{array} \qquad \begin{array}{ccc} (\mathrm{X} + \mathrm{Y}) \times \mathrm{C} & \xrightarrow{b_{[g,h]}} & \mathrm{X} + \mathrm{Y} \\ \uparrow{\scriptstyle \tilde{f}_{in_1} \times id_{\mathrm{C}}} & & \downarrow{\scriptstyle b_{in_1}} \\ \mathrm{X} \times \mathrm{C} & \xrightarrow[b_g]{} & \mathrm{X} \end{array}$$

Uniqueness of the copairing

We check the uniqueness of copairing by showing that for any two morphisms $g_1 : \mathrm{X} \to \mathrm{C}$ and $g_2 : \mathrm{Y} \to \mathrm{C}$, if $h \circ in_1 = g_1$ and $h \circ in_2 = g_2$ for any $h : \mathrm{X} + \mathrm{Y} \to \mathrm{C}$, then $h = [g_1, g_2]$.

We detail the cases for the forward and backward map; the case for the approximation map is identical to that of the forward map.

forward map We are required to show that $f_h(z) = f_{[g_1,g_2]}(z)$ for any $z \in \mathrm{X} + \mathrm{Y}$ assuming that $f_{in_1}; f_h = f_{g_1}$ and $f_{in_2}; f_h = f_{g_2}$. The desired conclusion follows by cases on z; i.e., $z = inl\ x$ for some $x \in \mathrm{X}$ or $z = inr\ y$ for some $y \in \mathrm{Y}$.

backward map We are required to show that $b_h(z, c) = b_{[g_1,g_2]}(z, c)$ for any $z \in X + Y$ and $c \in C$. Unfolding definitions in the assumptions $b_{h \circ in_1} = b_{g_1}$ and $b_{h \circ in_2} = b_{g_2}$, we have that $b_{in_1}(x, b_h(\tilde{f}_{in_1}(x), c_1)) = b_{g_1}(x, c_1)$ for any $x \in X$ and $c_1 \in C$ and $b_{in_2}(y, b_h(\tilde{f}_{in_2}(y), c_2)) = b_{g_2}(y, c_2)$ for any $y \in Y$ and $c_2 \in C$. We proceed by cases on z.

If $z = inl\ x$ for some $x \in X$ then we are required to show that

$$b_h(inl\ x, c) = inl\ (b_{g_1}(x, c)).$$

By definition of lens composition, we have that

$$d_{X+Y}\,(inl\ x, b_h(inl\ x, c)) \neq \infty,$$

so $b_h(inl\ x, c) = inl\ x_0$ for some $x_0 \in X$. By assumption, we then have that $b_{g_1}(x, c) = b_{in_1}(x, inl\ x_0) = x_0$, from which the desired conclusion follows.

The case of $z = inr\ y$ for some $y \in Y$ is identical.

B.3 Interpreting Bean Terms

In this section of the appendix, we detail the constructions for interpreting **Bean** terms (Definition 31).

Applications of the symmetry map $s_{X,Y} : X \times Y \to Y \times X$ and 2-monoidality $m_{r,A,B} : D_R(A \otimes B) \xrightarrow{\sim} D_r(A \otimes B)$ are often elided for succinctness. Recall the discrete diagonal $t_X : X \to X \times X$ (Lemma 15), which will be used frequently in the following constructions.

Case (Var). Suppose that $\Gamma = x_0 :_{q_0} \sigma_0, \ldots, x_{i-1} :_{q_{i-1}} \sigma_{i-1}$. Define the map $[\![\Phi \mid \Gamma, x :_r \sigma \vdash x : \sigma]\!]$ as the composition

$$\pi_i \circ (\varepsilon_{[\![\sigma_0]\!]} \otimes \cdots \otimes \varepsilon_{[\![\sigma_{i-1}]\!]} \otimes \varepsilon_{[\![\sigma]\!]}) \circ (m_{0 \le r, [\![\sigma_0]\!]} \otimes \cdots \otimes m_{0 \le r, [\![\sigma_{i-1}]\!]} \otimes m_{0 \le r, [\![\sigma]\!]}),$$

where the lens π_i is the ith projection. Note that all types σ and σ_j are interpreted as metric spaces, i.e., satisfying reflexivity.

Case (DVar). Define the map $[\![\Phi, z : \alpha \mid \Gamma \vdash x : \alpha]\!]$ as the ith projection lens π_i, assuming $\Phi = z_0 : \alpha_0, \ldots, z_{i-1} : \alpha_{i-1}$. Note that all discrete types α_j and α are interpreted as discrete metric spaces, i.e., with self-distance zero.

Case (Unit). Define the map $[\![\Phi \mid \Gamma \vdash () : \text{unit}]\!]$ as the lens $\mathcal{L}_{unit}$ from a tuple $\bar{x} \in [\![\Phi \mid \Gamma]\!]$ to the singleton of the carrier in $\mathrm{I} = (\{\star\}, \underline{0})$, defined as

$$f_{unit}(\bar{x}) \triangleq \star$$

$$\tilde{f}_{unit}(\bar{x}) \triangleq \star$$

$$b_{unit}(\bar{x}, \star) \triangleq \bar{x}.$$

We verify that the triple $\mathcal{L}_{unit}$ is an error lens.

Property 1. For any $\bar{x} \in \mathrm{X}_1 \otimes \cdots \otimes \mathrm{X}_i$ we must show

$$d_{\mathrm{X}_1 \otimes \cdots \otimes \mathrm{X}_i} (\bar{x}, b_c(\bar{x}, \star)) \leq d_{\mathrm{I}} \left(\tilde{f}_{unit}(\bar{x}), \star\right)$$
$$\max(d_{\mathrm{X}_1}(x_1, x_1), \cdots, d_{\mathrm{X}_i}(x_i, x_i)) \leq 0,$$

which holds under the assumption that all types are interpreted as metric spaces with negative self distance.

Property 2. For any $\bar{x} \in \mathrm{X}_1 \otimes \cdots \otimes \mathrm{X}_i$ we have

$$f_{unit} \left(b_{unit}(\bar{x}, \star)\right) = f_{unit}(\bar{x}) = \star.$$

Case ($\otimes$ I). Given the maps

$$h_1 = [\![\Phi \mid \Gamma \vdash e : \sigma]\!] : [\![\Phi \mid \Gamma]\!] \to [\![\sigma]\!]$$

$$h_2 = [\![\Phi \mid \Delta \vdash f : \tau]\!] : [\![\Phi \mid \Delta]\!] \to [\![\tau]\!]$$

define the map $[\![\Phi \mid \Gamma, \Delta \vdash (e, f) : \sigma \otimes \tau]\!]$ as the composition

$$(h_1 \otimes h_2) \circ (t_{[\![\Phi]\!]} \otimes id_{[\![\Gamma,\Delta]\!]}),$$

where the map $t_{[\![\Phi]\!]} : [\![\Phi]\!] \to [\![\Phi]\!] \otimes [\![\Phi]\!]$ is the diagonal lens on discrete metric spaces (Lemma 15).

Case ($\otimes$ E$^\sigma$). Given the maps

$$h_1 = [\![\Phi \mid \Gamma \vdash e : \tau_1 \otimes \tau_2]\!] : [\![\Phi]\!] \otimes [\![\Gamma]\!] \to [\![\tau_1 \otimes \tau_2]\!] \tag{B.12}$$

$$h_2 = [\![\Phi \mid \Delta, x :_r \tau_1, y :_r \tau_2 \vdash f : \sigma]\!] : [\![\Phi]\!] \otimes [\![\Delta]\!] \otimes \mathrm{D}_r[\![\tau_1]\!] \otimes \mathrm{D}_r[\![\tau_2]\!] \to [\![\sigma]\!] \tag{B.13}$$

we must define a $[\![\Phi \mid r + \Gamma, \Delta \vdash \text{let } (x, y) = e \text{ in } f : \sigma]\!]$. We first define a map

$$h : \mathrm{D}_r[\![\Phi]\!] \otimes \mathrm{D}_r[\![\Gamma]\!] \otimes [\![\Delta]\!] \to [\![\sigma]\!]$$

as the composition

$$h_2 \circ ((m^{-1}_{r,[\![\tau_1]\!],[\![\tau_2]\!]} \circ \mathrm{D}_r(h_1)) \otimes (\varepsilon_{[\![\Phi]\!]} \circ m_{0 \le r,[\![\Phi]\!]}) \otimes id_{[\![\Delta]\!]}) \circ (t_{\mathrm{D}_r[\![\Phi]\!]} \otimes id_{\mathrm{D}_r[\![\Gamma]\!] \otimes [\![\Delta]\!]}).$$

Now, observing that $[\![\Phi]\!]$ is a discrete space, the set maps from the lens h define the desired lens:

$$[\![\Phi]\!] \otimes \mathrm{D}_r[\![\Gamma]\!] \otimes [\![\Delta]\!] \to [\![\sigma]\!]$$

Case ($\otimes$ E$^\alpha$). Given the maps

$$h_1 = [\![\Phi \mid \Gamma \vdash e : \alpha_1 \otimes \alpha_2]\!] : [\![\Phi]\!] \otimes [\![\Gamma]\!] \to [\![\alpha_1 \otimes \alpha_2]\!] \tag{B.14}$$

$$h_2 = [\![\Phi, x : \alpha_1, y : \alpha_2; \Delta \vdash f : \sigma]\!] : [\![\Phi]\!] \otimes [\![\alpha_1]\!] \otimes [\![\alpha_2]\!] \otimes [\![\Delta]\!] \to [\![\sigma]\!] \tag{B.15}$$

define the map $[\![\Phi \mid \Gamma, \Delta \vdash \text{let } (x, y) = e \text{ in } f : \sigma]\!]$ as the composition

$$h_2 \circ (h_1 \otimes id_{[\![\Phi]\!] \otimes [\![\Delta]\!]}) \circ (t_{[\![\Phi]\!]} \otimes id_{[\![\Gamma]\!] \otimes [\![\Delta]\!]}).$$

Case (+ E). Given the maps

$$h_1 = [\![\Phi \mid \Gamma \vdash e' : \sigma + \tau]\!] : [\![\Phi]\!] \otimes [\![\Gamma]\!] \to [\![\sigma + \tau]\!]$$

$$h_2 = [\![\Phi \mid \Delta, x :_q \sigma \vdash e : \rho]\!] : [\![\Phi]\!] \otimes [\![\Delta]\!] \otimes \mathrm{D}_q[\![\sigma]\!] \to [\![\rho]\!]$$

$$h_3 = [\![\Phi \mid \Delta, y :_q \tau \vdash f : \rho]\!] : [\![\Phi]\!] \otimes [\![\Delta]\!] \otimes \mathrm{D}_q[\![\tau]\!] \to [\![\rho]\!]$$

we require a lens $[\![\Phi \mid q + \Gamma, \Delta \vdash \textbf{case } e' \textbf{ of } (\textbf{inl } x.e \mid \textbf{inr } y.f) : \rho]\!]$. We first define a lens

$$h : \mathrm{D}_q[\![\Phi]\!] \otimes \mathrm{D}_q[\![\Gamma]\!] \otimes [\![\Delta]\!] \to [\![\sigma]\!]$$

as the composition

$$[h_2, h_3] \circ \Theta \circ ((\eta \circ D_q(h_1)) \otimes (\varepsilon_{[\![\Phi]\!]} \circ m_{0 \leq r, [\![\Phi]\!]}) \otimes id_{[\![\Delta]\!]}) \circ (t_{D_q [\![\Phi]\!]} \otimes id_{D_q [\![\Gamma]\!] \otimes [\![\Delta]\!]})$$

Now, observing that $[\![\Phi]\!]$ is a discrete space, the set maps from the lens h define the desired lens. Above, the map $\eta : D_q[\![\sigma + \tau]\!] \to D_q[\![\sigma]\!] + D_q[\![\tau]\!]$ is the identity lens, and the map $\Theta_{X,Y,Z} : X \otimes (Y + Z) \to (X \otimes Y) + (X \otimes Z)$ is given by the triple

$$f_\Theta(x, w) \triangleq \begin{cases} inl\ (x, y) & \text{if } w = inl\ y \\ inr\ (x, z) & \text{if } w = inr\ z \end{cases}$$

$$\tilde{f}_\Theta(x, w) \triangleq f_\Theta(x, w)$$

$$b_\Theta((x, w), u) \triangleq \begin{cases} (\pi_1 a, inl\ (\pi_2 a)) & \text{if } u = inl\ a \\ (\pi_1 a, inr\ (\pi_2 a)) & \text{if } u = inr\ a \\ (x, w) & \text{otherwise.} \end{cases}$$

We check that the triple

$$\Theta_{X,Y,Z} : X \otimes (Y + Z) \to (X \otimes Y) + (X \otimes Z) \triangleq (f_\Theta, \tilde{f}_\Theta, b_\Theta)$$

is well-defined.

Property 1. For any $x \in X$, $w \in Y + Z$, and $u \in (X \otimes Y) + (X \otimes Z)$ we are required to show

$$d_{X \otimes (Y+Z)}\left((x, w), b_\Theta((x, w), u)\right) \leq d_{(X \otimes Y)+(X \otimes Z)}\left(\tilde{f}_\Theta(x, w), u\right) \tag{B.16}$$

supposing

$$d_{(X \otimes Y)+(X \otimes Z)}\left(\tilde{f}_\Theta(x, w), u\right) \neq \infty. \tag{B.17}$$

From Equation (B.17), and by unfolding definitions, we have

(a) if $w = inl\ y$ for some $y \in Y$, then $u = inl\ (x_1, y_1)$ for some $(x_1, y_1) \in X \otimes Y$

(b) if $w = inr\ z$ for some $z \in Z$, then $u = inr\ (x_1, z_1)$ for some $(x_1, z_1) \in X \otimes Z$.

In both cases, the Equation (B.16) is an equality.

Property 2. For any $x \in X$, $w \in Y + Z$, and $u \in (X \otimes Y) + (X \otimes Z)$ we are required to show

$$f_\Theta(b_\Theta((x, w), u)) = u \tag{B.18}$$

supposing Equation (B.17) holds.

We consider the cases when $u = inl\ (x_1, y_1)$ for some $(x_1, y_1) \in X \otimes Y$ and when $u = inr\ (x_1, z_1)$ for some $(x_1, z_1) \in X \otimes Z$ as we did for Property 1

In the first case, we have

$$\begin{aligned} f_\Theta(b_\Theta((x, w), u)) &= f_\Theta(x_1, inl\ y_1) \\ &= inl\ (x_1, y_1). \end{aligned}$$

In the second case we have

$$\begin{aligned} f_\Theta(b_\Theta((x, w), u)) &= f_\Theta(x_1, inr\ z_1) \\ &= inr\ (x_1, z_1). \end{aligned}$$

Case (+ $\mathbf{I}_{\mathrm{L,R}}$). Given the maps

$$h_l = [\![\Phi \mid \Gamma \vdash e : \sigma]\!] : [\![\Phi]\!] \otimes [\![\Gamma]\!] \to [\![\sigma]\!]$$
$$h_r = [\![\Phi \mid \Gamma \vdash e : \sigma]\!] : [\![\Phi]\!] \otimes [\![\Gamma]\!] \to [\![\tau]\!]$$

define the maps

$$[\![\Phi \mid \Gamma \vdash \mathbf{inl}\ e : \sigma + \tau]\!] \triangleq in_1 \circ h_l$$
$$[\![\Phi \mid \Gamma \vdash \mathbf{inr}\ e : \sigma + \tau]\!] \triangleq in_2 \circ h_r.$$

Case (Let). See Section 4.3.

Case (Disc). Given the lens $[\![\Phi \mid \Gamma \vdash e : \mathrm{num}]\!]$ from the premise, we can define the map $[\![\Phi \mid \Gamma \vdash e : \mathrm{dnum}]\!]$ directly by verifying the lens conditions.

Case (DLet). Given the maps

$$h_1 = [\![\Phi \mid \Gamma \vdash e : \alpha]\!] : [\![\Phi]\!] \otimes [\![\Gamma]\!] \to [\![\alpha]\!]$$
$$h_2 = [\![\Phi, x : \alpha \mid \Delta \vdash f : \sigma]\!] : [\![\Phi]\!] \otimes [\![\alpha]\!] \otimes [\![\Delta]\!] \to [\![\sigma]\!]$$

define the map $[\![\Phi \mid \Gamma \vdash \mathrm{let}\ x = e\ \mathrm{in}\ f : \sigma]\!]$ as the composition

$$h_2 \circ (h_1 \otimes id_{[\![\Phi]\!] \otimes [\![\Delta]\!]}) \circ (t_{[\![\Phi]\!]} \otimes id_{[\![\Gamma]\!] \otimes [\![\Delta]\!]})$$

Case (Add). See Section 4.3.

Case (Sub). We proceed the same as the case for (Add). We define a lens $\mathcal{L}_{sub} : \mathrm{D}_\varepsilon(\mathrm{R}) \otimes \mathrm{D}_\varepsilon(\mathrm{R}) \to \mathrm{R}$ given by the triple

$$f_{sub}(x_1, x_2) \triangleq x_1 - x_2$$
$$\tilde{f}_{sub}(x_1, x_2) \triangleq (x_1 - x_2)e^{\delta}; \quad |\delta| \leq \varepsilon$$
$$b_{sub}((x_1, x_2), x_3) \triangleq \left(\frac{x_3 x_1}{x_1 - x_2}, \frac{x_3 x_2}{x_1 - x_2} \right).$$

We check that $\mathcal{L}_{sub} : \mathrm{D}_\varepsilon(\mathrm{R}) \otimes \mathrm{D}_\varepsilon(\mathrm{R}) \to \mathrm{R}$ is well-defined.

For any $x_1, x_2, x_3 \in \mathrm{R}$ such that

$$d_{\mathrm{R}} \left(\tilde{f}_{sub}(x_1, x_2), x_3 \right) \neq \infty. \tag{B.19}$$

holds, we need to check that $\mathcal{L}_{sub}$ satisfies the properties of an error lens. We take the distance function d_{R} as the metric given in Equation (2.7), so Equation (B.19) implies that $(x_1 - x_2)$ and x_3 are either both zero or are both non-zero and of the same sign.

Property 1. We are required to show that

$$d_{\mathrm{R}\otimes\mathrm{R}} \left((x_1, x_2), b_{sub}((x_1, x_2), x_3) \right) - \varepsilon \leq d_{\mathrm{R}} \left(\tilde{f}_{sub}(x_1, x_2), x_3 \right)$$
$$\leq d_{\mathrm{R}} \left((x_1 - x_2)e^{\delta}, x_3 \right).$$

Without loss of generality, we consider the case when

$$d_{\mathrm{R}\otimes\mathrm{R}} \left((x_1, x_2), b_{sub}((x_1, x_2), x_3) \right) = d_{\mathrm{R}} \left(x_1, \frac{x_3 x_1}{x_1 - x_2} \right);$$

that is,

$$d_{\mathrm{R}} \left(x_2, \frac{x_3 x_2}{x_1 - x_2} \right) \leq d_{\mathrm{R}} \left(x_1, \frac{x_3 x_1}{x_1 - x_2} \right).$$

Unfolding the definition of the distance function given in Equation (2.7), we are required to show

$$\left| \ln \left(\frac{x_1 - x_2}{x_3} \right) \right| \leq \left| \ln \left(\frac{x_1 - x_2}{x_3} \right) + \delta \right| + \varepsilon. \tag{B.20}$$

which holds under the assumptions of $|\delta| \leq \varepsilon$ and $0 < \varepsilon$; the proof is identical to that given for the case of the Add rule.

Property 2.

$$f_{sub}\left(b_{sub}((x_1, x_2), x_3)\right) = f_{sub}\left(\frac{x_3 x_1}{x_1 - x_2}, \frac{x_3 x_2}{x_1 - x_2}\right) = x_3.$$

Case (Mul). See Section 4.3.

Case (Div). We proceed the same as the case for (Add), with slightly different indices. We define a lens $\mathcal{L}_{div} : \mathrm{D}_{\varepsilon/2}(\mathrm{R}) \otimes \mathrm{D}_{\varepsilon/2}(\mathrm{R}) \to (\mathrm{R} + \diamond)$ given by the triple

$$f_{div}(x_1, x_2) \triangleq \begin{cases} x_1/x_2 & \text{if } x_2 \neq 0 \\ \diamond & \text{otherwise} \end{cases}$$

$$\tilde{f}_{div}(x_1, x_2) \triangleq \begin{cases} x_1 e^{\delta}/x_2 & \text{if } x_2 \neq 0; \quad |\delta| \leq \varepsilon \\ \diamond & \text{otherwise} \end{cases}$$

$$b_{div}((x_1, x_2), x) \triangleq \begin{cases} \left(\sqrt{x_1 x_2 x_3}, \sqrt{x_1 x_2 / x_3}\right) & \text{if } x = inl\ x_3 \\ (x_1, x_2) & \text{otherwise} \end{cases}.$$

We check that $\mathcal{L}_{div} : \mathrm{D}_{\varepsilon/2}(\mathrm{R}) \otimes \mathrm{D}_{\varepsilon/2}(\mathrm{R}) \to (\mathrm{R} + \diamond)$ is well-defined.

For any $x_1, x_2 \in \mathrm{R}$ and $x \in \mathrm{R} + \diamond$ such that

$$d_{\mathrm{R}+\diamond}\left(\tilde{f}_{div}(x_1, x_2), x\right) \neq \infty. \tag{B.21}$$

holds, we are required to show that $\mathcal{L}_{div}$ satisfies the properties of an error lens. From Equation (B.21) and again assuming the distance function is given by Equation (2.7), we know $x = inl\ x_3$ for some $x_3 \in \mathrm{R}$, $x_2 \neq 0$, and x_1/x_2 and x_3 are either both zero or both non-zero and of the same sign; this guarantees that the backward map (containing square roots) is indeed well defined.

Property 1. We need to show

$$\begin{aligned} d_{\mathrm{R}\otimes\mathrm{R}}\left((x_1, x_2), b_{div}((x_1, x_2), x)\right) - \varepsilon/2 &\leq d_{\mathrm{R}+\diamond}\left(\tilde{f}_{div}(x_1, x_2), x\right) \\ &\leq d_{\mathrm{R}}\left(\frac{x_1}{x_2} e^{\delta}, x_3\right). \end{aligned} \tag{B.22}$$

Unfolding the definition of the distance function (Equation (2.7)), we have

$$\begin{aligned} d_{\mathbb{R}\otimes\mathbb{R}}\left((x_1,x_2), b_{div}((x_1,x_2),x_3)\right) &= d_{\mathbb{R}}\left(x_1, x_1\sqrt{x_1x_2x_3}\right) \\ &= d_{\mathbb{R}}\left(x_2, x_2\sqrt{x_1x_2/x_3}\right) \\ &= \frac{1}{2}\left|\ln\left(\frac{x_1}{x_2x_3}\right)\right|, \end{aligned}$$

and so we are required to show

$$\frac{1}{2}\left|\ln\left(\frac{x_1}{x_2x_3}\right)\right| \leq \left|\ln\left(\frac{x_1}{x_2x_3}\right) + \delta\right| + \frac{1}{2}\varepsilon,$$

which holds under the assumptions of $|\delta| \leq \varepsilon$ and $0 < \varepsilon$; the proof is identical to that given for the case of the Mul rule.

Property 2.

$$f_{div}\left(b_{div}((x_1,x_2),x_3)\right) = f_{div}\left(\sqrt{x_1x_2x_3}, \sqrt{x_1x_2/x_3}\right) = x_3.$$

Case (DMul). We proceed similarly as for (Add). We define a lens $\mathcal{L}_{dmul} : (\mathbb{R}^{\alpha} \otimes D_{\varepsilon}\mathbb{R}) \to \mathbb{R}$ given by the triple

$$\begin{aligned} f_{dmul}(x_1,x_2) &\triangleq x_1x_2 \\ \tilde{f}_{dmul}(x_1,x_2) &\triangleq x_1x_2e^{\delta}; \quad |\delta| \leq \varepsilon \\ b_{dmul}((x_1,x_2),x_3) &\triangleq (x_1, x_3/x_1). \end{aligned}$$

We check that $\mathcal{L}_{dmul} : \mathbb{R}^{\alpha} \otimes D_{\varepsilon}\mathbb{R} \to \mathbb{R}$ is well-defined.

For any $x_1, x_2, x_3 \in \mathbb{R}$ such that

$$d_{\mathbb{R}}\left(\tilde{f}_{dmul}(x_1,x_2), x_3\right) \neq \infty. \tag{B.23}$$

holds, we need to check the that $\mathcal{L}_{dmul}$ satisfies the properties of an error lens. We again take the distance function $d_{\mathbb{R}}$ as the metric given in Equation (2.7), so Equation (B.23) implies that (x_1x_2) and x_3 are either both zero or are both non-zero and of the same sign; this guarantees that the backward map (containing square roots) is indeed well defined.

Property 1. We are required to show

$$d_{\mathrm{R}^{\alpha}\otimes \mathrm{D}_{\varepsilon}\mathrm{R}}\left((x_1, x_2), b_{dmul}((x_1, x_2), x_3)\right) \le d_{\mathrm{R}}\left(\tilde{f}_{dmul}(x_1, x_2), x_3\right) \le d_{\mathrm{R}}\left(x_1 x_2 e^{\delta}, x_3\right)$$

Unfolding the definition of the distance function (Equation (2.7)), we have

$$d_{\mathrm{R}^{\alpha}\otimes \mathrm{D}_{\varepsilon}\mathrm{R}}\left((x_1, x_2), b_{dmul}((x_1, x_2), x_3)\right) = \max\left(d_{\alpha}(x_1, x_1), d_{\mathrm{R}}(x_2, x_3/x_1) - \varepsilon\right) = \left|\ln\left(\frac{x_1 x_2}{x_3}\right)\right| - \varepsilon,$$

and so we are required to show

$$\left|\ln\left(\frac{x_1 x_2}{x_3}\right)\right| \le \left|\ln\left(\frac{x_1 x_2}{x_3}\right) + \delta\right| + \varepsilon \tag{B.24}$$

which holds under the assumptions of $|\delta| \le \varepsilon$ and $0 < \varepsilon$; the proof is identical to that given in the Add rule.

Property 2.

$$f_{dmul}\left(b_{dmul}((x_1, x_2), x_3)\right) = f_{dmul}\left(x_1, x_3/x_1\right) = x_3.$$

B.4 Interpreting Λ_S Terms

This appendix provides the detailed constructions of the interpretation of Λ_S terms for Definition 32. The interpretation of terms is defined over the typing derivations for Λ_S given in Figure 4.4. For each case, the ideal interpretation $(\!|-|\!)_{id}$ is constructed explicitly, but the construction for $(\!|-|\!)_{ap}$ is nearly identical, requiring only that the forgetful functor U_{ap} is used in place of U_{id}.

Applications of the symmetry map $s_{\mathrm{X},\mathrm{Y}} : \mathrm{X} \times \mathrm{Y} \to \mathrm{Y} \times \mathrm{X}$ are elided for succinctness. The diagonal map $d_{\mathrm{X}} : \mathrm{X} \to \mathrm{X} \times \mathrm{X}$ on **Set** is used frequently and is not elided.

Case (Var). Define the maps $(\!|\Phi, \Gamma, x : \sigma, \Delta \vdash x : \sigma|\!)_{id}$ and $(\!|\Phi, \Gamma, x : \sigma, \Delta \vdash x : \sigma|\!)_{ap}$ in **Set** as the appropriate projection π_i.

Case (Unit). Define the set maps $(\!|\Phi, \Gamma \vdash () : \text{unit}|\!)_{id}$ and $(\!|\Phi, \Gamma \vdash () : \text{unit}|\!)_{ap}$ as the constant function returning the value $\star$.

Case (Const). Define the maps $(\!|\Phi, \Gamma \vdash k : \text{num}|\!)_{id}$ and $(\!|\Phi, \Gamma \vdash k : \text{num}|\!)_{ap}$ in **Set** as the constant function taking points in $(\!|\Phi, \Gamma|\!)$ to the value $k \in \mathrm{R}$.

Case ($\otimes$ I). Given the maps

$$h_1 = (\!|\Phi, \Gamma \vdash e : \sigma|\!)_{id} : (\!|\Phi|\!) \times (\!|\Gamma|\!) \to (\!|\sigma|\!)$$

$$h_2 = (\!|\Phi, \Delta \vdash f : \tau|\!)_{id} : (\!|\Phi|\!) \times (\!|\Delta|\!) \to (\!|\tau|\!)$$

in **Set**, define the map $(\!|\Phi, \Gamma, \Delta \vdash (e, f) : \sigma \otimes \tau|\!)_{id}$ as

$$(d_{(\!|\Phi|\!)}, id_{(\!|\Gamma|\!)}, id_{(\!|\Delta|\!)}); (h_1, h_2)$$

Case ($\otimes$ E). Given the maps

$$h_1 = (\!|\Phi, \Gamma \vdash e : \tau_1 \otimes \tau_2|\!)_{id} : (\!|\Phi|\!) \times (\!|\Gamma|\!) \to (\!|\tau_1|\!) \times (\!|\tau_2|\!)$$

$$h_2 = (\!|\Phi, \Delta, x : \tau_1, y : \tau_2 \vdash f : \sigma|\!)_{id} : (\!|\Phi|\!) \times (\!|\Delta|\!) \times (\!|\tau_1|\!) \times (\!|\tau_2|\!) \to (\!|\sigma|\!)$$

in **Set**, define $(\!|\Phi, \Gamma, \Delta \vdash \text{let } (x, y) = e \text{ in } f : \sigma|\!)_{ap}$ as

$$(d_{(\!|\Phi|\!)}, id_{(\!|\Gamma|\!)}, id_{(\!|\Delta|\!)}); (h_1, id_{(\!|\Phi|\!) \times (\!|\Delta|\!)}); h_2$$

Case (+ E). Given the maps

$$h_1 = (\!|\Phi, \Gamma \vdash e' : \sigma + \tau|\!)_{id} : (\!|\Phi|\!) \times (\!|\Gamma|\!) \to (\!|\sigma + \tau|\!)$$

$$h_2 = (\!|\Phi, \Delta, x : \sigma \vdash e : \rho|\!)_{id} : (\!|\Phi|\!) \times (\!|\Delta|\!) \times (\!|\sigma|\!) \to (\!|\rho|\!)$$

$$h_3 = (\!|\Phi, \Delta, y : \tau \vdash f : \rho|\!)_{id} : (\!|\Phi|\!) \times (\!|\Delta|\!) \times (\!|\tau|\!) \to (\!|\rho|\!)$$

in **Set**, define $(\!|\Phi, \Gamma, \Delta \vdash \textbf{case } e' \textbf{ of } (\textbf{inl } x.e \mid \textbf{inr } y.f) : \rho|\!)_{id}$ as

$$(d_{(\!|\Phi|\!)}, id_{(\!|\Gamma|\!) \times (\!|\Delta|\!)}); (h_1, id_{(\!|\Phi|\!) \times (\!|\Delta|\!)}); \Theta^{\mathrm{S}}_{(\!|\Phi|\!) \times (\!|\Delta|\!), (\!|\sigma|\!), (\!|\tau|\!)}; [h_2, h_3]$$

where $\Theta^{\mathrm{S}}_{\mathrm{X,Y,Z}}$ is a map in **Set**:

$$\Theta_{\mathrm{X,Y,Z}} : \mathrm{X} \times (\mathrm{Y} + \mathrm{Z}) \to (\mathrm{X} \times \mathrm{Y}) + (\mathrm{X} \times \mathrm{Z})$$

Case (+ $\mathbf{I}_L$). Given the map

$$h = (\!|\Phi, \Gamma \vdash e : \sigma|\!)_{id} : (\!|\Phi|\!) \times (\!|\Gamma|\!) \longrightarrow (\!|\sigma|\!)$$

in **Set**, define the map

$$(\!|\Phi, \Gamma \vdash \mathbf{inl}\ e : \sigma + \tau|\!)_{id}$$

as the composition

$$h; in_1$$

Case (+ I_R). Given the map

$$h = (\!|\Phi, \Gamma \vdash e : \sigma|\!)_{id} : (\!|\Phi|\!) \times (\!|\Gamma|\!) \longrightarrow (\!|\sigma|\!)$$

in **Set**, define the map

$$(\!|\Phi, \Gamma \vdash \mathbf{inr}\ e : \sigma + \tau|\!)_{id}$$

as the composition

$$h; in_r$$

Case (Let). Given the maps

$$h_1 = (\!|\Phi, \Gamma \vdash e : \sigma|\!)_{id} : (\!|\Phi|\!) \times (\!|\Gamma|\!) \longrightarrow (\!|\sigma|\!)$$

$$h_2 = (\!|\Phi, \Delta, x : \sigma \vdash f : \tau|\!)_{id} : (\!|\Phi|\!) \times (\!|\Delta|\!) \times (\!|\sigma|\!) \longrightarrow (\!|\tau|\!)$$

in **Set**, define the map

$$(\!|\Phi, \Gamma, \Delta \vdash \text{let } x = e \text{ in } f : \tau|\!)_{id}$$

as the composition

$$(t_{(\!|\Phi|\!)}, id_{(\!|\Gamma|\!) \times (\!|\Delta|\!)}); (h_1, id_{(\!|\Phi|\!) \times (\!|\Delta|\!)}); h_2$$

Case (Op). Given the maps

$$h_1 = (\!|\Phi, \Gamma \vdash e : \text{num}|\!)_{id} : (\!|\Phi|\!) \times (\!|\Gamma|\!) \longrightarrow (\!|\text{num}|\!)$$

$$h_2 = (\!|\Phi, \Delta \vdash f : \text{num}|\!)_{id} : (\!|\Phi|\!) \times (\!|\Delta|\!) \longrightarrow (\!|\text{num}|\!)$$

in **Set**, define the map

$$(\!|\Phi, \Gamma, \Delta \vdash \mathbf{op}\ e\ f : \mathrm{num}|\!)_{id}$$

as the composition

$$d_{(\!|\Phi|\!)}; (h_1, h_2); \mathrm{U}_{id}\mathcal{L}_{op} = d_{(\!|\Phi|\!)}; (h_1, h_2); f_{op}$$

for $\mathbf{op} \in \{\mathrm{add}, \mathrm{sub}, \mathrm{mul}, \mathrm{dmul}\}$.

Case (Div). Given the maps

$$h_1 = (\!|\Phi, \Gamma \vdash e : \mathrm{num}|\!)_{id} : (\!|\Phi|\!) \times (\!|\Gamma|\!) \longrightarrow (\!|\mathrm{num}|\!)$$

$$h_2 = (\!|\Phi, \Delta \vdash f : \mathrm{num}|\!)_{id} : (\!|\Phi|\!) \times (\!|\Delta|\!) \longrightarrow (\!|\mathrm{num}|\!)$$

in **Set**, define the map

$$(\!|\Phi, \Gamma, \Delta \vdash \mathbf{div}\ e\ f : \mathrm{num} + \mathbf{err}|\!)_{id}$$

as the composition

$$d_{(\!|\Phi|\!)}; (h_1, h_2); \mathrm{U}_{id}\mathcal{L}_{div} = d_{(\!|\Phi|\!)}; (h_1, h_2); f_{div}$$

B.5 Details for Soundness

This appendix provides details for the proofs in Section 4.4, which presents the main backward error soundness theorem for **Bean**. A detailed proof of the main theorem is provided in Appendix B.6. In this appendix, we provide the details for auxiliary results. The full typing relation for Λ_S is defined by the rules in Figure 4.4, and the operational semantics for Λ_S are given in Figure 4.5.

Lemma 16. *Let $\Phi \mid \Gamma \vdash e : \tau$ be a well-typed term in **Bean**. Then there is a derivation of $\Phi, \Gamma^\circ \vdash e : \tau$ in Λ_S.*

Proof. The proof of Lemma 16 follows by induction on the **Bean** derivation $\Phi \mid \Gamma \vdash e : \tau$. Most cases are immediate by application of the corresponding Λ_S rule. The rules for primitive operations

require application of the Λ_S (Var) rule. We demonstrate the derivation for the case of the (Add) rule:

Case (Add). Given a **Bean** derivation of

$$\Phi \mid \Gamma, x :_{\varepsilon+r_1} \text{num}, y :_{\varepsilon+r_2} \text{num} \vdash \text{add } x\ y : \text{num}$$

we are required to show a Λ_S derivation of

$$\Phi, \Gamma, x : \text{num}, y : \text{num} \vdash \text{add } x\ y : \text{num}$$

which follows by application of the Var rule for Λ_S:

$$\text{(Add)}\ \dfrac{\text{(Var)}\ \dfrac{}{\Phi, \Gamma, x : \text{num} \vdash x : \text{num}} \quad \text{(Var)}\ \dfrac{}{y : \text{num} \vdash y : \text{num}}}{\Phi, \Gamma, x : \text{num}, y : \text{num} \vdash \text{add } x\ y : \text{num}}$$

□

Lemma 18. *Let $\Phi \mid \Gamma \vdash e : \sigma$ be a **Bean** program. Then we have*

$$U_{id}[\![\Phi \mid \Gamma \vdash e : \sigma]\!] = (\!|\Phi, \Gamma^\circ \vdash e : \sigma|\!)_{id} \quad \textit{and} \quad U_{ap}[\![\Phi \mid \Gamma \vdash e : \sigma]\!] = (\!|\Phi, \Gamma^\circ \vdash e : \sigma|\!)_{ap}.$$

Proof. The proof of Lemma 18 follows by induction on the structure of the **Bean** derivation $\Phi \mid \Gamma \vdash e : \sigma$. We detail here the cases of pairing for the ideal semantics.

$$U_{id}[\![\Phi \mid \Gamma \vdash e : \sigma]\!] = (\!|\Phi, \Gamma \vdash e : \sigma|\!)_{id}$$

Case (Var).

$$\begin{aligned} U_{id}[\![\Phi \mid \Gamma, x :_r \sigma \vdash x : \sigma]\!] &= U_{id}(\varepsilon_{[\![\sigma]\!]} \circ m_{0 \le r, [\![\sigma]\!]} \circ \pi_i) && \text{(Definition 31)} \\ &= \pi_i && \text{(Definition of } U_{id}\text{)} \\ &= (\!|\Phi, \Gamma^\circ, x : \sigma \vdash x : \sigma|\!)_{id} && \text{(Definition 32)} \end{aligned}$$

Case (DVar).

$$\mathrm{U}_{id}[\![\Phi, z : \alpha \mid \Gamma \vdash z : \sigma]\!] = \pi_i \quad \text{(Definition 31)}$$

$$= (\!|\Phi, x : \sigma, \Gamma^\circ \vdash x : \sigma|\!)_{id} \quad \text{(Definition 32)}$$

Case (Unit).

$$\mathrm{U}_{id}[\![\Phi \mid \Gamma \vdash () : \mathrm{unit}]\!] = f_{unit} \quad \text{(Definition 31)}$$

$$= (\!|\Phi \mid \Gamma^\circ \vdash () : \mathrm{unit}|\!)_{id} \quad \text{(Definition 32)}$$

Case ($\otimes$ I). From the induction hypothesis we have

$$\mathrm{U}_{id}(h_1) = (\!|\Phi, \Gamma \vdash e : \sigma|\!)_{id}$$

$$\mathrm{U}_{id}(h_2) = (\!|\Phi, \Delta \vdash f : \tau|\!)_{id}$$

We conclude as follows:

$$\mathrm{U}_{id}[\![\Phi \mid \Gamma, \Delta \vdash (e, f) : \sigma \otimes \tau]\!] = \mathrm{U}_{id}(t_{[\![\Phi]\!]} \otimes id_{[\![\Gamma \otimes \Delta]\!]}); \mathrm{U}_{id}(h_1 \otimes h_2) \quad \text{(Definition 31)}$$

$$= (t_{(\!|\Phi|\!)}, id_{(\!|\Gamma|\!) \times (\!|\Delta|\!)}); (\mathrm{U}_{id}(h_1), \mathrm{U}_{id}(h_2)) \quad \text{(Definition of } \mathrm{U}_{id}\text{)}$$

$$= (\!|\Phi, \Gamma, \Delta \vdash (e, f) : \sigma \otimes \tau|\!)_{id} \quad \text{(IH \& Definition 32)}$$

Case ($\otimes$ E$_\sigma$). From the induction hypothesis we have

$$\mathrm{U}_{id}(h_1) = (\!|\Phi, \Gamma^\circ \vdash e : \tau_1 \otimes \tau_2|\!)_{id}$$

$$\mathrm{U}_{id}(h_2) = (\!|\Phi, \Delta^\circ, x : \tau_1, y : \tau_2 \vdash f : \sigma|\!)_{id}$$

We conclude with the following:

$$\mathrm{U}_{id}[\![\Phi \mid r + \Gamma, \Delta \vdash \mathrm{let}\ (x, y) = e \ \mathrm{in}\ f : \sigma]\!]$$

$$= \mathrm{U}_{id}\left(h_2 \circ ((m^{-1}_{r, [\![\tau_1]\!], [\![\tau_2]\!]} \circ \mathrm{D}_r(h_1)) \otimes (\varepsilon_{[\![\Phi]\!]} \circ m_{0 \le r, [\![\Phi]\!]}) \otimes id_{[\![\Delta]\!]}) \circ (t_{\mathrm{D}_r [\![\Phi]\!]} \otimes id_{\mathrm{D}_r [\![\Gamma]\!] \otimes [\![\Delta]\!]})\right)$$

(Definition 31)

$$= (t(\!|\Phi|\!), id_{(\!|\Gamma|\!) \times (\!|\Delta|\!)}); (\mathrm{U}_{id}(h_1), id_{(\!|\Phi|\!)}, id_{(\!|\Delta|\!)}); \mathrm{U}_{id}(h_2) \quad \text{(Definition of } \mathrm{U}_{id}\text{)}$$

$$= (t(\!|\Phi|\!), id_{(\!|\Gamma|\!) \times (\!|\Delta|\!)}); (\mathrm{U}_{id}(h_1), id_{(\!|\Phi|\!) \times (\!|\Delta|\!)}); \mathrm{U}_{id}(h_2) \quad \text{(Definition of } \mathrm{U}_{id}\text{)}$$

$$= (\!|\Phi, \Gamma^\circ, \Delta^\circ \vdash \mathrm{let}\ (x, y) = e \ \mathrm{in}\ f : \sigma|\!)_{id} \quad \text{(IH \& Definition 32)}$$

Case ($\otimes$ $\mathbf{E}_\alpha$) From the induction hypothesis we have

$$\mathrm{U}_{id}(h_1) = (\!|\Phi, \Gamma^\circ \vdash e : \tau_1 \otimes \tau_2|\!)_{id}$$

$$\mathrm{U}_{id}(h_2) = (\!|\Phi, \Delta^\circ, x : \tau_1, y : \tau_2 \vdash f : \sigma|\!)_{id}$$

We conclude with the following:

$$[\![\Phi \mid \Gamma, \Delta \vdash \text{let } (x, y) = e \text{ in } f : \sigma]\!] = h_2 \circ (h_1 \otimes id_{[\![\phi]\!] \otimes [\![\Delta]\!]}) \circ (t_{[\![\Phi]\!]} \otimes id_{[\![\Gamma]\!] \otimes [\![\Delta]\!]})$$

(Definition 31)

$$= (\mathrm{U}_{id}(h_1), id_{(\!|\phi|\!) \times (\!|\Delta|\!)}); \mathrm{U}_{id}(h_2) \qquad \text{(Definition of } \mathrm{U}_{id})$$

$$= (\mathrm{U}_{id}(h_1), id_{(\!|\Delta|\!)}); \mathrm{U}_{id}(h_2) \qquad \text{(Definition of } \mathrm{U}_{id})$$

$$= (\!|\Phi, \Gamma^\circ, \Delta^\circ \vdash \text{let } (x, y) = e \text{ in } f : \sigma|\!)_{id}$$

(IH &Definition 32)

Case (Let). From the induction hypothesis we have

$$\mathrm{U}_{id}(h_1) = (\!|\Phi, \Gamma^\circ \vdash e : \tau|\!)_{id}$$

$$\mathrm{U}_{id}(h_2) = (\!|\Phi, \Delta^\circ, x : \tau \vdash f : \sigma|\!)_{id}.$$

We conclude with the following:

$$\mathrm{U}_{id}[\![\Phi \mid r + \Gamma, \Delta \vdash \text{let } x = e \text{ in } f : \sigma]\!]$$

$$= \mathrm{U}_{id}\big(h_2 \circ (\mathrm{D}_r(h_1) \otimes (\varepsilon_{[\![\Phi]\!]} \circ m_{0 \le r, [\![\Phi]\!]}) \otimes id_{[\![\Delta]\!]})$$

$$\circ\, (m_{r, [\![\Phi]\!], [\![\Gamma]\!]} \otimes id_{\mathrm{D}_r [\![\Phi]\!]} \otimes id_{[\![\Delta]\!]}) \circ (t_{\mathrm{D}_r [\![\Phi]\!]} \otimes id_{\mathrm{D}_r [\![\Gamma]\!] \otimes [\![\Delta]\!]})\big)$$

(Definition 31)

$$= (t_{(\!|\Phi|\!)}, id_{(\!|\Gamma|\!) \times (\!|\Delta|\!)}); (\mathrm{U}_{id}(h_1), id_{(\!|\Phi|\!)}, id_{(\!|\Delta|\!)}); \mathrm{U}_{id}(h_2) \qquad \text{(Definition of } \mathrm{U}_{id})$$

$$= (\!|\Phi^\circ, (r + \Gamma)^\circ, \Delta^\circ \vdash \text{let } x = e \text{ in } f : \sigma|\!) \qquad \text{(IH \& Definition 32)}$$

Case (+ E). From the induction hypothesis, we have

$$\mathrm{U}_{id}(h_1) = (\!|\Phi, \Gamma^\circ \vdash e' : \sigma + \tau|\!)_{id}$$

$$\mathrm{U}_{id}(h_2) = (\!|\Phi, \Delta^\circ, x :_q \sigma \vdash e : \rho|\!)_{id}$$

$$\mathrm{U}_{id}(h_3) = (\!|\Phi, \Delta^\circ, y :_q \tau \vdash f : \rho|\!)_{id}$$

We conclude with the following:

$$\mathrm{U}_{id}[\![\Phi \mid q + \Gamma, \Delta \vdash \textbf{case } e' \textbf{ of } (\textbf{inl } x.e \mid \textbf{inr } y.f) : \sigma]\!]$$

$$= \mathrm{U}_{id}\big([h_2, h_3] \circ \Theta \circ ((\eta \circ \mathrm{D}_q(h_1)) \otimes (\varepsilon_{[\![\Phi]\!]} \circ m_{0 \leq r, [\![\Phi]\!]}) \otimes id_{[\![\Delta]\!]}) \circ (t_{\mathrm{D}_q[\![\Phi]\!]} \otimes id_{\mathrm{D}_q[\![\Gamma]\!] \otimes [\![\Delta]\!]})\big)$$

(Definition 31)

$$= (t(\!|\Phi|\!), id_{(\!|\Gamma|\!) \times (\!|\Delta|\!)}); (\mathrm{U}_{id}(h_1), id_{(\!|\Phi|\!)}, id_{(\!|\Delta|\!)}); \mathrm{U}_{id}(\Theta); [\mathrm{U}_{id}(h_2), \mathrm{U}_{id}(h_3)]$$

(Definition of U_{id})

$$= (\!|\Phi, \Gamma^\circ, \Delta^\circ \vdash \textbf{case } e' \textbf{ of } (\textbf{inl } x.e \mid \textbf{inr } y.f) : \sigma|\!)$$ (IH & Definition 32)

Case (+ I). From the induction hypothesis, we have

$$\mathrm{U}_{id}(h) = (\!|\Phi, \Gamma^\circ \vdash e : \sigma|\!)$$

$$\mathrm{U}_{id}([\![\Phi \mid \Gamma \vdash \textbf{inl } e : \sigma + \tau]\!]) = \mathrm{U}_{id}\,(in_1 \circ h)$$ (Definition 31)

$$= \mathrm{U}_{id}(h); \mathrm{U}_{id}(in_1)$$ (Definition of U_{id})

$$= (\!|\Phi, \Gamma \vdash \textbf{inl } e : \sigma + \tau|\!)_{id}$$ (IH & Definition 32)

Case (Add). From Definition 31 we have

$$[\![\Phi \mid \Gamma, x :_\varepsilon \text{num}, y :_\varepsilon \text{num} \vdash \text{add } x\ y : \text{num}]\!]$$

$$= \pi_i \circ \cdots \circ (id_{[\![\Phi]\!] \otimes [\![\Gamma]\!]} \otimes \mathrm{L}_{add}) \circ (id_{[\![\Phi]\!] \otimes [\![\Gamma]\!]} \otimes m_{\varepsilon \leq \varepsilon + q, [\![\text{num}]\!]} \otimes m_{\varepsilon \leq \varepsilon + r, [\![\text{num}]\!]}),$$

We conclude as follows:

$$\mathrm{U}_{id}[\![\Phi \mid \Gamma, x :_\varepsilon \text{num}, y :_\varepsilon \text{num} \vdash \text{add } x\ y : \text{num}]\!] = \pi_i; (id_{(\!|\Phi|\!) \otimes (\!|\Gamma|\!)}, f_{add})$$ (Definition of U_{id})

$$= (\!|\Phi, \Gamma^\circ, x : \text{num}, y : \text{num} \vdash \text{add } x\ y : \text{num}|\!)_{id}$$

(Definition 32)

The cases for the remaining arithmetic operations are nearly identical to the case for **Add**. ◻

Substitution

Theorem 6. *Let $\Gamma \vdash e : \tau$ be a well-typed Λ_S term. Then for any well-typed substitution $\bar{\gamma} \vDash \Gamma$ of closed values, there is a derivation $\emptyset \vdash e[\bar{\gamma}/dom(\Gamma)] : \tau$.*

Proof. By induction on the structure of the derivation $\Gamma \vdash e : \tau$. The cases for (Var), (Unit), (Const), and (+ I) are trivial; Λ_S is a simple first-order language and the remaining cases are routine.

Case (⊗ I). We have a well-typed substitution of closed values $\bar{\gamma} \vDash dom(\Phi, \Gamma, \Delta)$ and it is straightforward to show that the induction hypothesis yields the premises needed for applying the typing rule (⊗ I). The desired conclusion then follows from the definition of substitution.

Case (⊗ E). We are required to show

$$\emptyset \vdash (\text{let } (x, y) = e \text{ in } f)[\bar{\gamma}/dom(\Phi, \Gamma, \Delta)] : \sigma$$

given the well-typed substitution of closed values $\bar{\gamma} \vDash dom(\Phi, \Gamma, \Delta)$. From $\bar{\gamma}$ we derive a substitution $\bar{\gamma}' \vDash \Phi, \Gamma$, and from the induction hypothesis on the left premise we have $\emptyset \vdash e[\bar{\gamma}'/dom(\Phi, \Gamma)] : \tau_1 \otimes \tau_2$; by inversion on this hypothesis, we derive a substitution which allows us to use the induction hypothesis for the right premise. This provides the premises needed to apply the typing rule (⊗ E). The desired conclusion then follows from the definition of substitution.

Case (+ E). We are required to show

$$\emptyset \vdash (\textbf{case } e' \textbf{ of } (\textbf{inl } x.e \mid \textbf{inr } y.e))[\bar{\gamma}/dom(\Phi, \Gamma, \Delta)] : \rho$$

given the well-typed substitution of closed values $\bar{\gamma} \vDash dom(\Phi, \Gamma, \Delta)$. From $\bar{\gamma}$ we derive a substitution $\bar{\gamma}' \vDash \Phi, \Gamma$, and from the induction hypothesis on the left premise we have

$\emptyset \vdash e'[\bar{\gamma}'/dom(\Phi,\Gamma)] : \sigma + \tau$; we first apply inversion to this hypothesis and then reason by cases to derive a substitution which allows us to use the induction hypothesis for the right premise. This provides the premises needed to apply the typing rule (+ E). The desired conclusion then follows from the definition of substitution.

Case (Let). We are required to show

$$\emptyset \vdash (\text{let } x = e \text{ in } f)[\bar{\gamma}/dom(\Phi,\Gamma,\Delta)] : \sigma$$

given a well-typed substitution of closed values $\bar{\gamma} \vdash dom(\Phi,\Gamma,\Delta)$. From $\bar{\gamma}$ we derive a substitution $\bar{\gamma}' \vDash \Phi,\Gamma$, and from the induction hypothesis on the left premise we have $\emptyset \vdash e[\bar{\gamma}'/dom(\Phi,\Gamma)]$; by inversion on this hypothesis, we derive a substitution which allows us to use the induction hypothesis for the right premise. This provides the premises needed to apply the typing rule (Let). The desired conclusion then follows from the definition of substitution.

Case (Op). We have a well-typed substitution of closed values $\bar{\gamma} \vDash dom(\Phi,\Gamma,\Delta)$ and it is straightforward to show that the induction hypothesis yields the premises needed for applying the typing rule (Op). The desired conclusion then follows from the definition of substitution.

Case (Div). We have a well-typed substitution of closed values $\bar{\gamma} \vDash dom(\Phi,\Gamma,\Delta)$ and it is straightforward to show that the induction hypothesis yields the premises needed for applying the typing rule (Div). The desired conclusion then follows from the definition of substitution.

□

Soundness of $(\!|-|\!)_{id}$

In this section of the appendix, we prove the soundness of our denotational semantics. Namely, we show that our interpretation of Λ_S (Definition 32) respects the operational semantics given in Figure 4.5.

Applications of the symmetry map $s_{X,Y} : X \times Y \to Y \times X$ are elided for succinctness. Recall the diagonal map $d_X : X \to X \times X$ on **Set**, which is used frequently in the interpretation of Λ_S.

Theorem 8. *Let $\Gamma \vdash e : \tau$ be a well-typed Λ_S term. Then for any well-typed substitution of closed values $\bar{\gamma} \vDash \Gamma$, if $e[\bar{\gamma}/dom(\Gamma)] \Downarrow_{id} v$ for some value v, then $(\!|\Gamma \vdash e : \tau|\!)_{id}(\!|\bar{\gamma}|\!)_{id} = (\!|v|\!)_{id}$ (and similarly for $\Downarrow_{ap}$ and $(\!|-|\!)_{ap}$).*

Proof. By induction on the structure of the Λ_S derivations $\Gamma \vdash e : \tau$. The cases for (Var), (Unit), (Const), and (+ I) are trivial. In each case we apply inversion on the step relation to obtain the premise for the induction hypothesis.

Case ($\otimes$ I). We are required to show

$$(\!|\Phi, \Gamma, \Delta \vdash (e, f) : \sigma \otimes \tau|\!)_{id}(\!|\bar{\gamma}|\!)_{id} = (\!|(u, v)|\!)_{id}$$

for some well-typed closed substitution $\bar{\gamma} \vDash \Phi, \Gamma, \Delta$ and value (u, v) such that

$$(e, f)[\bar{\gamma}'/dom(\Phi, \Gamma, \Delta)] \Downarrow_{id} (u, v)$$

From $\bar{\gamma}$ we derive the substitutions $\bar{\gamma}' \vDash \Phi$, $\bar{\gamma}_1 \vDash \Gamma$, and $\bar{\gamma}_2 \vDash \Delta$. By inversion on the step relation we then have

$$e[\bar{\gamma}', \bar{\gamma}_1/dom(\Phi, \Gamma)] \Downarrow_{id} u$$

$$f[\bar{\gamma}', \bar{\gamma}_2/dom(\Phi, \Delta)] \Downarrow_{id} v$$

We conclude as follows:

$$(\!|\Phi, \Gamma, \Delta \vdash (e, f) : \sigma \otimes \tau|\!)_{id}(\!|\bar{\gamma}|\!)_{id} = \big((d_{(\!|\Phi|\!)}, id_{(\!|\Gamma|\!)}, id_{(\!|\Delta|\!)}); ((\!|\Phi, \Gamma \vdash e : \sigma|\!)_{id}, (\!|\Phi, \Delta \vdash f : \tau|\!)_{id})\big)(\!|\bar{\gamma}|\!)_{id}$$

(Definition 32)

$$= \big((\!|\Phi, \Gamma \vdash e : \sigma|\!)_{id}, (\!|\Phi, \Delta \vdash f : \tau|\!)_{id}\big)\big((\!|\bar{\gamma}'|\!), (\!|\bar{\gamma}_1|\!), (\!|\bar{\gamma}'|\!), (\!|\bar{\gamma}_2|\!)\big)$$

(Definition of $d_{(\!|\Phi|\!)}$)

$$= ((\!|u|\!)_{id}, (\!|v|\!)_{id}) \qquad \text{(IH)}$$

Case ($\otimes$ E). We are required to show

$$(\!|\Phi, \Gamma, \Delta \vdash \text{let } (x, y) = e \text{ in } f : \sigma|\!)_{id} (\!|\bar{\gamma}|\!)_{id} = (\!|w|\!)_{id}$$

for some well-typed closed substitution $\bar{\gamma} \vDash \Phi, \Gamma, \Delta$ and value w such that

$$(\text{let } (x, y) = e \text{ in } f)[\bar{\gamma}/dom(\Phi, \Gamma, \Delta)] \Downarrow_{id} w$$

From $\bar{\gamma}$ we derive the substitutions $\bar{\gamma}' \vDash \Phi$, $\bar{\gamma}_1 \vDash \Gamma$, and $\bar{\gamma}_2 \vDash \Delta$. By inversion on the step relation we then have

$$e[\bar{\gamma}', \bar{\gamma}_1/dom(\Phi, \Gamma)] \Downarrow_{id} (u, v)$$
$$f[\bar{\gamma}', \bar{\gamma}_2/dom(\Phi, \Delta)][u/x][v/y] \Downarrow_{id} w$$

We conclude as follows:

$$
\begin{aligned}
&(\!|\Phi, \Gamma, \Delta \vdash \text{let } (x, y) = e \text{ in } f : \sigma|\!)_{id} (\!|\bar{\gamma}|\!)_{id} \\
&= \big((d_{(\!|\Phi|\!)}, id_{(\!|\Gamma|\!)}, id_{(\!|\Delta|\!)}); (h_1, id_{(\!|\Phi|\!)\times(\!|\Delta|\!)}); h_2\big)(\!|\bar{\gamma}|\!)_{id} && \text{(Definition 32)} \\
&= \big((h_1, id_{(\!|\Phi|\!)\times(\!|\Delta|\!)}); h_2\big)\big((\!|\bar{\gamma}'|\!), (\!|\bar{\gamma}_1|\!), (\!|\bar{\gamma}'|\!), (\!|\bar{\gamma}_2|\!)\big) && \text{(Definition of } d_{(\!|\Phi|\!)}) \\
&= \big((\!|\Phi, \Delta, x : \tau_1, y : \tau_2 \vdash f : \sigma|\!)_{id}\big)\big((\!|\bar{\gamma}'|\!), (\!|\bar{\gamma}_2|\!), (\!|u|\!), (\!|v|\!)\big) && \text{(IH)} \\
&= (\!|w|\!)_{id} && \text{(IH)}
\end{aligned}
$$

Case (+ E). We are required to show

$$(\!|\Phi, \Gamma, \Delta \vdash \textbf{case } e' \textbf{ of } (\textbf{inl } x.e \mid \textbf{inr } y.f) : \rho|\!)_{id} (\!|\bar{\gamma}|\!)_{id} = (\!|w|\!)_{id}$$

for some well-typed closed substitution $\bar{\gamma} \vDash \Phi, \Gamma, \Delta$ and value w such that

$$(\textbf{case } e' \textbf{ of } (\textbf{inl } x.e \mid \textbf{inr } y.f))[\bar{\gamma}/dom(\Phi, \Gamma, \Delta)] \Downarrow_{id} w$$

We consider the case when $e' = \textbf{inl } e_1$ for some $e_1 : \sigma$. From $\bar{\gamma}$ we derive the substitutions $\bar{\gamma}' \vDash \Phi$, $\bar{\gamma}_1 \vDash \Gamma$, and $\bar{\gamma}_2 \vDash \Delta$. By inversion on the step relation we then have

$$e'[\bar{\gamma}', \bar{\gamma}_1/dom(\Phi, \Gamma)] \Downarrow_{id} \textbf{inl } v$$
$$e[\bar{\gamma}', \bar{\gamma}_2/dom(\Phi, \Delta)][v/x] \Downarrow_{id} w$$

We conclude as follows:

$$
\begin{aligned}
&(\!|\Phi, \Gamma, \Delta \vdash \textsf{case } e' \textsf{ of } (\textsf{inl } x.e \mid \textsf{inr } y.f) : \rho|\!)_{id}(\!|\bar{\gamma}|\!)_{id} \\
&= \big((d_{(\!|\Phi|\!)}, id_{(\!|\Gamma|\!)\times(\!|\Delta|\!)}); (h_1, id_{(\!|\Phi|\!)\times(\!|\Delta|\!)}); \Theta^{S}_{(\!|\Phi|\!)\times(\!|\Delta|\!),(\!|\sigma|\!),(\!|\tau|\!)}; [h_2, h_3]\big)(\!|\bar{\gamma}|\!)_{id} && \text{(Definition 32)} \\
&= \big((h_1, id_{(\!|\Phi|\!)\times(\!|\Delta|\!)}); \Theta^{S}_{(\!|\Phi|\!)\times(\!|\Delta|\!),(\!|\sigma|\!),(\!|\tau|\!)}; [h_2, h_3]\big)\big((\!|\bar{\gamma}'|\!), (\!|\bar{\gamma}_1|\!), (\!|\bar{\gamma}'|\!), (\!|\bar{\gamma}_2|\!)\big) && \text{(Definition of } d_{(\!|\Phi|\!)_{id}}) \\
&= \big(\Theta^{S}_{(\!|\Phi|\!)\times(\!|\Delta|\!),(\!|\sigma|\!),(\!|\tau|\!)}; [h_2, h_3]\big)\big((\!|\bar{\gamma}'|\!), (\!|\bar{\gamma}_2|\!), (\!|\textbf{inl } v|\!)\big) && \text{(IH)} \\
&= (\!|w|\!)_{id} && \text{(IH)}
\end{aligned}
$$

Case (Let). We are required to show

$$(\!|\Gamma, \Delta \vdash \text{let } x = e \text{ in } f : \tau|\!)_{id}(\!|\bar{\gamma}|\!)_{id} = (\!|v|\!)_{id}$$

for some well-typed closed substitution $\bar{\gamma} \vDash \Phi, \Gamma, \Delta$ and value w such that

$$(\text{let } x = e \text{ in } f)[\bar{\gamma}/dom(\Phi, \Gamma, \Delta)] \Downarrow_{id} v$$

From $\bar{\gamma}$ we derive the substitutions $\bar{\gamma}' \vDash \Phi$, $\bar{\gamma}_1 \vDash \Gamma$, and $\bar{\gamma}_2 \vDash \Delta$. By inversion on the step relation we then have

$$e[\bar{\gamma}', \bar{\gamma}_1/dom(\Phi, \Gamma)] \Downarrow_{id} u$$

$$f[\bar{\gamma}', \bar{\gamma}_2/dom(\Phi, \Delta)][u/x] \Downarrow_{id} v$$

We conclude as follows:

$$
\begin{aligned}
&(\!|\Phi, \Gamma, \Delta \vdash \text{let } x = e \text{ in } f : \sigma|\!)_{id}(\!|\bar{\gamma}|\!)_{id} \\
&= \big((d_{(\!|\Phi|\!)}, id_{(\!|\Gamma|\!)}, id_{(\!|\Delta|\!)}); (h_1, id_{(\!|\Phi|\!)\times(\!|\Delta|\!)}); h_2\big)(\!|\bar{\gamma}|\!)_{id} && \text{(Definition 32)} \\
&= \big((h_1, id_{(\!|\Phi|\!)\times(\!|\Delta|\!)}); h_2\big)\big((\!|\bar{\gamma}'|\!), (\!|\bar{\gamma}_1|\!), (\!|\bar{\gamma}'|\!), (\!|\bar{\gamma}_2|\!)\big) && \text{(Definition of } d_{(\!|\Phi|\!)}) \\
&= \big((\!|\Phi, \Delta, x : \tau_1 \vdash f : \sigma|\!)_{id}\big)\big((\!|\bar{\gamma}'|\!), (\!|\bar{\gamma}_2|\!), (\!|u|\!)\big) && \text{(IH)} \\
&= (\!|v|\!)_{id} && \text{(IH)}
\end{aligned}
$$

Case (Op). We are required to show

$$(\!|\Phi, \Gamma, \Delta \vdash \textbf{Op } e\ f : \text{num}|\!)_{id}(\!|\bar{\gamma}|\!)_{id} = (\!|f_{op}(k_1, k_2)|\!)_{id}$$

for some well-typed closed substitution $\gamma \vDash \Phi, \Gamma, \Delta$ and value $f_{op}(k_1, k_2)$ such that

$$(\Phi, \Gamma, \Delta \vdash \mathbf{Op}\ e\ f)[\bar{\gamma}/dom(\Phi, \Gamma, \Delta)] \Downarrow_{id}\ f_{op}(k_1, k_2)$$

From $\bar{\gamma}$ we derive the substitutions $\bar{\gamma}' \vDash \Phi$, $\bar{\gamma}_1 \vDash \Gamma$, and $\bar{\gamma}_2 \vDash \Delta$. By inversion on the step relation we then have

$$e[\bar{\gamma}', \bar{\gamma}_1/dom(\Phi, \Gamma)] \Downarrow_{id} k_1$$
$$f[\bar{\gamma}', \bar{\gamma}_2/dom(\Phi, \Delta)][u/x] \Downarrow_{id} k_2$$

We conclude as follows:

$$\begin{aligned}
&(\!|\Phi, \Gamma, \Delta \vdash \mathbf{Op}\ e\ f : \mathrm{num}|\!)_{id}(\!|\bar{\gamma}|\!)_{id} \\
&= \left(d_{(\!|\Phi|\!)}; (h_1, h_2); f_{op}\right)(\!|\bar{\gamma}|\!)_{id} && \text{(Definition 32)} \\
&= \left((h_1, h_2); f_{op}\right)\left((\!|\bar{\gamma}'|\!), (\!|\bar{\gamma}_1|\!), (\!|\bar{\gamma}'|\!), (\!|\bar{\gamma}_2|\!)\right) && \text{(Definition of } d_{(\!|\Phi|\!)}\text{)} \\
&= (\!|f_{op}(k_1, k_2)|\!)_{id} && \text{(IH)}
\end{aligned}$$

Case (Div). Identical to the proof for (Op).

□

Adequacy of $(\!|-|\!)_{id}$

In this section of the appendix, we prove the computational adequacy of our denotational semantics. Namely, we show that if two Λ_S terms are equal under our denotational semantics (Definition 32), then they will evaluate to the same value under our operational semantics given in Figure 4.5.

Theorem 9. *Let $\Gamma \vdash e : \tau$ be a well-typed Λ_S term. Then for any well-typed substitution of closed values $\bar{\gamma} \vDash \Gamma$, if $(\!|\Gamma \vdash e : \tau|\!)_{id}(\!|\bar{\gamma}|\!)_{id} = (\!|v|\!)_{id}$ for some value v, then $e[\bar{\gamma}/dom(\Gamma)] \Downarrow_{id} v$ (and similarly for $\Downarrow_{ap}$ and $(\!|-|\!)_{ap}$).*

Proof. The proof follows directly by cases on e. Many cases are immediate and the remaining cases, given that Λ_S is deterministic, follow by substitution (Theorem 6) and normalization (Theorem 7). We show two representative cases.

Case. Given $\Gamma, x : \sigma, \Delta \vdash x : \sigma$ and $(\!|\Gamma, x : \sigma, \Delta \vdash x : \sigma|\!)_{id}(\!|\bar{\gamma}|\!)_{id} = (\!|v|\!)_{id}$ for some value v and some well-typed substitution $\bar{\gamma} \vDash \Gamma, x : \sigma, \Delta$ we are required to show

$$x[\bar{\gamma}/dom(\Gamma, x : \sigma, \Delta)] \Downarrow_{id} v$$

which follows by substitution (Theorem 6) and normalization (Theorem 7).

Case. Given $\Gamma, \Delta \vdash \text{let } x = e \text{ in } f : \tau$ and $(\!|\Gamma, \Delta \vdash \text{let } x = e \text{ in } f : \tau|\!)_{id}(\!|\bar{\gamma}|\!)_{id} = (\!|w|\!)_{id}$ for some value w and some well-typed derivation $\bar{\gamma} \vDash \Gamma, \Delta$ we are required to show

$$(\text{let } x = e \text{ in } f)[\bar{\gamma}/dom(\Gamma, \Delta)] \Downarrow_{id} w$$

which follows by substitution (Theorem 6) and normalization (Theorem 7).

□

B.6 Proof of Backward Error Soundness

This appendix provides a detailed proof of the main backward error soundness theorem for **Bean** (Theorem 10).

Theorem 10. *Let $\Phi \mid x_1 :_{r_1} \sigma_1, \cdots, x_n :_{r_n} \sigma_n = \Gamma \vdash e : \sigma$ be a well-typed **Bean** term. Then for any well-typed substitutions $\bar{p} \vDash \Phi$ and $\bar{k} \vDash \Gamma^\circ$, if*

$$e[\bar{p}/dom(\Phi)][\bar{k}/dom(\Gamma)] \Downarrow_{ap} v$$

for some value v, then the well-typed substitution $\bar{l} \vDash \Gamma^\circ$ exists such that

$$e[\bar{p}/dom(\Phi)][\bar{l}/dom(\Gamma)] \Downarrow_{id} v,$$

and $d_{[\![\sigma_i]\!]}(k_i, l_i) \leq r_i$ for each $k_i \in \bar{k}$ and $l_i \in \bar{l}$.

Proof. From the lens semantics (Definition 31) of **Bean** we have the triple

$$[\![\Phi \mid \Gamma \vdash e : \sigma]\!] = (f, \tilde{f}, b) : [\![\Phi]\!] \otimes [\![\Gamma]\!] \to [\![\sigma]\!].$$

Then, using the backward map b, we can define the tuple of vectors of values $(\bar{s}, \bar{l}) \triangleq b((\bar{p}, \bar{k}), v)$ such that $\bar{s} \models \Phi$ and $\bar{l} \models \Gamma$.

From the second property of backward error lenses we then have

$$f(\!|(\bar{s}, \bar{l})|\!)_{id} = f(\!|b((\bar{p}, \bar{k}), v)|\!)_{id} = v.$$

We can now show a backward error result, i.e., $\tilde{f}(\!|(\bar{p}, \bar{k})|\!)_{ap} = f(\!|(\bar{s}, \bar{l})|\!)_{id}$:

$$\begin{aligned}
(\!|\Phi, \Gamma \vdash e : \sigma|\!)_{ap}(\!|(\bar{p}, \bar{k})|\!)_{ap} &= \mathrm{U}_{ap}[\![\Phi \mid \Gamma \vdash e : \sigma]\!](\!|(\bar{p}, \bar{k})|\!)_{ap} && \text{(Lemma 18)} \\
&= \tilde{f}(\!|(\bar{p}, \bar{k})|\!)_{ap} && \text{(Definition 31)} \\
&= v && \text{(Theorem 8)} \\
&= f(\!|(\bar{s}, \bar{l})|\!)_{id}
\end{aligned}$$

From the first property of error lenses we have $d_{[\![\Phi]\!] \otimes [\![\Gamma]\!]}\left((\bar{p}, \bar{k}), b((\bar{p}, \bar{k}), v)\right) \leq d_{[\![\sigma]\!]}\left(\tilde{f}(\bar{p}, \bar{k}), v\right)$ so long as

$$d_{[\![\sigma]\!]}\left(\tilde{f}(\bar{p}, \bar{k}), v\right) = d_{[\![\sigma]\!]}(v, v) \neq \infty. \tag{B.25}$$

If the base numeric type is interpreted as a metric space with a standard distance function, then $d_{[\![\sigma]\!]}(v, v) \neq \infty$ for any type σ, and so Equation (B.25) is satisfied.

Unfolding definitions, and using the fact that $\tilde{f}(\!|(\bar{p}, \bar{k})|\!)_{ap} = v$ from above, we have

$$\max\left(d_{[\![\Phi]\!]}(\bar{p}, \bar{s}), d_{[\![\Gamma]\!]}(\bar{k}, \bar{l})\right) \leq d_{[\![\sigma]\!]}(v, v) \tag{B.26}$$

From Equation (B.26) we can conclude two things. First, using the definition of the distance function on discrete metric spaces, we can conclude $\bar{p} = \bar{s}$: the discrete variables carry no backward error.

Second, for linear variables we can derive the required backward error bound:

$$\max\left(d_{\llbracket\sigma_1\rrbracket}(k_1, l_1) - r_1, \ldots, d_{\llbracket\sigma_n\rrbracket}(k_n, l_n) - r_n\right) \leq 0.$$

□

B.7 Type Checking Algorithm

This appendix, authored by Laura Zielinski, presents the type-checking algorithm for **Bean** as described in Section 4.6.1, along with proofs of its soundness and completeness. First, we give the full type checking algorithm in Figure B.2. Recall that algorithm calls are written as $\Phi \mid \Gamma^\bullet; e \Longrightarrow \Gamma; \sigma$ where $\Gamma^\bullet$ is a linear context skeleton, e is a **Bean** program, Γ is, intuitively, the *minimal* linear context required to type e such that $\overline{\Gamma} \sqsubseteq \Gamma^\bullet$, and σ is the type of e. Note that we only require Φ to contain the discrete variables used in the program and we do nothing more; thus, it is not returned by the algorithm. We do require that discrete and linear contexts are always disjoint, and we will denote linear variables by x and y and discrete variables by z. Finally, we define the *max* of two linear contexts, $\max\{\Gamma, \Delta\}$, to have domain $\operatorname{dom}\Gamma \cup \operatorname{dom}\Delta$ and, if $x :_q \sigma \in \Gamma$ and $x :_r \sigma \in \Delta$, then $x :_{\max\{q,r\}} \sigma \in \max\{\Gamma, \Delta\}$.

Before we give proofs of Theorem 11 and Theorem 12, we must prove two lemmas about type system and algorithm weakening. Intuitively, type system weakening says that if we can derive the type of a program from a context Γ, then we can also derive the same program from a larger context Δ which subsumes Γ.

Lemma 26 (Type System Weakening)**.** If $\Phi \mid \Gamma \vdash e : \sigma$ and $\Gamma \sqsubseteq \Delta$, then $\Phi \mid \Delta \vdash e : \sigma$.

Proof. Suppose $\Phi \mid \Gamma \vdash e : \sigma$. We proceed by induction on the final typing rule applied and show some representative cases below.

$$\frac{}{\Phi \mid \Gamma^\bullet, x : \sigma; x \Rightarrow \{x :_0 \sigma\}; \sigma}\ (\text{Var}) \qquad \frac{}{\Phi, z : \alpha \mid \Gamma^\bullet; z \Rightarrow \emptyset; \alpha}\ (\text{DVar})$$

$$\frac{\Phi \mid \Gamma^\bullet; e \Rightarrow \Gamma_1; \sigma \qquad \Phi \mid \Gamma^\bullet; f \Rightarrow \Gamma_2; \tau \qquad \operatorname{dom} \Gamma_1 \cap \operatorname{dom} \Gamma_2 = \emptyset}{\Phi \mid \Gamma^\bullet; (e, f) \Rightarrow \Gamma_1, \Gamma_2; \sigma \otimes \tau}\ (\otimes \text{ I})$$

$$\frac{}{\Phi \mid \Gamma^\bullet; () \Rightarrow \emptyset; \text{unit}}\ (\text{Unit})$$

$$\frac{\Phi \mid \Gamma^\bullet; e \Rightarrow \Gamma_1; \tau_1 \otimes \tau_2 \qquad \Phi \mid \Gamma^\bullet, x : \tau_1, y : \tau_2; f \Rightarrow \Gamma_2; \sigma \qquad \operatorname{dom} \Gamma_1 \cap \operatorname{dom} \Gamma_2 = \emptyset}{\Phi \mid \Gamma^\bullet; \text{let } (x, y) = e \text{ in } f \Rightarrow (r + \Gamma_1), \Gamma_2 \setminus \{x, y\}; \sigma}\ (\otimes \text{ E}_\sigma)$$

where $x, y \notin \Gamma^\bullet$ and $r = \max\{r_1, r_2\}$ if at least one of $x :_{r_1} \tau_1, y :_{r_2} \tau_2 \in \Gamma_2$ (else $r = 0$)

$$\frac{\Phi \mid \Gamma^\bullet; e \Rightarrow \Gamma_1; \alpha_1 \otimes \alpha_2 \qquad \Phi, z_1 : \alpha_1, z_2 : \alpha_2 \mid \Gamma^\bullet; f \Rightarrow \Gamma_2; \sigma \qquad \operatorname{dom} \Gamma_1 \cap \operatorname{dom} \Gamma_2 = \emptyset}{\Phi \mid \Gamma^\bullet; \text{dlet } (z_1, z_2) = e \text{ in } f \Rightarrow \Gamma_1, \Gamma_2; \sigma}\ (\otimes \text{ E}_\alpha)$$

where $z_1, z_2 \notin \Phi$

$$\frac{\Phi \mid \Gamma^\bullet; e' \Rightarrow \Gamma_1; \sigma + \tau \qquad \Phi \mid \Gamma^\bullet, x : \sigma; e \Rightarrow \Gamma_2; \rho \qquad \Phi \mid \Gamma^\bullet, y : \tau; f \Rightarrow \Gamma_3; \rho \qquad \begin{array}{r} \operatorname{dom} \Gamma_1 \cap \operatorname{dom} \Gamma_2 \\ = \operatorname{dom} \Gamma_1 \cap \operatorname{dom} \Gamma_3 \\ = \emptyset \end{array}}{\Phi \mid \Gamma^\bullet; \text{case } e' \text{ of } (x.e \mid y.f) \Rightarrow (q + \Gamma_1), \max\{\Gamma_2 \setminus \{x\}, \Gamma_3 \setminus \{y\}\}; \rho}\ (+ \text{ E})$$

where $x, y \notin \Gamma^\bullet$ and $q = \max\{q_1, q_2\}$ if at least one of $x :_{q_1} \sigma \in \Gamma_2$ or $y :_{q_2} \tau \in \Gamma_3$ (else $q = 0$)

$$\frac{\Phi \mid \Gamma^\bullet; e \Rightarrow \Gamma; \sigma}{\Phi \mid \Gamma^\bullet; \textbf{inl}_{\,\tau}\ e \Rightarrow \Gamma; \sigma + \tau}\ (+ \text{ I}_\text{L}) \qquad \frac{\Phi \mid \Gamma^\bullet; e \Rightarrow \Gamma; \tau}{\Phi \mid \Gamma^\bullet; \textbf{inr}_{\,\sigma}\ e \Rightarrow \sigma + \tau}\ (+ \text{ I}_\text{R})$$

$$\frac{\Phi \mid \Gamma^\bullet; e \Rightarrow \Gamma_1; \tau \qquad \Phi \mid \Gamma^\bullet, x : \tau; f \Rightarrow \Gamma_2; \sigma \qquad \operatorname{dom} \Gamma_1 \cap \operatorname{dom} \Gamma_2 = \emptyset}{\Phi \mid \Gamma^\bullet; \text{let } x = e \text{ in } f \Rightarrow (r + \Gamma_1), \Gamma_2 \setminus \{x\}; \sigma}\ (\text{Let})$$

where $x \notin \Gamma^\bullet$ and $x :_r \sigma \in \Gamma_2$ (else $r = 0$)

$$\frac{\Phi \mid \Gamma^\bullet; e \Rightarrow \Gamma; \text{num}}{\Phi \mid \Gamma^\bullet; !e \Rightarrow \Gamma; \text{dnum}}\ (\text{Disc})$$

$$\frac{\Phi \mid \Gamma^\bullet; e \Rightarrow \Gamma_1; \text{dnum} \qquad \Phi, z : \text{dnum} \mid \Gamma^\bullet; f \Rightarrow \Gamma_2; \sigma \qquad \operatorname{dom} \Gamma_1 \cap \operatorname{dom} \Gamma_2 = \emptyset}{\Phi \mid \Gamma^\bullet; \text{dlet } z = e \text{ in } f \Rightarrow \Gamma_1, \Gamma_2; \sigma}\ (\text{DLet})$$

where $z \notin \Phi$

$$\frac{}{\Phi \mid \Gamma^\bullet, x : \text{num}, y : \text{num}; \{\text{add}, \text{sub}\}\ x\ y \Rightarrow \{x :_\varepsilon \text{num}, y :_\varepsilon \text{num}\}; \text{num}}\ (\text{Add, Sub})$$

$$\frac{}{\Phi \mid \Gamma^\bullet, x : \text{num}, y : \text{num}; \text{mul}\ x\ y \Rightarrow \{x :_{\varepsilon/2} \text{num}, y :_{\varepsilon/2} \text{num}\}; \text{num}}\ (\text{Mul})$$

$$\frac{}{\Phi \mid \Gamma^\bullet, x : \text{num}, y : \text{num}; \{\text{add}, \text{sub}\}\ x\ y \Rightarrow \{x :_\varepsilon \text{num}, y :_\varepsilon \text{num}\}; \text{num} + \textbf{err}}\ (\text{Div})$$

$$\frac{}{\Phi, z : \text{dnum} \mid \Gamma^\bullet, x : \text{num}; \text{dmul}\ z\ x \Rightarrow \{x :_\varepsilon \text{num}\}; \text{num}}\ (\text{DMul})$$

Figure B.2: Type checking algorithm for **Bean**.

Case (Var). Suppose the last rule applied was

$$\Phi \mid \Gamma, x :_r \sigma \vdash x : \sigma.$$

Let Δ be a context such that $(\Gamma, x :_r \sigma) \sqsubseteq \Delta$. Thus, $x :_q \sigma \in \Delta$ where $r \leq q$. By the same rule, $\Phi \mid \Delta \vdash x : \sigma$.

Case ($\otimes$ I). Suppose the last rule applied was

$$\Phi \mid \Gamma, \Delta \vdash (e, f) : \sigma \otimes \tau$$

and thus, we also have that

$$\Phi \mid \Gamma \vdash e : \sigma \text{ and } \Phi \mid \Delta \vdash f : \tau.$$

Let Λ be a context such that $(\Gamma, \Delta) \sqsubseteq \Lambda$. As Γ and Δ are disjoint, we can split Λ into the contexts Γ_1 and Δ_1 such that $\Gamma \sqsubseteq \Gamma_1$ and $\Delta \sqsubseteq \Delta_1$. By out inductive hypothesis, it follows that

$$\Phi \mid \Gamma_1 \vdash e : \sigma \text{ and } \Phi \mid \Delta_1 \vdash f : \tau.$$

By the same rule, we conclude that

$$\Phi \mid \Gamma_1, \Delta_1 \vdash (e, f) : \sigma \otimes \tau.$$

Case ($\otimes$ E$_\sigma$). Suppose the last rule applied was

$$\Phi \mid r + \Gamma, \Delta \vdash \text{let } (x, y) = e \text{ in } f : \sigma.$$

Let Λ be a context such that $(r + \Gamma, \Delta) \sqsubseteq \Lambda$ and $x, y \notin \operatorname{dom} \Lambda$. As before, split Λ into contexts Γ_1 and Δ_1 such that $(r + \Gamma) \sqsubseteq \Gamma_1$ and $\Delta \sqsubseteq \Delta_1$ but where $\operatorname{dom} \Gamma = \operatorname{dom} \Gamma_1$. Now, for each $x \in \operatorname{dom} \Gamma_1$, we have that $x :_q \sigma \in \Gamma_1$ where $r \leq q$. Therefore, we can define the context $-r + \Gamma_1$ which subtracts r from the error bound of every variable in Γ_1, and hence $\Gamma \sqsubseteq (-r + \Gamma_1)$. Finally, use our inductive hypothesis to get that

$$\Phi \mid (-r + \Gamma_1) \vdash e : \tau_1 \otimes \tau_2 \text{ and } \Phi \mid \Delta_1, x :_r \tau_1, y :_r \tau_2 \vdash f : \sigma$$

and we can apply the same rule to get our conclusion.

Case (Add). Suppose the last rule applied was

$$\Phi \mid \Gamma, x :_{\varepsilon+r_1} \text{num}, y :_{\varepsilon+r_2} \text{num} \vdash \text{add } x\ y : \text{num}$$

Let Δ be a context such that $(\Gamma, x :_{\varepsilon+r_1} \text{num}, y :_{\varepsilon+r_2} \text{num}) \sqsubseteq \Delta$. Hence, $x :_{q_1} \text{num}, y :_{q_2} \text{num} \in \Delta$ where $\varepsilon + r_1 \leq q_1$ and $\varepsilon + r_2 \leq q_2$. Rewrite $q_1 = \varepsilon + (q_1 - \varepsilon)$ and $q_2 = \varepsilon + (q_2 - \varepsilon)$ and apply the same rule.

□

Similarly, algorithm weakening says that if we pass a context skeleton $\Gamma^\bullet$ into the algorithm and it infers context Γ, then if we pass in a larger skeleton $\Delta^\bullet$, the algorithm will still infer context Γ. (Here, we extend the notion of subcontexts to context skeletons, where $\Gamma^\bullet \sqsubseteq \Delta^\bullet$ if $\Gamma^\bullet \subseteq \Delta^\bullet$.) This is because the algorithm discards unused variables from the context.

Lemma 27 (Type Checking Algorithm Weakening)**.** If $\Phi \mid \Gamma^\bullet; e \Rightarrow \Gamma; \sigma$ and $\Gamma^\bullet \sqsubseteq \Delta^\bullet$, then $\Phi \mid \Delta^\bullet; e \Rightarrow \Gamma; \sigma$.

Proof. Suppose that $\Phi \mid \Gamma^\bullet; e \Rightarrow \Gamma; \sigma$. We proceed by induction on the final algorithmic step applied and show some representative cases below.

Case (Var). Suppose the last step applied was

$$\Phi \mid \Gamma^\bullet, x : \sigma; x \Rightarrow \{x :_0 \sigma\}; \sigma.$$

Let $\Delta^\bullet$ be a context skeleton such that $(\Gamma^\bullet, x : \sigma) \sqsubseteq \Delta^\bullet$. Thus, $x : \sigma \in \Delta^\bullet$ so we can apply the same rule.

Case ($\otimes$ $\mathbf{E}_\sigma$). Suppose the last step applied was

$$\Phi \mid \Gamma^\bullet; \text{let } (x, y) = e \text{ in } f \Rightarrow (r + \Gamma_1), \Gamma_2 \setminus \{x, y\}; \sigma.$$

Let $\Delta^\bullet$ be a context skeleton such that $\Gamma^\bullet \sqsubseteq \Delta^\bullet$ and $x, y \notin \Delta^\bullet$. By induction, we have that

$$\Phi \mid \Delta^\bullet; e \Rightarrow \Gamma_1; \tau_1 \otimes \tau_2 \text{ and } \Phi \mid \Delta^\bullet, x : \tau_1, y : \tau_2; f \Rightarrow \Gamma_2; \sigma$$

and $\mathrm{dom}\,\Gamma_1 \cap \mathrm{dom}\,\Gamma_2 = \emptyset$. By the same rule, we conclude that

$$\Phi \mid \Delta^\bullet; \mathrm{let}\ (x, y) = e \ \mathrm{in}\ f \Rightarrow (r + \Gamma_1), \Gamma_2 \setminus \{x, y\}; \sigma.$$

□

Finally, we give proofs of algorithmic soundness and completeness. Soundness states that if the algorithm returns a linear context Γ, then we can use Γ to derive the program using **Bean**'s type system.

Theorem 11. *If* $\Phi \mid \Gamma^\bullet; e \Rightarrow \Gamma; \sigma$*, then* $\overline{\Gamma} \sqsubseteq \Gamma^\bullet$ *and the derivation* $\Phi \mid \Gamma \vdash e : \sigma$ *exists.*

Proof. Suppose that $\Phi \mid \Gamma^\bullet; e \Rightarrow \Gamma; \sigma$. We proceed by induction on the final algorithmic step applied and show some representative cases below. We use the fact that if Γ, Δ are disjoint, then $\overline{\Gamma, \Delta} = \overline{\Gamma}, \overline{\Delta}$.

Case (Var). Suppose the last step applied was

$$\Phi \mid \Gamma^\bullet, x : \sigma; x \Rightarrow \{x :_0 \sigma\}; \sigma.$$

By the typing rule (Var$_\sigma$), we have that $\Phi \mid \{x :_0 \sigma\} \vdash x : \sigma$. Moreover, $\overline{\{x :_0 \sigma\}} \sqsubseteq (\Gamma^\bullet, x : \sigma)$.

Case ($\otimes$ I). Suppose the last step applied was

$$\Phi \mid \Gamma^\bullet; (e, f) \Rightarrow \Gamma_1, \Gamma_2; \sigma \otimes \tau$$

where

$$\Phi \mid \Gamma^\bullet; e \Rightarrow \Gamma_1; \sigma \text{ and } \Phi \mid \Gamma^\bullet; f \Rightarrow \Gamma_2; \sigma$$

and $\mathrm{dom}\,\Gamma_1 \cap \mathrm{dom}\,\Gamma_2 = \emptyset$. By our inductive hypothesis, we have that $\Phi \mid \Gamma_1 \vdash e : \sigma$ and $\Phi \mid \Gamma_2 \vdash f : \tau$. Therefore, we can apply the typing rule ($\otimes$ I) to get that

$$\Phi \mid \Gamma_1, \Gamma_2 \vdash (e, f) : \sigma \otimes \tau.$$

Finally, as $\overline{\Gamma_1} \sqsubseteq \Gamma^\bullet$ and $\overline{\Gamma_2} \sqsubseteq \Gamma^\bullet$, we have that $\overline{\Gamma_1, \Gamma_2} \sqsubseteq \Gamma^\bullet$.

Case ($\otimes$ E$_\sigma$). Suppose the last step applied was

$$\Phi \mid \Gamma^\bullet; \text{let } (x, y) = e \text{ in } f \Rightarrow (r + \Gamma_1), \Gamma_2 \setminus \{x, y\}; \sigma$$

By induction, we have that

$$\Phi \mid \Gamma_1 \vdash e : \tau_1 \otimes \tau_2 \text{ and } \Phi \mid \Gamma_2 \vdash f : \sigma,$$

where x, y may be in $\operatorname{dom} \Gamma_2$. Let $\Delta = \Gamma_2 \setminus \{x, y\}$. Since r is defined to be the maximum of the bounds on x, y if they exist in Γ_2, we have that $\Gamma_2 \sqsubseteq (\Delta, x :_r \tau_1, y :_r \tau_2)$. From Lemma 26, it follows that

$$\Phi \mid \Delta, x :_r \tau_1, y :_r \tau_2 \vdash f : \sigma.$$

Thus, we can apply the typing rule ($\otimes$ E$_\sigma$) to conclude that

$$\Phi \mid r + \Gamma_1, \Delta \vdash \text{let } (x, y) = e \text{ in } f : \sigma.$$

Finally, as $\overline{\Gamma_1} \sqsubseteq \Gamma^\bullet$ and $\overline{\Gamma_2} \sqsubseteq (\Gamma^\bullet, x : \tau_1, y : \tau_2)$, we have that

$$\overline{r + \Gamma_1, \Delta} = \overline{\Gamma_1, \Gamma_2 \setminus \{x, y\}} = \overline{\Gamma_1}, \overline{\Gamma_2 \setminus \{x, y\}} \sqsubseteq \Gamma^\bullet.$$

Case (+ E). Suppose the last step applied was

$$\Phi \mid \Gamma^\bullet; \text{case } e' \text{ of } (x.e \mid y.f) \Rightarrow (q + \Gamma_1), \max\{\Gamma_2 \setminus \{x\}, \Gamma_3 \setminus \{y\}\}; \rho.$$

By induction, we have that

$$\Phi \mid \Gamma_1 \vdash e' : \sigma + \tau \text{ and } \Phi \mid \Gamma_2 \vdash e : \rho \text{ and } \Phi \mid \Gamma_3 \vdash f : \rho.$$

Let $\Delta = \max\{\Gamma_2 \setminus \{x\}, \Gamma_3 \setminus \{y\}\}$, and we still have that $\operatorname{dom} \Gamma_1 \cap \operatorname{dom} \Delta = \emptyset$. By Lemma 26, it follows that

$$\Phi \mid \Delta, x :_q \sigma \vdash e : \rho \text{ and } \Phi \mid \Delta, y :_q \tau \vdash f : \rho$$

by weakening the bounds on x and y to q. Thus, we can apply typing rule (+ E) to conclude that

$$\Phi \mid q + \Gamma_1, \Delta \vdash \textbf{case } e' \textbf{ of } (\textbf{inl } x.e \mid \textbf{inr } y.f) : \rho.$$

Moreover, as $\overline{\Gamma_1} \sqsubseteq \Gamma^\bullet$ and $\overline{\Gamma_2} \sqsubseteq (\Gamma^\bullet, x : \sigma)$ and $\overline{\Gamma_3} \sqsubseteq (\Gamma^\bullet, y : \tau)$, we have that

$$\overline{q + \Gamma_1, \Delta} = \overline{\Gamma_1}, \overline{\max\{\Gamma_2 \setminus \{x\}, \Gamma_3 \setminus \{y\}\}} \sqsubseteq \Gamma^\bullet.$$

Case (Add). Suppose the last step applied was

$$\Phi \mid \Gamma^\bullet, x : \text{num}, y : \text{num}; \text{add } x\ y \Rightarrow \{x :_\varepsilon \text{num}, y :_\varepsilon \text{num}\}; \text{num}.$$

By the typing rule (Add) we have that

$$\Phi \mid \{x :_\varepsilon \text{num}, y :_\varepsilon \text{num}\} \vdash \text{add } x\ y : \text{num}.$$

□

Conversely, completeness says that if from Γ we can derive the type of a program e, then inputting $\overline{\Gamma}$ and e into the algorithm will yield a valid output.

Theorem 12. *If $\Phi \mid \Gamma \vdash e : \sigma$ is a valid derivation in* ***Bean****, then there exists a context $\Delta \sqsubseteq \Gamma$ such that $\Phi \mid \overline{\Gamma}; e \Rightarrow \Delta; \sigma$.*

Proof. Suppose that $\Phi \mid \Gamma \vdash e : \sigma$. We proceed by induction on the final typing rule applied and show some representative cases below.

Case (Var$_\sigma$). Suppose the last rule applied was

$$\Phi \mid \Gamma, x :_r \sigma \vdash x : \sigma.$$

By algorithm step (Var), we have that

$$\Phi \mid \overline{\Gamma}, x : \sigma; x \Rightarrow \{x :_0 \sigma\}; \sigma$$

and $\{x :_0 \sigma\} \sqsubseteq (\Gamma, x :_r \sigma)$ as $0 \leq r$.

Case (⊗ I). Suppose the last rule applied was

$$\Phi \mid \Gamma, \Delta \vdash (e, f) : \sigma \otimes \tau.$$

From this, we deduce that $\mathrm{dom}\,\Gamma \cap \mathrm{dom}\,\Delta = \emptyset$. By induction, there exist $\Gamma_1 \sqsubseteq \Gamma$ and $\Delta_1 \sqsubseteq \Delta$ such that

$$\Phi \mid \overline{\Gamma}; e \Rightarrow \Gamma_1; \sigma \text{ and } \Phi \mid \overline{\Delta}; f \Rightarrow \Delta_1; \tau.$$

Moreover, $\mathrm{dom}\,\Gamma_1 \cap \mathrm{dom}\,\Delta_1 = \emptyset$ as well. By Lemma 27, we also have that

$$\Phi \mid \overline{\Gamma, \Delta}; e \Rightarrow \Gamma_1; \sigma \text{ and } \Phi \mid \overline{\Gamma, \Delta}; f \Rightarrow \Delta_1; \tau.$$

Thus, we can apply algorithm step (⊗ I) to conclude

$$\Phi \mid \overline{\Gamma, \Delta}; (e, f) \Rightarrow \Gamma_1, \Delta_1; \sigma \otimes \tau$$

where we know $(\Gamma_1, \Delta_1) \sqsubseteq (\Gamma, \Delta)$.

Case (⊗ $\mathbf{E}_\sigma$). Suppose the last rule applied was

$$\Phi \mid r + \Gamma, \Delta \vdash \text{let } (x, y) = e \text{ in } f : \sigma.$$

By induction, there exist $\Gamma_1 \sqsubseteq \Gamma$ and $\Delta_1 \sqsubseteq (\Delta, x :_r \tau_1, y :_r \tau_2)$ such that

$$\Phi \mid \overline{\Gamma}; e \Rightarrow \Gamma_1; \tau_1 \otimes \tau_2 \text{ and } \Phi \mid \overline{\Delta}, x : \tau_1, y : \tau_2; f \Rightarrow \Delta_1; \sigma.$$

If $x :_{r_1} \tau_1, y :_{r_2} \tau_2 \in \Delta_1$, let $r' = \max\{r_1, r_2\}$. As $\Delta_1 \sqsubseteq (\Delta, x :_r \tau_1, y :_r \tau_2)$, we know $r' \leq r$.

Using Lemma 27, we can apply algorithm step (⊗ E_σ) of

$$\Phi \mid \overline{\Gamma, \Delta}; \text{let } (x, y) = e \text{ in } f \Rightarrow (r' + \Gamma_1), \Delta_1 \setminus \{x, y\}; \sigma.$$

Moreover, $(r' + \Gamma_1) \sqsubseteq (r + \Gamma)$ and $(\Delta_1 \setminus \{x, y\}) \sqsubseteq \Delta$.

Case (+ E). Suppose the last rule applied was

$$\Phi \mid q + \Gamma, \Delta \vdash \textbf{case } e' \textbf{ of } (\textbf{inl } x.e \mid \textbf{inr } y.f) : \rho.$$

By induction, there exist $\Gamma_1 \sqsubseteq \Gamma$, $\Delta_1 \sqsubseteq (\Delta, x :_q \sigma)$, and $\Delta_2 \sqsubseteq (\Delta, y :_q \tau)$ such that

$$\Phi \mid \overline{\Gamma}; e' \Rightarrow \Gamma_1; \sigma + \tau \text{ and } \Phi \mid \overline{\Delta}, x : \sigma; e \Rightarrow \Delta_1; \rho \text{ and } \Phi \mid \overline{\Delta}, y : \tau; f \Rightarrow \Delta_2; \rho.$$

If $x :_{q_1} \sigma \in \Delta_1$ and $y :_{q_2} \tau \in \Delta_2$, let $q' = \max\{q_1, q_2\}$, and we know $q' \leq q$. Using Lemma 27, we can apply algorithm step (+ E) of

$$\Phi \mid \overline{\Gamma, \Delta}; \textbf{case } e' \textbf{ of } (\textbf{inl } x.e \mid \textbf{inr } y.f) \Rightarrow (q' + \Gamma_1), \max\{\Delta_1 \setminus \{x\}, \Delta_2 \setminus \{y\}\}; \rho.$$

Furthermore, we know $(q' + \Gamma_1) \sqsubseteq (q + \Gamma)$ and $\max\{\Delta_1 \setminus \{x\}, \Delta_2 \setminus \{y\}\} \sqsubseteq \Delta$.

Case (Add). Suppose the last rule applied was

$$\Phi \mid \Gamma, x :_{\varepsilon + r_1} \text{num}, y :_{\varepsilon + r_2} \text{num} \vdash \text{add } x\ y : \text{num}$$

where $r_1, r_2 \geq 0$. We can apply algorithm step (Add) of

$$\Phi \mid \overline{\Gamma}, x : \text{num}, y : \text{num}; \text{add } x\ y \Rightarrow \{x :_\varepsilon \text{num}, y :_\varepsilon \text{num}\}; \text{num}$$

and we have $\{x :_\varepsilon \text{num}, y :_\varepsilon \text{num}\} \sqsubseteq (\Gamma, x :_{\varepsilon + r_1} \text{num}, y :_{\varepsilon + r_2} \text{num})$.

□