STATIC ANALYSIS OF DIFFERENTIABLE PROGRAMS

BY

JACOB SCOTT LAUREL

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2024

Urbana, Illinois

Doctoral Committee:

Associate Professor Saša Misailović, Chair
Assistant Professor Gagandeep Singh,
Professor Darko Marinov,
Dr. Jan Hückelheim, Argonne National Laboratory

ABSTRACT

Differentiable Programming which includes Automatic Differentiation (AD), serves as the backbone for machine learning and simultaneously pervades many other domains including graphics and scientific computing. Despite AD's ubiquity, automated formal reasoning about the derivatives AD computes has lagged. The difficulty in developing automated analyses for differentiable programs stems from the fact that these programs use complicated semantics, contain highly nonlinear operations (e.g., the chain rule) and involve thousands of program variables. Thus, developing program analyses for AD that are simultaneously general, precise, and scalable remains a challenging task. To tackle these challenges, this dissertation develops a novel, unified framework for statically analyzing differentiable programs by coupling abstract interpretation with automatic differentiation.

To obtain program analyses with the desired generality, Chapter 2 of this dissertation proposes DeepJ, the first abstract interpretation framework built upon Clarke Generalized Jacobians. By leveraging a generalized notion of derivatives, DeepJ can soundly reason about gradient properties for both differentiable and non-differentiable (but Lipschitz continuous) functions that result from programs with control flow.

The need for generality also extends to higher derivatives and richer program abstractions. To enable these extensions, Chapter 3 proposes the first general construction for abstract interpretation of higher-order AD. This construction allows one to automatically build sound abstract interpreters for arbitrary orders of derivatives with general numerical abstract domains for precise analysis of program properties defined over higher derivatives.

To obtain the desired precision, Chapter 4 proposes Pasado which is the first automated technique to synthesize precise static analyzers tailored to the structure of AD. By formulating abstract interpretation as a tractable optimization problem, Pasado optimally solves for precise abstractions of groups of multiple nonlinear operations which correspond to the chain rule, product rule, and quotient rule. Since these rules underlie forward and reverse mode AD, Pasado's generality enables abstract interpretation of both modes. Compared to the prior state of the art, this precision allows Pasado to compute derivative bounds and local Lipschitz constant bounds that are orders of magnitude more precise.

In addition, this dissertation shows extensive experimental results in Chapters 2, 3, and 4 highlighting how these static analyses obtain high scalability. These experiments demonstrate how the proposed analyses successfully reason about derivatives of large convolutional neural architectures involving hundreds of thousands of nonlinear operations.

To my Mom and Dad for their unwavering support and unconditional love.

ACKNOWLEDGMENTS

While on a symbolic submarine ride through graduate school, I was privileged to have the support of so many people throughout my journey. My deepest thanks goes to my advisor Sasa Misailovic for taking a chance on a first year PhD student who had just switched areas, had never taken a formal methods or compilers class, and was on academic probation. Undeterred by all these factors, Sasa gave me both the encouragement and freedom to carve out my own research path. Sasa devotes an incredible amount of time to each of his students and always believes in them even when they doubt themselves. Sasa is an incredible researcher, mentor, and person and it has been an immense privilege to learn from him. Going forward, I will always strive to "be brilliant!".

I owe an almost equally big thank you to Gagandeep Singh for being an outstanding collaborator and incredible mentor, and also for his constant encouragement. Gagandeep has proven exceptional at helping me take half-baked ideas and turn them into crisp and impactful research problems. Gagandeep devotes a tremendous level of support to everyone he collaborates with, an example that I will always strive to follow.

One day in Fall 2017, Darko Marinov left a note on my desk saying "If you ever want to talk, please don't hesitate to stop by my office" when he noticed my struggles with adjusting to the new expectations of graduate school. I am forever grateful for this encounter and for Darko's incredible care for every student he meets. I likely would never have ventured into the PL/FM/SE area had it not been for that note. In addition, Darko has been a delightful source of wisdom and witty humor.

I would also like to thank Jan Hückelheim for both his service on my thesis committee, and for the wonderful internship experience at Argonne in Fall 2024.

Keyur, Saikat, Vimuth, Zixin, Yifan, and Shubham are the best labmates a PhD student could ever ask for. Whether it was traveling to conferences with Saikat, taking over the night shift with Yifan, watching a game with Shubham, walking through the arboretum with Keyur, having whiteboard discussions with Zixin, or dinner with Vimuth, building friendships with each of you has been an incredible privilege. I am also thankful for the excellent collaborations I had with other PhD students such as when Ashitabh and I brought many of the ideas found in this thesis full circle with our DATE paper.

I am deeply grateful to the talented undergraduates I had the privilege of working with: Rem, Atharva, Robert and Brant (Siyuan). From the time I first met Rem at the research fair to the final trip to Portugal with Brant, the deadlines (and subsequent Golden Harbor

visits) we experienced together were the most enriching and fulfilling parts of my research. You guys may not realize it, but I learned more from you all than you did from me!

Outside of research, I owe a big thanks to my UIUC friends who (in no particular order) made my 7+ years in Champaign so much fun: Adithya, Angello, Liia, Wing, Ian, Federico, Adam, James, Omri, Marc and all the others. I will especially miss our Friday extravaganzas.

The administrative staff who keep the UIUC Computer Science ship sailing also deserve a tremendous thank you, in particular Kalen McGowan, Kara MacGregor, Dana Garard, Jennifer Comstock and Viveka Kudaligama. Their assistance has proven invaluable.

I am also grateful to all the faculty who helped me prepare for my academic job search, especially Sayan Mitra, Tianyin Xu, Robin Kravets, Elsa Gunter and Tandy Warnow. In addition, I received very helpful feedback on my statements (and countless proofreads) from the members of the job market seminar (especially Darko), the UIUC writing center, and Anna, who also helped with my countless practice talks. A big thank you to all of them.

I also thank the anonymous reviewers of my submissions who provided valuable feedback and the institutions that helped fund my research. Specifically, this work has been funded in part by NSF Grants No. CCF-1846354, CCF-1956374, CCF-2008883, CNS-2148583, USDA NIFA Grant No. AG NIFA 2021-67021-33449, and a Sloan UCEM Graduate Scholarship.

The strong foundations my UIUC successes were built upon, stem from many outstanding undergraduate professors and high school teachers. Specifically, I thank Arie Nakhmani, Niels Lobo, and Boqing Gong for my first research experiences. Going back before college, I thank Mr. Aquila, Mr. Thorne, and Mrs. Burkavich for pushing me to be my best. I also thank Justin, Nitish, Harish, Aman, Matt, and all my friends from back home.

I owe a tremendous thanks to my Aunt Liz and my 4 wonderful cousins who comprise the Zirgaitis clan for being my home away from home in Illinois. Liz and the girls welcomed me with arms wide open whenever I visited Chicago and these visits gave me much needed, restorative breaks from the PhD grind. Liz and Badri also showed me incredible hospitality while I stayed with them during my Argonne internship. Though Cincinnati is slightly further away, the times I visited Aunt Lisa, Uncle Larry, Mamaw and my equally wonderful cousins from the Graham clan (plus the times Matthew visited me) were just as enjoyable. I am also deeply grateful for all the love and support my grandparents Roy, Carol, Sue, and Servando provided to me over the years as well as for the support of all my extended family.

My sisters Marissa, Nicole, and Cameryn have always been there for me no matter what. Seeing them every break kept me sane and they are truly the best sisters I could ever ask for. Lastly, and most importantly, I am forever thankful for my mother, Camilla, and my father, Hector and this dissertation is dedicated to them. The unconditional love and support they have blessed me with truly has no limits. Without them, none of this would be possible.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1
1.1 Preliminaries 1
1.2 Challenges 9
1.3 Thesis Statement 11
1.4 Dissertation Organization 11

CHAPTER 2 DEEPJ - A DUAL NUMBER ABSTRACTION FOR STATIC ANALYSIS OF CLARKE JACOBIANS 13
2.1 Introduction 13
2.2 Example 17
2.3 Preliminaries 22
2.4 Language Syntax and Semantics 26
2.5 Interval Clarke Jacobian 34
2.6 Dual Interval Domain 48
2.7 Methodology 61
2.8 Experimental Evaluation 63
2.9 Related Work 72
2.10 Summary 73

CHAPTER 3 A GENERAL CONSTRUCTION FOR ABSTRACT INTERPRETATION OF HIGHER-ORDER AUTOMATIC DIFFERENTIATION 75
3.1 Introduction 75
3.2 Example 77
3.3 Preliminaries 84
3.4 Language Syntax and Semantics 87
3.5 Abstract Semantics of Higher-Order AD 97
3.6 Instantiations 105
3.7 Case Studies 108
3.8 Related Work 113
3.9 Summary 114

CHAPTER 4 SYNTHESIZING PRECISE STATIC ANALYZERS WITH PASADO 115
4.1 Introduction 115
4.2 Example 118
4.3 Preliminaries 126
4.4 Synthesizing Precise AD Static Analyzers 129
4.5 Case Studies 148
4.6 Related Work 158

4.7 Summary 159

CHAPTER 5 CONCLUSIONS AND FUTURE WORK 160
5.1 Conclusions 160
5.2 Future Work 160

REFERENCES 163

CHAPTER 1: INTRODUCTION

Differentiable Programming which includes Automatic Differentiation (AD), serves as the backbone for machine learning and simultaneously pervades many other domains including graphics and scientific computing. Despite AD's ubiquity, automated formal reasoning about the derivatives AD computes has lagged. This absence of formal reasoning about AD code poses problems since many important program properties are defined over derivatives. For instance, in TinyML, when computing gradients in low precision datatypes, one must ensure the gradient's range does not overflow past the datatype's supported range, since overflows would ruin a computation's result. As another instance, in high-stakes social settings, ML systems may automate hiring decisions. Hence one may want to ensure that if two people are comparably qualified (e.g., they have the same education level), but one has more experience, the more experienced person should be hired, otherwise that hiring system could be unfair. Fairness in this example can be formalized as a monotonicity property on the experience level, which equivalently means all derivatives are strictly positive.

Thus practitioners across multiple domains (like TinyML and trustworthy ML) need answers to questions like "*is the largest possible gradient always less than some threshold?*" or "*are all the computed derivatives strictly positive?*". Despite the need to analyze and verify properties such as these, such questions remain severely understudied.

This dissertation answers these questions by combining abstract interpretation with AD. Abstract interpretation [1] is a scalable and general program analysis framework that allows programmers to automatically reason about *sets* of program executions instead of a single execution. By merging abstract interpretation and AD, my dissertation research helps programmers automate the verification of formal properties like monotonicity that are defined over *sets* of gradients. Hence, my dissertation builds a unified and automated framework that gives programmers these formal guarantees and removes the burden of needing to know all the intricacies of calculus. Further, as in the case of monotonicity, by analyzing and verifying properties over a program's set of gradients, the techniques proposed in this dissertation can verify properties about the original program itself.

1.1 PRELIMINARIES

Before proceeding further, we now provide the preliminaries necessary for formulating the contribution of this dissertation. Additionally, while this dissertation strives to use consistent notation wherever possible, each chapter's notation can differ slightly. For instance the

$$
\begin{array}{rcl}
P & ::= & P_1; P_2 \mid x_i = Expr \\
Expr & ::= & x_j + x_k \mid x_j - x_k \mid x_j * x_k \\
 & \mid & 1/x_j \mid \log(x_j) \mid \exp(x_j) \\
 & \mid & \cos(x) \mid \sin(x) \mid \sigma(x_j) \mid c \in \mathbb{R}
\end{array}
$$

Figure 1.1: Differentiable Function Syntax

symbols used to denote ideas in one chapter may overlap with symbols used to present different ideas in other chapters. Hence to avoid confusion, the notations found in each chapter should be treated as separate notations valid only for their respective chapters.

1.1.1 Differentiable Programming and Automatic Differentiation

Given that computer programs often define mathematical functions, one may ask: can these mathematical functions be differentiable? This question motivates the idea of Differentiable Programming, also called Automatic Differentiation (AD) which is a way to automatically construct programs that compute the derivatives of other programs. Automatic Differentiation has a long history in Computer Science, going back to at least the 1960s [2], and these ideas have been independently redeveloped over subsequent decades [3, 4]. Indeed, the rediscovery of these techniques across disjoint communities (programming languages, machine learning and scientific computing) has led to slightly different and overlapping terminology for describing the same ideas. For instance, Differentiable Programming is the term commonly used by the programming languages (PL) community, and it also carries a connotation of a PL-centric perspective. This perspective focuses on program semantics and correctness with respect to the underlying mathematics. In contrast, Automatic Differentiation (AD) is the term used more commonly in machine learning and scientific computing, which simultaneously carries a more practical and implementation-focused connotation. For simplicity, this dissertation will use these two terms interchangeably as such fine-grained distinctions are unnecessary for describing the techniques proposed by this dissertation.

To describe Automatic Differentiation, we first introduce in Fig. 1.1, a core language used to construct programs that represent differentiable functions. We will later develop different variations of this language in Chapters 2, 3, and 4. For a given language, AD frameworks are typically built upon techniques like operator overloading or source code transformations, and these frameworks enable automatically applying the rules of calculus (e.g., chain rule) at the level of a program's source code in order to obtain a new program that is a valid mathematical derivative of the original program (called the primal) [5]. For instance, given

```
1  x = Input();
2  y = sin(x)+x;
3  return y;
```

(a)

```
1  x = Input();
2  x_d = 1;
3  y = sin(x)+x;
4  y_d = (cos(x)+1)*x_d;
5  return y,y_d;
```

(b)

Figure 1.2: (a) Primal Program and (b) Forward-Mode Differentiated Program

the simple program `x = input(); y = sin(x)+x; return y;` (shown in Fig. 1.2a), AD can produce a new program (shown in Fig. 1.2b) that computes $cos(x) + 1$ so that we can obtain $\frac{\partial \mathtt{y}}{\partial \mathtt{x}}$. The original program is often referred to as the *primal*. In this example, forward mode AD implemented by source code transformation is used. The variables corresponding to derivatives of intermediate program variables are inserted by the compiler and are denoted with a **_d** suffix. In particular, `y_d` stores the value of $\frac{\partial \mathtt{y}}{\partial \mathtt{x}}$.

Beyond computing first derivatives as shown in the example above, AD can be iterated to compute higher-derivatives (e.g., Hessians), an idea we formally expand upon in Chapter 3.

AD has two main variants: forward-mode and reverse-mode. In the forward mode (which is used in Fig. 1.2), the derivatives are computed side-by-side with the original program variables, whereas in reverse mode AD, the entire original program is first computed before any derivatives are computed [5]. Forward-mode AD can be thought of as a forward propagation (through the computational graph) of derivatives of intermediate program variables with respect to fixed input variables. Reverse-mode AD can be thought of as a backward propagation (through the computational graph) of the derivative of a fixed output variable with respect to intermediate program variables.

To compute the entire Jacobian of a function $f : \mathbb{R}^m \to \mathbb{R}^n$ (expressed as a differentiable program), the time complexity of forward-mode AD is proportional to the number of *input* variables: $\mathcal{O}(m)$. In contrast, the time complexity of reverse-mode AD is proportional to the number of *output* variables: $\mathcal{O}(n)$. We will subsequently show how the analyses proposed in this dissertation support both forward mode AD in Chapters 2, 3 and reverse mode AD in Chapter 4.

Differentiable Programming Language Expressivity

Since the language supports function compositions (e.g., exp), multiplication, and division, the computational graph described by a program is naturally differentiable. Indeed, the chain rule, product rule and quotient rule will respectively be applied by the compiler

for each of those operations in the original program. While this language may appear restricted, it remains expressive enough to encode important functions and programs from a variety of applications across Machine Learning, Optimization, and Scientific Computing. For instance, our language can easily express deep neural networks (DNNs) which we will show in Chapter 2, and our language can also encode numerical ODE solvers like Euler and Runge-Kutta solvers (shown in Chapter 4). We will also show in Chapter 2 how to add expressivity to the language by enabling limited branching support.

1.1.2 Formal Properties Defined Over Derivatives

The first derivatives specified by the Jacobian matrix form the foundation of many prominent learning paradigms and are used in all facets of the machine learning pipeline, from training to testing. Beyond ML, derivatives (including higher-derivatives) are used extensively in scientific computing for tasks as diverse as climate modeling [6], analyzing differential equations [7, 8] and sensitivity analysis [9].

Since AD enables one to differentiate through complex programs written for these aforementioned tasks, one can now define and analyze formal properties over those derivatives. Hence, AD allows one to go beyond analyzing properties over the program's original variables and instead analyze properties over those variables' *derivatives*. For this reason, many key formal properties are defined over the derivatives and gradients that AD computes. Hence differentiable programming opens up a new class of formal methods problems. We now describe key formal properties that are defined over derivatives computed by AD.

Lipschitz Robustness. Lipschitz constants offer a natural way to reason about a function's behavior. Lipschitz constants can be used as a metric to compare the stability and smoothness of the output of neural networks prior to deployment, as a network with a smaller constant is often preferable [10]. Formal bounds on the Lipschitz constant can also be used during training to learn classifiers that are certifiably robust to adversarial perturbations [11], robust to quantizations [10], or to improve interpretability by making network explanations themselves more robust [12]. Further, analyzing the Lipschitz constant has direct applications in algorithmic fairness [13] and differential privacy [14], where fairness and privacy are established by certifying a small Lipschitz constant. This formal property can now be stated in Eg. 1.1. For a differentiable function $f(x) : \mathbb{R}^m \to \mathbb{R}$ and local region $\mathcal{X} \subset \mathbb{R}^m$

$$\max_{x \in \mathcal{X}} \left\| \frac{\partial f(x)}{\partial x} \right\| \leq K \implies \forall x_1, x_2 \in \mathcal{X},\ \|f(x_1) - f(x_2)\| \leq K\|x_1 - x_2\| \tag{1.1}$$

In this setting $K \in \mathbb{R}_{>0}$ is the local Lipschitz constant. Intuitively, the local Lipschitz

constant means that the function f can be bounded by a line with slope K. This dissertations provides experiments analyzing this Lipschitz robustness property can be found in Chapters 2, 3, and 4.

Optimization Analysis. Beyond using the Jacobian for formally bounding (local) Lipschitz constants, a Jacobian analysis can also be used to formally reason about the local optimization geometry of ML models [15]. As an example, one may wish to certify that for some local region, $\mathcal{X} \subset \mathbb{R}^m$, a function of interest $f(x) : \mathbb{R}^m \to \mathbb{R}$ never attains its extrema. This property can now be stated in Eq. 1.2 as:

$$\forall x' \in \mathcal{X}, \quad \frac{\partial f(x)}{\partial x}\bigg|_{x=x'} \neq 0 \tag{1.2}$$

Existing work [16] discusses how derivative bounds which provably exclude zero can then be incorporated into branch-and-bound optimization solvers to provably rule out entire (local) regions of the input space. Hence by knowing that a region excludes any optimal values, an optimization solver can avoid paying the computational cost to explore that region. Chapter 2 of this dissertation provides experimental results that analyze this property. Indeed Chapter 2 will even show how this property can still be formalized and generalized for functions which contain points of non-differentiability.

Convexity and Concavity. Similar to the first-derivative properties defined above, another formal property that can be defined over derivatives is convexity. Unlike the above properties, the convexity property of a function is defined over *second* derivatives. For a function $f : \mathbb{R}^m \to \mathbb{R}$ and a local region $\mathcal{X} \subset \mathbb{R}^m$, f is convex over $\mathcal{X}$ is the following holds:

$$\min_{x \in \mathcal{X}} \quad x^T \cdot H(f, x) \cdot x \geq 0 \tag{1.3}$$

In this setting $H(f, x)$ is the Hessian, or matrix of all partial second derivatives. Similarly, f is concave over $\mathcal{X}$ if Eq. 1.4 holds.

$$\max_{x \in \mathcal{X}} \quad x^T \cdot H(f, x) \cdot x \leq 0 \tag{1.4}$$

We note that for functions of a single variable, these properties expressed over second derivatives reduce to:

$$\min_{x \in \mathcal{X}} \ \frac{\partial^2 f(x)}{\partial^2 x} \geq 0 \implies f \text{ is convex} \tag{1.5}$$

$$\max_{x \in \mathcal{X}} \ \frac{\partial^2 f(x)}{\partial^2 x} \leq 0 \implies f \text{ is concave} \tag{1.6}$$

While this dissertation does not study the convexity/concavity property, it has been briefly explored in the literature [16].

Sensitivity Analysis. Sensitivities are often expressed with derivatives. For instance to understand how sensitive a function f is to some input x_i, one often computes $\frac{\partial f}{\partial x_i}$. These derivatives are commonly computed by AD [9], hence allowing one to perform sensitivity analysis of entire programs. One may aim to prove that in some region $\mathcal{X} \subset \mathbb{R}^m$ that the sensitivity is bounded by some amount. This property is stated in Eq. 1.7 and it corresponds to a *robust* sensitivity analysis.

$$\forall x \in \mathcal{X}, \quad \frac{\partial f(x)}{\partial x_i} \leq K \tag{1.7}$$

Chapter 4 of this dissertation provides an instance of this property by performing robust sensitivity analysis of a climate model.

Explainable ML: Feature Attributions and Interactions Very similar in spirit to sensitivity analysis is Explainable Machine Learning (ML) which aims to interpret how ML models like neural networks make decisions. In Explainable ML, one may want to know which features are more important or salient than other features. The reason is that one may wish to *attribute* an output to a particular feature, hence why they are often referred to as feature attributions. Hence formal specifications over derivatives can be used to define the ranking of which features are more salient than others [17, 18]. This formal specification can be stated as follows: for a function (e.g., a DNN) $f : \mathbb{R}^m \to \mathbb{R}$, a local region $\mathcal{X} \subset \mathbb{R}^m$ and features x_i and x_j where $i \neq j$, we want to verify the property specified in Eq. 1.8 holds.

$$\forall x \in \mathcal{X}, \quad \frac{\partial f(x)}{\partial x_j} \leq \frac{\partial f(x)}{\partial x_i} \tag{1.8}$$

This specification establishes that feature x_i is more salient than x_j. In practice we can verify this specification by proving the inequality shown in Eq. 1.9:

$$\max_{x \in \mathcal{X}} \frac{\partial f(x)}{\partial x_j} \leq \min_{x \in \mathcal{X}} \frac{\partial f(x)}{\partial x_i} \tag{1.9}$$

Hence as long as the upper bound of one feature's attribution is less than the lower bound of another feature's attribution, one can provably rank the importance of the features. An example of this specification can be seen in Chapter 3.

In explainable ML, higher-order derivatives can be used to express formal properties corresponding to the interaction of *multiple* input features [19, 20]. Thus for a function of interest $f(x_1, ..., x_m) : \mathbb{R}^m \to \mathbb{R}$, a local region $\mathcal{X}$, and n (not necessarily distinct) input features of interest $x_i, ..., x_k$ one may wish to find the tightest range $a, b \in \mathbb{R}$ such that:

$$\forall x \in \mathcal{X},\ a \leq \frac{\partial^n f(x)}{\partial x_i, ..., \partial x_k} \leq b \tag{1.10}$$

Chapter 3 of this dissertation provides experimental results analyzing the higher-derivative interaction property for neural networks.

Monotonicity. In many applications, the formal property one needs to certify is that a function, such as DNN behaves monotonically. The monotonicity property can be useful in high-stakes social settings such as for ML systems that hire candidates or decide to offer applicants loans [21]. Indeed many algorithmic fairness properties can be formalized as a monotonicity condition [22, 23]. For example, one may try to ensure that for two otherwise equally qualified job candidates, the candidate with more work experience is preferred.

The monotonicity property for a (differentiable) function $f : \mathbb{R}^m \to \mathbb{R}$ (e.g., a DNN) with respect to a feature x_i over a range $\mathcal{X} \subset \mathbb{R}^m$ can be stated as follows: f is monotonically increasing with respect to feature x_i if the implication shown in Eq. 1.11 holds.

$$\forall x, y \in \mathcal{X}, x_i \leq y_i, \forall j \neq i, x_j = y_j \implies f(x) \leq f(y) \tag{1.11}$$

However the above definition is relational since it is defined over separate inputs x and y where $x = (x_1, ..., x_i, ..., x_m)$ and $y = (y_1, ..., y_i, ..., y_m)$. Using *derivatives*, the monotonically increasing specification can be recast as:

$$\forall x \in \mathcal{X},\quad \frac{\partial f(x)}{\partial x_i} \geq 0 \tag{1.12}$$

and f is monotonically decreasing (with respect to x_i) if

$$\forall x \in \mathcal{X},\quad \frac{\partial f(x)}{\partial x_i} \leq 0 \tag{1.13}$$

Chapter 4 provides experiments that analyze the monotonicity property.

Range Analysis. One may also need to perform a range analysis on the gradients. For instance in [24] the authors needed to reason about how small and how large gradients obtained during gradient descent can be so that their compiler can select an appropriate fixed point arithmetic data type that has sufficient number of integer bits to avoid gradient overflows and underflows. The range specification can be stated as follows: for a differentiable function f, an input region $\mathcal{X} \subset \mathbb{R}$ one needs to find the tightest $a, b \in \mathbb{R}$ such that

$$\forall x' \in \mathcal{X},\quad a \leq \left.\frac{\partial f}{\partial x}\right|_{x=x'} \leq b \tag{1.14}$$

Domain Specific Properties. In specific domains, researchers have formalized proper-

ties over derivatives for highly specific classes of models such as those formalized to reason about physics-inspired DNNs in scientific computing [25]. As another instance, [16] certify bounds over third-order derivatives to compute sensitivities needed to select where to apply approximations to a program. In addition, other work has certified bounds on derivatives in connection with barrier functions for verifying self-driving control systems [26].

1.1.3 Abstract Interpretation

While we formally defined the properties of interest above, we still need an automated method to analyze and verify those properties. As a solution, this dissertation will use abstract interpretation as the foundational framework to analyze these derivative properties. First developed by the Cousots [1], abstract interpretation is a formal framework for analyzing the behavior of a program on sets of inputs. This task is performed statically (e.g., at compile time), meaning the analysis occurs before the program is ever executed on concrete inputs. One first defines the standard concrete semantics of the programs and then defines an abstraction of those semantics which is called the abstract semantics. The abstract semantics are then designed to *soundly over-approximate* the true concrete semantics. The notion of a sound over-approximation means that any program execution that could occur on a concrete input (when executed under the concrete semantics) is always accounted for in the set of possible executions obtained by interpreting the program under the abstract semantics. While abstract interpretation is sound, it is not complete, meaning that even if the abstract interpreter says a possible execution could happen, it may never actually happen for any concrete input, in which case the abstract interpreter produced a false alarm. The reason we settle for sound, but incomplete program analyses is because answering program analyses questions *exactly* is generally an undecideable task due to Rice's Theorem [27].

One can define abstract semantics and abstract interpreters for various tasks such as determining the sign (e.g., + or −) of numerical program variables [1], or determining the interval range of values of numerical program variables [1, 28]. In particular, analyses of the last case can be useful for checking if the range of numbers a program may divide by includes 0, since such an analysis could check for, or prove the absence of divide-by-zero errors. The program analyses proposed in this dissertation focus on this last case of computing (interval) numeric ranges for program variables corresponding to derivatives. For this reason we will use numerical abstract domains [28] to describe the local regions $\mathcal{X} \subset \mathbb{R}^m$ for which we want to prove a formal property defined over derivatives described in Section 1.1.2.

Due to the over-approximation inherent to abstract interpretation, a core challenge in building program analyses is to ensure that they are as precise as possible. Precision of

a static program analyses means that the over-approximation (which intuitively can be thought of as the amount of false alarms), is kept to a minimum. In Chapters 3 and 4 we will highlight specific strategies that can successfully tame the over-approximation inherent to abstract interpretation.

1.2 CHALLENGES

While Automatic Differentiation and Abstract Interpretation offer ample opportunities for synergy, combining these two distinct techniques encounters several challenges. In particular, one must ensure that static analysis of differentiable programs by abstract interpretation attains *generality, precision and scalability*. While these concerns are also faced by other types of program analyses, the setting of differentiable programming provides unique challenges and opportunities. In this context, *generality* means the ability of an analysis to support multiple different features of AD such as higher-order derivatives and non-differentiable functions. *Precision* means the analysis should compute as tight of a bound as possible on the derivative expressions which is a challenging task since most derivative expressions are highly nonlinear. Lastly, *scalability* means the analysis should compute derivative bounds for programs with as many variables as possible - a core necessity for large programs like Deep Neural Networks. We now describe these challenges in more detail.

1.2.1 Precision

Obtaining precision for AD static analyses remains difficult. This difficulty stems from the large amount of nonlinear operations inherent to AD. Nonlinear operations pose challenges because most abstract interpretations were designed for linear operations [29, 30]. In the context of AD, compared to just the original program (called the *primal*), the derivative program AD computes (called the *adjoint*) can have 2-5× more non-linear operations [5], e.g., for the most common operations:

- Every composition with a *non-linear function* in the primal requires a separate composition with that function's derivative in the adjoint and an additional multiplication, due to the chain rule.
- A single multiplication in the primal leads to 2 separate multiplications in the adjoint due to the product rule.
- A single division in the primal leads to 4 nonlinear operations in the adjoint due to quotient rule.

As a strategy to tame the imprecision resulting from this increased amount of nonlinearity (compared to other kinds of programs) one could try to design optimal abstractions for groups of operations. However the challenge then becomes how to choose the right level of granularity for a more precise abstraction, a question which lacks an easy answer.

1.2.2 Generality

Formal, compositional reasoning about the semantics of differentiable programs presents challenges because computer programs are often *non-differentiable*. These points of non-differentiability stem from branch statements in the program [31, 32]. These mathematical pathologies in the program mean one thing: to prove formal guarantees about AD code, more generalized types of derivatives are needed.

The need for generality also extends to *higher* derivatives and *richer* abstract domains. Formal properties are often defined over higher derivatives. Previously, a programmer would have to define an AD semantics for the desired order of derivative and then prove the corresponding abstract semantics sound for a chosen abstract domain. To compute a different order of derivative or use a different abstract domain, all proofs would need to redone which puts a heavy burden on the programmer.

1.2.3 Scalability

Since derivative computations in AD typically have 2×-5× more operations than the original function that was differentiated [5], scalability becomes a primary concern. Furthermore, tackling the precision challenge also affects the scalability, as more precise analyses tend to be more expensive and thus less scalable.

The challenges of *precision*, *generality*, and *scalability* do not exist in isolation. In fact these three dimensions mutually influence each other [33]. For instance generality and precision are often competing goals and similarly obtaining more precision often comes at the cost of scalability. In addition, many concerns exist at the intersection of these dimensions. For example supporting the analysis of higher-order AD creates challenges for precision since higher-derivatives contain more nonlinearity than first derivatives and creates challenges for generality since a new semantics are needed. Hence finding the appropriate "sweet spot" between these three dimensions is no easy feat. A visual representation of these dimensions and the AD-specific concerns that cut across these three dimensions is shown in Fig. 1.3.

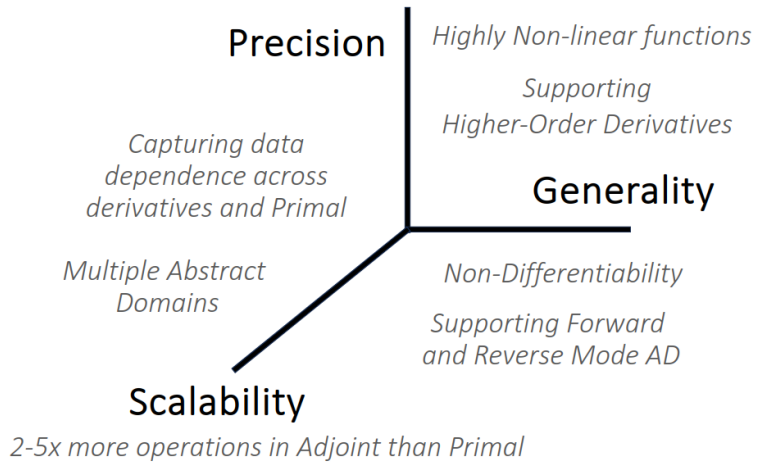


Figure 1.3: Precision, Generality and Scalability dimensions along with their associated analysis concerns which are shared across multiple dimensions.

1.3 THESIS STATEMENT

Having introduced the requisite program analysis and differentiable programming preliminaries, we can now state the core idea of this dissertation. This dissertation's thesis statement can be stated as follows:

By combining abstract interpretation with differentiable programming, we can achieve precise, scalable and general static analyses for analyzing and certifying formal properties defined over a program's derivatives.

1.4 DISSERTATION ORGANIZATION

The remainder of this dissertation is organized as follows:

Chapter 2: DeepJ - A Dual Number Abstraction for Static Analysis of Clarke Jacobians. This chapter presents DeepJ [34] which is the first abstract interpretation of Clarke Generalized Jacobians. By defining an abstract semantics based upon Clarke Jacobians, DeepJ is general enough to compositionally reason about gradient properties for both Differentiable and Non-Differentiable (but Lipschitz continuous) functions. Due to DeepJ's generality, it is the first work to certify Lispchitz robustness guarantees on deep neural networks when they are adversarially perturbed by non-differentiable perturbations like image rotations, a threat model no prior work handled.

Chapter 3: A Generalized Construction for Abstract Interpretation of Higher-Order AD. This chapter presents and proves sound the first general framework for building

abstract interpreters for arbitrary orders of derivatives with general classes of abstract domains [35]. Thus this technique frees the programmer from the burden of repeatedly having to reprove their semantics and abstraction sound each time they want to use a different abstract domain or compute a different order of derivative.

Chapter 4: Synthesizing Precise Static Analyzers with Pasado. This chapter presents Pasado [36], the first automated technique for synthesizing precise static analyzers tailored specifically to the structure of AD. By formulating abstract interpretation as a tractable optimization problem, Pasado is able to optimally solve for precise abstractions of not just individual operations, but rather groups of multiple non-linear operations corresponding to the chain rule, product rule and quotient rule. Since these rules are intrinsic to both forward and reverse mode AD, Pasado is general enough to support both modes. By targeting the expressions of these 3 calculus rules directly, Pasado's continuous optimization-based abstraction offers significant improvements in precision while maintaining scalability. Furthermore, Pasado's optimization-based AD abstraction can scale to large CNNs which have tens of thousands of neurons in a given intermediate layer and also compute local Lipschitz constant bounds that are >2000x more precise than the prior state of the art.

Chapter 5: Conclusions and Future Directions This chapter summarizes the contributions of this dissertation and concludes by introducing new directions that build upon the ideas presented in this dissertation.

CHAPTER 2: DEEPJ - A DUAL NUMBER ABSTRACTION FOR STATIC ANALYSIS OF CLARKE JACOBIANS

This chapter presents a novel abstraction for bounding the Clarke Jacobian of a Lipschitz continuous, but not necessarily differentiable function over a local input region. To do so, we leverage a novel abstract domain built upon dual numbers, adapted to soundly over-approximate all first derivatives needed to compute the Clarke Jacobian. We formally prove that our novel forward-mode dual interval evaluation produces a sound, interval domain-based over-approximation of the true Clarke Jacobian for a given input region.

Due to the generality of this formalism, we can compute and analyze interval Clarke Jacobians for a broader class of functions than previous works supported – specifically, arbitrary compositions of neural networks with Lipschitz, but non-differentiable perturbations. We implement our technique in a tool called DeepJ and evaluate it on multiple deep neural networks and non-differentiable input perturbations to showcase both the generality and scalability of our analysis. Concretely, we can obtain interval Clarke Jacobians to analyze Lipschitz robustness and local optimization landscapes of both fully-connected and convolutional neural networks for rotational, contrast variation, and haze perturbations, as well as their compositions.

2.1 INTRODUCTION

The needs of ML researchers have rapidly outpaced formal development on the programming languages side. For instance, ML techniques regularly must differentiate through functions that may have points of non-differentiability, such as the commonly used ReLU activation in neural networks, which is not differentiable at the origin. In practice, for such a situation, existing AD frameworks (e.g., [37]) may return an arbitrary number in the interval $[0, 1]$. While such an ad-hoc approach may suffice in many practical scenarios, for critical domains requiring formal certification of properties defined over the first derivatives, this lack of rigor is troubling. To resolve this limitation, programming language researchers have begun using generalizations of the Jacobian for describing AD semantics [32, 38], such as the *Clarke Generalized Jacobian* [39] for non-smooth, but Lipschitz continuous functions.

However, these developments still cannot provide the desired formal guarantees for practical verification tasks, such as obtaining formal bounds on the Lipschitz constant (which measures how rapidly a function's output changes) of the composition of a non-differentiable perturbation and a differentiable network within an input region.

Challenges

We focus on designing a static analysis that can formally reason about not only Jacobians of functions that are differentiable (e.g., tanh), but can also handle non-differentiable behavior due to functions like ReLU and the branching that can arise in differentiable programs. Furthermore, we must also handle high-dimensional computational graphs with arbitrary arithmetic operations (instead of, e.g., solely neural networks that avoid non-scalar multiplication and division). Simultaneously handling *all* of these requires a theoretical formalism that is beyond the scope of the existing works [15, 40, 41, 42]. One of the main difficulties arises from the fact that generalized notions of Jacobians do not always obey the same rules as classical Jacobians. For example, the Clarke Jacobian cannot simply be computed by concatenating partial Clarke derivatives into a matrix [39, 43]. In this same light, we want our desired generality to also come with compositionality: we want to be able to combine different functions together instead of restricting ourselves to a non-compositional analysis limited to a specific, fixed type of function (e.g., a DNN) or input perturbation. More practically, we want this analysis to be scalable and fully end-to-end, specifically for real-world problems such as analyzing Lipschitz robustness with respect to multiple input perturbations. This task has proven challenging, as most formalisms that aim for broad theoretical generality cannot scale beyond toy examples [44], do not have implementations [40, 45, 46, 47], or lack end-to-end integration with specific analyses for practical problems. Moreover, the Lipschitz robustness analyses that go beyond toy examples, such as RecurJac [15] and ProLip [41], are either tailored for handling fully-connected architectures (like RecurJac) and therefore cannot immediately analyze the state-of-the-art convolutional architectures, or are heavily restricted in the activation functions supported (like ProLip). These issues substantially limit the practical applicability of these tools.

This Work

To address these challenges, we propose DeepJ, a forward-mode interval abstraction built atop dual numbers (the canonical number system used for implementing forward-mode AD). A dual number $a + b\epsilon$ has two components: the *real part* a and the *dual part* b, which in applications will correspond to a function's derivative at a.

The analysis is adapted to compute an interval over-approximation of the Clarke Jacobian. Our key insight is that formalizing the static analysis on top of dual numbers as a forward-mode analysis represents a general solution suitable for reasoning about Clarke derivatives, which can simultaneously offer both an intuitive and scalable implementation.

Hence, we reduce the problem of bounding a Clarke Jacobian in a local region to the problem of bounding results of dual number arithmetic and functions. Forward-mode analysis can be particularly useful for multiple practical problems, such as analyzing Lipschitz robustness with respect to individual input perturbations or their compositions. These problems have small input dimension, for which a forward-mode analysis requires fewer passes than a reverse-mode analysis.

DeepJ analyzes a first-order core (without unbounded loops or recursion) of the language proposed by [38], which we extend with conditional branch expressions and also show necessary conditions for the well-definedness of the Clarke Jacobian of these branches (Section 2.4). We then recursively define an interval-domain abstraction of the Clarke Jacobian for sets of points and prove this over-approximation sound (Section 2.5). Next, we show how to equivalently compute this interval Clarke Jacobian in a forward pass using a novel, interval-domain abstraction of dual numbers (Section 2.6). Finally, we demonstrate how DeepJ leverages the interval over-approximation of the Clarke Jacobian for multiple practical uses in a fully end-to-end and scalable manner, namely analyzing Lipschitz robustness and local optimization geometry of large neural networks in the face of non-smooth input perturbations (Section 2.8). DeepJ's implementation also optionally offers floating-point soundness [48], i.e., its result can capture all possible outputs under different rounding modes and under different orders of computations of floating-point operations. This guarantee is not possible with any other existing method.

The novelty in our work lies in the fact that we are the first to formalize a static analysis that is simultaneously (a) defined for the more general Clarke Jacobian, thus supporting both differentiable and non-differentiable, but still Lipschitz functions, (b) extends to all arithmetic operations and is defined for branching beyond just min and max, (c) is fully compositional by leveraging an interval abstraction of forward-mode dual numbers, and (d) is integrated in a fully end-to-end manner for practical tasks that no prior work could tackle.

Results

We implement DeepJ and evaluate it on two tasks:

- *Lipschitz Robustness:* Certifies bounds on the local Lipschitz constant of a given input region.
- *Local Optimization Landscape:* Uses the Clarke Jacobian to analyze a function's local geometry in a specified input region to determine the absence of stationary points.

We apply DeepJ to neural networks with both fully-connected and convolutional layers that are trained on the CIFAR10 [49] and MNIST [50] datasets. For each of our analyses, we compose the neural networks with three perturbation functions: *haze* (which models images as foggy), *contrast variation* (which accentuates the differences between bright and dark pixels), and *rotation* (which rotates the image by a specified angle θ with bilinear interpolation). Analyzing the Jacobian of a network with respect to these perturbations is out of reach of the existing techniques.

For each perturbation, DeepJ leverages its fully localized analysis, which computes Clarke Jacobians solely for the specified input region, to obtain local Lipschitz constants that can be up to several orders of magnitude smaller than a baseline analysis based on [51] (which multiplies a network's global Lipschitz constant by the perturbation's local Lipschitz constant instead of being fully localized). DeepJ can also extend the analysis to compositions of multiple perturbations. Furthermore, the localized Jacobian analysis can certify the absence of stationary points in a network's optimization landscape.

DeepJ's parallel CPU-based implementation is efficient: it can precisely analyze 100 CIFAR10 images on our largest convolutional network containing $>$ 62K neurons within a median time of 15 seconds each for haze and contrast variations, and under 1.4 minutes for rotation. It can also compute precise results for 10 CIFAR10 images on the same network within a median time of 9 seconds for contrast variation composed with haze, and under 51 seconds for contrast variation or haze composed with rotation. Our largest network has the same architecture as the one commonly handled by state-of-the-art CPU-based robustness verifiers [52, 53, 54]. The differences in the computed constants between the DeepJ versions with and without sound floating-point rounding are negligible, with 2.8-4.1x execution time overhead for the sound version.

Contributions

This chapter makes the following main contributions:

1. A new dual number-based interval abstraction for analyzing Clarke Jacobians. Our domain soundly over-approximates the Clarke Generalized Jacobian of locally Lipschitz and piecewise differentiable functions, allowing it to soundly handle functions like max, ReLU, and limited branches. Furthermore, as our abstraction is defined for sets of points, we can analyze local properties of locally Lipschitz functions beyond the scope of prior work.

2. A novel Clarke Jacobian analysis and Lipschitz certification of neural networks com-

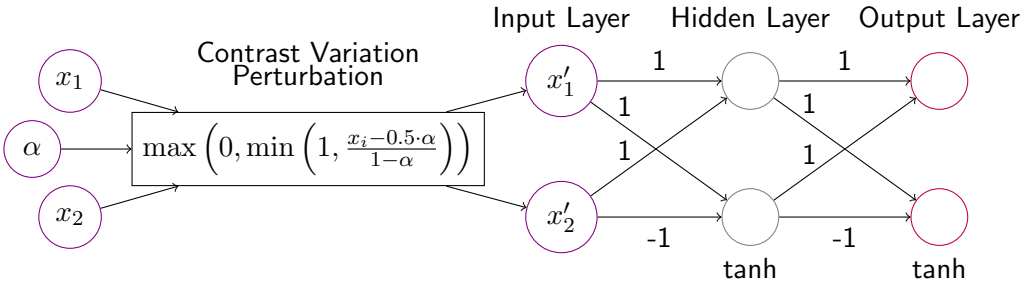


Figure 2.1: Composition of a Neural Network with the Contrast Variation Perturbation

posed with non-differentiable perturbations.

3. A scalable, optionally floating-point sound, implementation of our method which supports both convolutional and fully-connected neural networks.

4. An extensive evaluation against multiple perturbation types on several deep neural networks, showing that DeepJ can (a) achieve orders of magnitude tighter bounds on local Lipschitz constants compared to a baseline analysis and (b) certify the absence of stationary points within a given input region.

DeepJ is available at `https://github.com/uiuc-arc/DeepJ`. Our implementation exploits CPU-level parallelism. We believe that DeepJ can be easily parallelized over GPUs to boost the scalability to even larger architectures, such as those considered by GPU-based verifiers [55]. The GPU extension of DeepJ may help train networks to be robust to semantic non-differentiable perturbations like rotation and contrast variation, which is beyond the reach of existing robust training methods [56, 57, 58, 59, 60].

2.2 EXAMPLE

In this section, we start with a small illustrative example that showcases a real-world use of our abstraction for a scenario not handled by any prior work.

Running Example: Contrast Variation Perturbation

We consider the simple fully-connected network shown in Fig. 2.1. For simplicity, the network takes two inputs in the input layer, and we assume the network has been fully trained. The network contains a single hidden layer and all activation functions in both the hidden and output layers are tanh (with no biases). Most importantly, we compose the network with a perturbation function modeling contrast variation. We are interested in

knowing precisely how sensitive the network's outputs are to inputs that are perturbed by contrast variation, which often arises when the image passed to a network was obtained with a fixed aperture lens [61]. Hence, instead of passing input image pixels x_1 and x_2 directly into the input layer, we pass their *perturbed* values, x_1' and x_2'. The contrast variation perturbation for a pixel x_i is given in [61] as

$$x_i' = \max\left(0, \min\left(1, \frac{x_i - 0.5 \cdot \alpha}{1 - \alpha}\right)\right) \tag{2.1}$$

where α specifies the amount of contrast variation. We apply this perturbation to every input pixel (there are only two in this example). We can thus think of the perturbation as a function of the perturbation parameter α.

Jacobian Analysis

We will measure the sensitivity of the network with respect to the perturbation by computing the Jacobian of the composition of the network with the perturbation. This allows us to analyze how sensitive and robust the network's outputs are with respect to a change in α in some local region. For this example, the local region we are interested in is when $\alpha \in [0, 0.1]$. This allows us to analyze what happens when the amount of perturbation ranges from none up to a modest amount. Because of the combination of both non-differentiable (max and min) and differentiable (tanh) functions, as well as the division in the perturbation function, computing a Jacobian for the composition of the network and the perturbation is beyond the capabilities of existing frameworks.

Abstract Domain

To perform the analysis, we need to compute bounds on the Jacobian of the composition of the network and the perturbation. However, this is complicated by the fact that (a) we cannot settle for the derivatives at a single point (as standard automatic differentiation gives) and instead need a bound on the derivatives for an entire input region, (b) max and min are Lipschitz, but not differentiable, thus we need a more general notion of differentiation that works for such functions, and (c) the analysis cannot be restricted to only neural networks, since it needs to be able to compositionally handle arbitrary combinations of functions. As mentioned, prior work cannot address these challenges, thus our solution necessitates a novel abstract domain. The full formalism is described in Sections 2.5 and 2.6.

Our abstract domain associates to each variable an interval bounding the variable itself

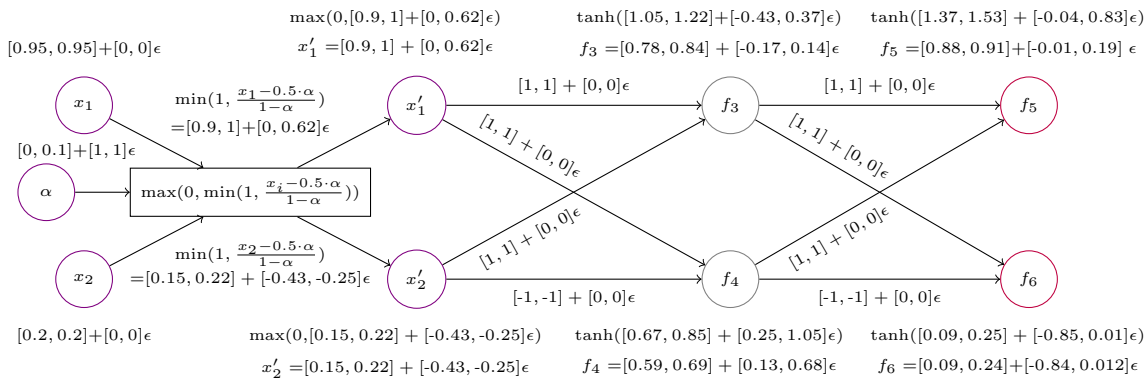


Figure 2.2: The Dual Interval Abstraction of the Neural Network and Perturbation Function from Fig. 2.1

and an interval bounding that variable's Clarke derivative via a *dual interval* of the form $[a, b] + [c, d]\epsilon$, where $a, b, c, d \in \mathbb{R} \cup \{\pm\infty\}$, $a \leq b$ and $c \leq d$. We will call $[a, b]$ the real part and $[c, d]$ the dual part. Dual intervals are an adapted interval-domain abstraction of the canonical dual numbers of forward-mode automatic differentiation. The key benefit of this approach is that we can leverage an existing numerical system to track derivatives instead of relying on non-extensible, ad-hoc approaches as in [15, 41, 46]. However, a naive adaptation is not sufficient, as one still has to contend with non-differentiable functions like max. Furthermore, all primitive operations must be reinterpreted for dual interval arithmetic.

Abstract Interpretation of the Perturbation and Network

We now step through the abstract interpretation of the composition of the neural network and the contrast variation perturbation, which is detailed in Fig. 2.2. As every term in the abstract domain must be a dual interval, to perform the analysis, we first must lift the constant edge weights w_i to dual intervals of the form $[w_i, w_i] + [0, 0]\epsilon$, as shown in Fig. 2.2. Upon lifting all constants and edge weights to the abstract domain, we start from the inputs x_1, x_2 and the perturbation parameter α. As our goal is to compute the sensitivity solely with respect to α, we set its dual part to $[1, 1]\epsilon$. This means that we treat x_1 and x_2 as constant; since we do not need to compute derivatives with respect to x_1 and x_2, the analysis sets their dual part to $[0, 0]\epsilon$. Furthermore, as x_1 and x_2 are pixel values of some fixed image, their real parts are just the degenerate interval of their pixel intensities: $[0.95, 0.95]$ and $[0.2, 0.2]$, respectively.

We begin by propagating the dual interval inputs through the contrast variation per-

turbation function in Eq. 2.1. This requires that all arithmetic operations $(+, -, \cdot, /)$ and function primitives (min and max) be redefined for dual intervals. Section 2.6 presents a full formalization of dual intervals.

For each pixel x_i, we first compute $\frac{x_i - 0.5 \cdot \alpha}{1-\alpha}$ inside the min function. Though not shown in the function, the constants 0.5 and 1 are actually interpreted as the dual intervals $[0.5, 0.5] + [0, 0]\epsilon$ and $[1, 1] + [0, 0]\epsilon$, respectively. The numerator $x_1 - 0.5 \cdot \alpha$ abstractly evaluates to $[0.9, 0.95] + [\text{-}0.5, \text{-}0.5]\epsilon$ and likewise $x_2 - 0.5 \cdot \alpha$ evaluates to $[0.15, 0.2] + [\text{-}0.5, \text{-}0.5]\epsilon$. These follow from the rules of dual interval arithmetic: scaling by a constant scales a term's real and dual part, and dual interval addition between terms adds their respective real and dual components. Therefore, we have implicitly encoded the notion of linearity of the derivative.

To compute the entire quotient for each variable, we next perform dual interval division (described in Section 2.6.1). The terms $\frac{x_1 - 0.5 \cdot \alpha}{1-\alpha}$ and $\frac{x_2 - 0.5 \cdot \alpha}{1-\alpha}$ ultimately evaluate to $[0.9, 1.05] + [0.4, 0.62]\epsilon$ and $[0.15, 0.22] + [\text{-}0.43, \text{-}0.25]\epsilon$, respectively. Dual interval division implicitly encodes the quotient rule of differentiation.

Non-Differentiable Functions

Upon computing $\frac{x_1 - 0.5 \cdot \alpha}{1-\alpha}$ and $\frac{x_2 - 0.5 \cdot \alpha}{1-\alpha}$, we now must take the min of each with $[1, 1] + [0, 0]\epsilon$. This is highly challenging as min is not differentiable. To resolve this, we must use a more general notion of differentiation, specifically the Clarke Jacobian [39], which has also recently emerged in the programming languages literature [38, 44]. The Clarke Jacobian can compute a generalized derivative for non-differentiable, but Lipschitz functions like min, max, ReLU, abs, and even functions defined by conditional branching statements (provided one proves such functions are continuous and piecewise differentiable). The Clarke Jacobian does so by returning a *convex set of points* as the "derivative" instead of just a single point. To perform the Clarke differentiation through the min function, one takes the standard derivative of whichever of the min function's two arguments attains the minimum. The caveat is that if both arguments attain the minimum, one must take the *convex hull* of both arguments' derivatives. However, this becomes even more difficult for the interval domain, as due to the inherent uncertainty when two intervals overlap, it is possible that either could attain the minimum. Thus, if the real parts of two dual intervals overlap, our analysis must take the convex hull of their respective dual parts.

Recall that for x_1 we must take the min of $[1, 1] + [0, 0]\epsilon$ and $\frac{x_1 - 0.5 \cdot \alpha}{1-\alpha} = [0.9, 1.05] + [0.4, 0.62]\epsilon$. In this case, the real parts overlap, so we must take the convex hull (denoted by **co**) of their

respective dual parts: $[0, 0]$ and $[0.4, 0.62]$, as shown in (multiline) Equation 2.2.

$$\begin{aligned}\min\left(1, \frac{x_1 - 0.5 \cdot \alpha}{1 - \alpha}\right) &= \min([1, 1] + [0, 0]\epsilon, [0.9, 1.05]+[0.4, 0.62]\epsilon) \\ &= \min([1, 1], [0.9, 1.05]) + \mathbf{co}([0, 0], [0.4, 0.62])\epsilon \\ &= [0.9, 1]+[0, 0.62]\epsilon\end{aligned} \tag{2.2}$$

For x_2, when computing the min of $[1, 1] + [0, 0]\epsilon$ and $\frac{x_2 - 0.5 \cdot \alpha}{1 - \alpha}$ =$[0.15, 0.22] + [\text{-}0.43, \text{-}0.25]\epsilon$, the real parts do not intersect, thus the result is exactly just $[0.15, 0.22] + [\text{-}0.43, \text{-}0.25]\epsilon$.

Finally, as the contrast variation perturbation is also composed with the max function, we must repeat the same procedure, albeit with max instead of min. For x_1 and x_2, we compute their perturbed values x_1' and x_2' as:

$$x_1' = \max([0, 0] + [0, 0]\epsilon, [0.9, 1]+[0, 0.62]\epsilon) = [0.9, 1] + [0, 0.62]\epsilon$$

$$x_2' = \max([0, 0] + [0, 0]\epsilon, [0.15, 0.22] + [\text{-}0.43, \text{-}0.25]\epsilon) = [0.15, 0.22] + [\text{-}0.43, \text{-}0.25]\epsilon \tag{2.3}$$

Propagation through the Network

Upon computing the perturbed inputs x_1' and x_2', we propagate their abstracted values through the network itself. For simplicity in presentation, we give each network node a fresh variable name (f_3-f_6). To compute the value of f_3, we first multiply x_1' and x_2' by the corresponding edge weights using dual interval multiplication and sum incoming terms, resulting in the dual interval $[1.05, 1.22]+[\text{-}0.43, 0.37]\epsilon$. Then, we apply the dual interval lifting of tanh to the argument $[1.05, 1.22]+[\text{-}0.43, 0.37]\epsilon$. The dual interval lifting of tanh applies the interval lifting of tanh (where $\tanh([a, b]) = [\tanh(a), \tanh(b)]$) to the real part of its argument, then applies the interval lifting of the closed-form derivative $(1 - \tanh^2)$ to the dual part. This computation is shown in (multiline) Equation 2.4:

$$\begin{aligned}f_3 = \tanh([1.05, 1.22]+[\text{-}0.43, 0.37]\epsilon) &= \tanh([1.05, 1.22]) + \Big((1 - \tanh^2([1.05, 1.22])) \cdot [\text{-}0.43, 0.37]\Big)\epsilon \\ &= [0.78, 0.84] + [\text{-}0.17, 0.14]\epsilon\end{aligned} \tag{2.4}$$

The dual part of the result is also multiplied by the dual part of the input ($[\text{-}0.43, 0.37]$), implicitly encoding the chain rule. In this example, f_3 evaluates to $[0.78, 0.84]+[\text{-}0.17, 0.14]\epsilon$.

Likewise for f_4, the input to the tanh function is the sum of x_1' and x_2' scaled by the edge weights, which is just $[0.67, 0.85] + [0.25, 1.05]\epsilon$. Hence, f_4 evaluates to $[0.59, 0.69] + [0.13, 0.68]\epsilon$. As our analysis is fully compositional, we easily repeat this procedure for the

subsequent (final) layer. Multiplying f_3 and f_4 by their respective edge weights, then passing these values to the tanh activations, allows us to determine that f_5 =[0.88, 0.91]+[-0.01, 0.19] ϵ and f_6 =[0.09, 0.24]+[-0.84, 0.012]ϵ.

Interval Clarke Jacobian

Upon computing the outputs f_5 and f_6, we take their dual parts as the Interval Clarke Jacobian. This is because we show in Section 2.6.3 that abstractly evaluating functions with dual intervals is equivalent to computing the Interval Clarke Jacobian, which in turn soundly over-approximates the true Clarke Jacobian. Since we are modeling the composition of the network and the perturbation as a function of only α (while holding all other inputs fixed), the Interval Clarke Jacobian is a 2×1 interval matrix, as the network has 2 outputs. In this case, it is [[-0.01, 0.19],[-0.84, 0.012]]^T.

Practical Applications

Upon computing this over-approximation of the Clarke Jacobian, we can use it for several practical applications. For instance, we can compute the local Lipschitz constant in the region $\alpha \in [0, 0.1]$ by taking the maximum norm of the Interval Clarke Jacobian, which intuitively gives us a point summary of the network's sensitivity to perturbations by α in this local region. For this example, the Lipschitz constant (with respect to the ℓ_∞-norm) evaluates to 0.84.

We can also use the over-approximation of the Clarke Jacobian to analyze the local landscape of the composition of the network and the perturbation for stationary points. If a point is a local extremum, then the Clarke Jacobian at that point contains $\mathbf{0}$. Therefore, if any entry of the Interval Clarke Jacobian does not contain 0, it certifies that no point in the input range is a local extremum. In this example, as both entries contain 0, the analysis determines that the input region $\alpha \in [0, 0.1]$ could still contain a local extremum.

2.3 PRELIMINARIES

We define all the mathematical preliminaries needed to describe automatic differentiation, as well as the Clarke Jacobian. We start with the definition of the standard Jacobian.

Definition 2.1. The Jacobian of a function $f : \mathbb{R}^m \to \mathbb{R}^n$ differentiable at a point $\mathbf{x_0} \in \mathbb{R}^m$

is

$$\mathbf{J}(f, \mathbf{x_0}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} |_{x=\mathbf{x_0}} & \cdots & \frac{\partial f_1}{\partial x_m} |_{x=\mathbf{x_0}} \\ \cdots & \cdots & \cdots \\ \frac{\partial f_n}{\partial x_1} |_{x=\mathbf{x_0}} & \cdots & \frac{\partial f_n}{\partial x_m} |_{x=\mathbf{x_0}} \end{bmatrix} \tag{2.5}$$

When $m = n = 1$, the Jacobian is merely the classical derivative: $\mathbf{J}(f, \mathbf{x_0}) = \frac{df}{dx}|_{x=\mathbf{x_0}}$.

Dual Numbers

The question then arises, how do we automatically compute this Jacobian? The most popular method is via *Automatic Differentiation*, or AD. AD has two modes: reverse and forward [5]. The former recursively evaluates derivatives of sub-expressions, and can be thought of as a generalization of *backpropagation*. We focus on the latter, as forward-mode automatic differentiation is much easier to implement and excels when the function's input dimension is small (as will be in our use cases). To implement forward-mode automatic differentiation, one may overload all primitive arithmetic operations to work on *dual numbers* [62, 63], which we now detail in Definition 2.2.

Definition 2.2. Dual numbers are numbers of the form $a + b\epsilon$, where $a, b \in \mathbb{R}$ and ϵ is a symbolic variable (akin to i for imaginary numbers). We denote the set of all dual numbers as $\mathbb{D}$. Dual number arithmetic is given by the following rules, shown in Eqs. 2.6-2.8:

$$(a + b\epsilon) + (c + d\epsilon) = (a + c) + (b + d)\epsilon \tag{2.6}$$

$$(a + b\epsilon) \cdot (c + d\epsilon) = (ac) + (ad + bc)\epsilon \tag{2.7}$$

$$(a + b\epsilon)/(c + d\epsilon) = \left(\frac{a}{c}\right) + \left(\frac{bc - ad}{c^2}\right)\epsilon \tag{2.8}$$

The above arithmetic rules for dual numbers implicitly encode linearity, the product rule, and the quotient rule in the computation of their dual part. To access the real part of a dual number, we write $fst(a + b\epsilon) = a$, and likewise the dual part is accessed by $snd(a + b\epsilon) = b$. For any differentiable function $f : \mathbb{R} \to \mathbb{R}$, we use the standard lifting of f to dual numbers $f : \mathbb{D} \to \mathbb{D}$ given in Eq. 2.9 as

$$f(a + b\epsilon) = f(a) + (f'(a) \cdot b)\epsilon \tag{2.9}$$

Therefore, the dual part of a dual number corresponds to the value of the function's derivative evaluated at the real part, a. Further, multiplying the derivative $f'(a)$ by the existing dual part b implicitly encodes the chain rule of calculus.

2.3.1 Lipschitz Continuity

We subsequently show how to extend the previous concepts to non-differentiable but locally Lipschitz functions. Hence, we first define the local Lipschitz property.

Definition 2.3. A function $f : \mathbb{R}^m \to \mathbb{R}^n$ is locally Lipschitz on $X \subseteq \mathbb{R}^m$ if there exists a positive constant $K^{\alpha,\beta} \in \mathbb{R}_{>0}$ such that for any $x_1, x_2 \in X$ we have

$$\|f(x_1) - f(x_2)\|_\beta \leq K^{\alpha,\beta}\|x_1 - x_2\|_\alpha \tag{2.10}$$

where $\|\cdot\|_\alpha$ and $\|\cdot\|_\beta$ are arbitrary p-norms over $\mathbb{R}^m$ and $\mathbb{R}^n$, respectively. Furthermore, if for a given point $x_0 \in \mathbb{R}^m$, there exists a positive real $\delta > 0$ such that f is locally Lipschitz within a ball of radius δ centered at x_0, we say *f is Lipschitz near x_0*.

Lipschitz Constant

The constant $K^{\alpha,\beta}$ is called the (local) Lipschitz constant, which provides a formal bound on how much a function's output can change (measured by the $\|\cdot\|_\beta$ norm) given a change in input (measured by the $\|\cdot\|_\alpha$ norm). The constant can easily be obtained once one has the Jacobian $\mathbf{J}$. For a Lipschitz function $f : \mathbb{R}^m \to \mathbb{R}^n$ that is differentiable, one can compute the local Lipschitz constant on a region X by taking the maximum dual norm of the Jacobian over X:

$$K^{\alpha,\beta} = \sup_{x \in X} \|\mathbf{J}(f, x)\|_{\alpha,\beta} \tag{2.11}$$

where the dual norm of any matrix $M \in \mathbb{R}^{n \times m}$ is given as $\|M\|_{\alpha,\beta} = \sup_{\|v\|_\alpha \leq 1} \|Mv\|_\beta$. For common values of α and β, there is a closed-form expression for $\|M\|_{\alpha,\beta}$. For instance, $\|M\|_{1,1} = \max_{1 \leq j \leq m}(\sum_{i=1}^{n} |M_{i,j}|)$, thus we simply take the norm of the Jacobian (over all points in X) that has the maximum absolute column sum.

2.3.2 Clarke Generalized Jacobian

However, the following question arises: what if we need to compute the derivative at a point where it is not defined, such as $\mathrm{ReLU}(x)$ at $x = 0$? Can one extend Jacobians (and methods to compute them) to non-differentiable functions? We follow recent work by [38] and employ the notion of the Clarke Generalized Jacobian [39] for this extension. Intuitively, this idea generalizes the notion of a Jacobian to non-differentiable, but still Lipschitz continuous functions.

Convexity

The Clarke Jacobian of a function $f : \mathbb{R}^m \to \mathbb{R}^n$ evaluates to a *convex* set of $n \times m$ matrices, hence we define the following operators. Let $\mathbf{Co}(\mathbb{R}^{n\times m})$ denote all convex sets of $n \times m$ real matrices. Further, let $\mathbf{co} : \mathcal{P}(\mathbb{R}^{n\times m}) \to \mathbf{Co}(\mathbb{R}^{n\times m})$ be the convex hull operator, which given a set of matrices, takes their convex hull. We can now define the Clarke Generalized Jacobian.

Definition 2.4. (Clarke Thm. 2.5.1 [39]) Let $f : \mathbb{R}^m \to \mathbb{R}^n$ be a function that is locally Lipschitz at $\mathbf{x_0} \in \mathbb{R}^m$, where the set of non-differentiable points of f has Lebesgue measure 0 (we denote that set as $S \subset \mathbb{R}^m$). The Clarke Generalized Jacobian of f at the point $\mathbf{x_0}$ is denoted with the following signature $\partial_c : (\mathbb{R}^m \to \mathbb{R}^n) \times \mathbb{R}^m \to \mathbf{Co}(\mathbb{R}^{n\times m})$, and is given as:

$$\partial_c(f, \mathbf{x_0}) = \mathbf{co}\{\lim_{j\to\infty} \mathbf{J}(f, \mathbf{x}_j) : \lim_{j\to\infty} \mathbf{x}_j = \mathbf{x_0} \text{ and } \mathbf{x}_j \notin S \text{ for all } j \in \mathbb{N}\} \tag{2.12}$$

We first detail this definition when $\mathbf{x}_0$ is a point of non-differentiability of f. Intuitively, since we cannot compute the actual Jacobian $\mathbf{J}$ at a point of non-differentiability, we instead take any *sequence of points* that converges to that point of non-differentiability, such that the Jacobian is defined for all points in that sequence; we then compute what the Jacobians evaluated at those points in the sequence converge to. The Clarke Jacobian is then just the convex hull over all such sequences' respective limiting Jacobians. Thus, we obtain a convex set of matrices. When the function *is* differentiable at the point $\mathbf{x}_0$, this limit reduces to exactly the standard Jacobian at $\mathbf{x}_0$ (and the convex hull becomes superfluous), hence $\partial_c(f, \mathbf{x_0})$ will be a singleton set containing exactly $\mathbf{J}(f, \mathbf{x_0})$.

Example 2.1. Let $f(x) = \text{ReLU}(x)$. Then $\partial_c(f, 0) = \mathbf{co}\{0, 1\} = [0, 1]$.

2.3.3 Clarke Jacobian Properties

One might be tempted to think that the Clarke Jacobian simply just lifts all the rules of the ordinary Jacobian to operate on convex sets, but this is not the case. There exist key cases where the Clarke Jacobian does *not* satisfy the analogous rule for ordinary Jacobians [39]. For instance, the notion of partial Clarke Jacobians (denoted as ∂_c^i when taken with respect to variable x_i) does not always obey an equality or containment relation, meaning in general, they are incomparable:

$$\partial_c(f, (x_1, x_2)) \not\lessgtr \partial_c^1(f, (x_1, x_2)) \times \partial_c^2(f, (x_1, x_2)) \tag{2.13}$$

Consequently, as pointed out by [43], this means that computing the Clarke Jacobian for multi-variate functions is not as simple as just concatenating the partial Clarke Jacobians ∂_c^i

into a matrix (as is done with the classical Jacobian). Likewise, a chain rule for the Clarke Jacobian only holds in select cases. Because of this, our language only uses expressions where the Clarke Jacobian *does* always obey an explicit, closed-form rule analogous to the ordinary Jacobian, which ensures that the analysis is fully compositional, *by construction*. Hence, by restricting to these primitives, we do not need to compute gradients of arbitrary composite functions in situations where the classical Jacobian would require differentiating with respect to a single variable while holding all others as constants (since the Clarke Jacobian does not obey this rule).

However, as restrictive as this observation sounds, it still allows for us to have a very expressive set of language primitives where the semantics can be defined precisely.

2.4 LANGUAGE SYNTAX AND SEMANTICS

We describe our differentiable programming language, which is based upon λ_S [38] but with additional branching primitives. Our language is first-order and purely functional (no side-effects), yet expressive enough to encode neural networks and locally Lipschitz perturbations.

2.4.1 Syntax

Figure 2.3 presents the syntax of the language. DeepJ syntactically supports standard arithmetic operations, differentiable function primitives, and limited branching for encoding Lipschitz but non-differentiable functions like min and max. One can encode neural networks in DeepJ; however, instead of encoding them as a series of edge-weight matrix multiplications, our syntax constructs them inductively via compositions: $\circ$, Cartesian products: $\times$, and branching (e.g., for ReLU networks).

Arithmetic Operations

We support all of the key arithmetic primitives: addition, multiplication, and division. While syntactically speaking, these are only defined for functions of a single output variable, one can easily encode multi-variable versions (e.g., vector addition) by also making use of the Cartesian product, $\times$, where $(f_1 \times f_2)(x) = (f_1(x), f_2(x))$. For example, $(f_1 \times f_2) + (f_3 \times f_4) = (f_1 + f_3) \times (f_2 + f_4)$.

$$
\begin{array}{rcl}
f & ::= & f_1 \times f_2 \\
& | & f_1 + f_2 \\
& | & f_1 \cdot f_2 \\
& | & 1/f_1 \\
& | & C^1 \circ f_1 \\
& | & f_0 > c \ ? \ f_1 : f_2 \\
& | & x_i \\
& | & c \in \mathbb{R} \\
\\
C^1 & ::= & \sin(x) \mid \cos(x) \\
& | & e^x \mid \log(x) \mid \text{sqrt}(x) \\
& | & \sigma(x) \mid \tanh(x)
\end{array}
$$

$$\frac{}{\Gamma \vdash c : \mathbb{R}^m \to \mathbb{R}} \qquad \frac{}{\Gamma \vdash x_i : \mathbb{R}^m \to \mathbb{R}}$$

$$\frac{\Gamma \vdash f_1 : \mathbb{R}^m \to \mathbb{R}^n \quad \Gamma \vdash f_2 : \mathbb{R}^m \to \mathbb{R}^p}{\Gamma \vdash f_1 \times f_2 : \mathbb{R}^m \to \mathbb{R}^{n+p}}$$

$$\frac{\Gamma \vdash f_1 : \mathbb{R}^m \to \mathbb{R} \quad \Gamma \vdash f_2 : \mathbb{R}^m \to \mathbb{R}}{\Gamma \vdash f_1 + f_2 : \mathbb{R}^m \to \mathbb{R}}$$

$$\frac{\Gamma \vdash f_1 : \mathbb{R}^m \to \mathbb{R} \quad \Gamma \vdash f_2 : \mathbb{R}^m \to \mathbb{R}}{\Gamma \vdash f_1 \cdot f_2 : \mathbb{R}^m \to \mathbb{R}}$$

$$\frac{\Gamma \vdash f_1 : \mathbb{R}^m \to \mathbb{R}}{\Gamma \vdash 1/f_1 : \mathbb{R}^m \to \mathbb{R}} \qquad \frac{\Gamma \vdash f_1 : \mathbb{R}^m \to \mathbb{R}}{\Gamma \vdash C^1 \circ f_1 : \mathbb{R}^m \to \mathbb{R}}$$

$$\frac{\Gamma \vdash f_0 : \mathbb{R}^m \to \mathbb{R} \quad \Gamma \vdash f_1 : \mathbb{R}^m \to \mathbb{R}^n \quad \Gamma \vdash f_2 : \mathbb{R}^m \to \mathbb{R}^n}{\Gamma \vdash f_0 > c \ ? \ f_1 : f_2 : \mathbb{R}^m \to \mathbb{R}^n}$$

Figure 2.3: Locally Lipschitz Function Syntax and Typing Rules

Differentiable Functions

Syntactically, we support as primitives all the standard primitive differentiable functions (e.g., e^x, $\sin(x)$, $\tanh(x)$, etc.). Hence, we denote these as C^1 because each function is C^1-smooth, meaning the function is continuous and differentiable everywhere on its domain, and the first derivative is also continuous everywhere on its domain.

Non-differentiable, Lipschitz Functions

We support a branching primitive, $f_0 > 0 \ ? \ f_1 : f_2$, to implement Lipschitz, but not necessarily differentiable functions such as abs, ReLU, min, and max (and by extension max-pooling). For example, abs can be implemented as $x > 0 \ ? \ x : -x$. However, one could easily define a discontinuous function such as $x > 0 \ ? \ 1 : 0$ using our syntax. Thus, for the computed Jacobian (and by extension Lipschitz constant) to be semantically meaningful and valid, we restrict the type of branching we support. We will later show that checking for these restrictions is undecidable in Section 2.4.2, and thus the responsibility of ensuring that the restrictions are satisfied rests upon the programmer or an (incomplete) program analysis.

$$\partial_c : \Big(\Big(\mathbb{R}^m \to \mathbb{R}^n\Big) \times \mathbb{R}^m\Big) \to \mathbf{Co}\Big(\mathbb{R}^{n\times m}\Big)$$

$$\begin{aligned}
\partial_c(f_1 \times f_2, \mathbf{x_0}) &\subseteq \begin{bmatrix}\partial_c(f_1, \mathbf{x_0}) \\ \partial_c(f_2, \mathbf{x_0})\end{bmatrix} \\
&\quad \text{if } f_1 \;\&\; f_2 \text{ Lipschitz near } \mathbf{x_0}, \text{ else } \top \\
\partial_c(f_1 + f_2, \mathbf{x_0}) &\subseteq \partial_c(f_1, \mathbf{x_0}) +_c \partial_c(f_2, \mathbf{x_0}) \\
&\quad \text{if } f_1 \;\&\; f_2 \text{ Lipschitz near } \mathbf{x_0}, \text{ else } \top \\
\partial_c(f_1 \cdot f_2, \mathbf{x_0}) &\subseteq f_1(\mathbf{x_0}) \cdot_c \partial_c(f_2, \mathbf{x_0}) +_c f_2(\mathbf{x_0}) \cdot_c \partial_c(f_1, \mathbf{x_0}) \\
&\quad \text{if } f_1 \;\&\; f_2 \text{ Lipschitz near } \mathbf{x_0}, \text{ else } \top \\
\partial_c(1/f_1, \mathbf{x_0}) &\subseteq -\partial_c(f_1, \mathbf{x_0}) /_c f_1(\mathbf{x_0})^2 \\
&\quad \text{if } f_1 \text{ Lipschitz near } \mathbf{x_0} \text{ and } f_1(\mathbf{x_0}) \neq 0, \text{ else } \top \\
\partial_c(C^1 \circ f_1, \mathbf{x_0}) &= \mathbf{J}(C^1, f_1(\mathbf{x_0})) \cdot_c \partial_c(f_1, \mathbf{x_0}) \\
&\quad \text{if } f_1 \text{ Lipschitz near } \mathbf{x_0} \text{ and } C^1 \text{ differentiable at } f_1(\mathbf{x_0}), \text{ else } \top \\
\partial_c(f_0 > c \;?\; f_1 : f_2, \mathbf{x_0}) &\subseteq \begin{cases} \partial_c(f_1, \mathbf{x_0}) & \text{if } f_0(\mathbf{x_0}) > c \\ \partial_c(f_2, \mathbf{x_0}) & \text{if } f_0(\mathbf{x_0}) < c \\ \mathbf{co}\Big(\partial_c(f_1, \mathbf{x_0}), \partial_c(f_2, \mathbf{x_0})\Big) & \text{otherwise} \end{cases} \\
&\quad \text{if } f_1 \;\&\; f_2 \text{ Lipschitz near } \mathbf{x_0} \text{ and} \\
&\quad f_1, f_2 \text{ agree on } \{x : f_0(x) = c\}, \text{ else } \top \\
\partial_c(x_i, \mathbf{x_0}) &= \{\mathbf{e}_i\} \\
&\quad \text{provided } i \in \{1, ..., m\} \\
\partial_c(c, \mathbf{x_0}) &= \{\mathbf{0}\}
\end{aligned}$$

Figure 2.4: Clarke Jacobian Rules

Lastly, even though we restrict the type of branches we support, more complex branches with arbitrary Boolean predicates can be systematically desugared into simpler ones. For example, the branch $\big(c_1 < x \wedge x < c_2\big)\ ?\ f_1 : f_2$ can be desugared into $c_1 < x\ ?\ \big(x < c_2\ ?\ f_1 : f_2\big) : f_2$. Disjunctions and negations of Booleans can be encoded similarly.

Type System

As our language only employs real-valued functions, the typing rules for a function are simple and based on standard real-valued arithmetic. For instance, when adding two functions, their dimensions must agree. Likewise, when dividing by a function f_1 (as in $1/f_1$),

the output dimension of the function f_1 must be 1. The full typing rules can be seen in Fig. 2.3, where Γ corresponds to the typing context that maps all arithmetic expressions (including intermediate ones) to their types.

2.4.2 Standard Interpretation

As our language is an augmented differentiable programming language, the semantic interpretation of a function f is its derivative, which in our case corresponds to its Clarke Generalized Jacobian $\partial_c(f, \cdot)$. Our language is inspired by [38], hence we follow their convention of lifting the Clarke Jacobian to become a *total* function by defining the result to be $\top$ whenever ∂_c would be undefined, such as trying to evaluate the Clarke Jacobian of $\log(x)$ at $x = 0$. Because of this, $\top$ corresponds to the entire space of all real $n \times m$ matrices ($\mathbb{R}^{n\times m}$).

The semantic rules for recursively defining the Clarke Generalized Jacobian of each language primitive are shown in Figure 2.4. Unlike the regular Jacobian, these rules use $\subseteq$ instead of equality. This means that the *exact* Clarke Jacobian is not computable; however, our end goal is to compute a sound over-approximation.

Convex Arithmetic Operations

We denote $+_c : \mathbf{Co}(\mathbb{R}^{n\times m}) \times \mathbf{Co}(\mathbb{R}^{n\times m}) \to \mathbf{Co}(\mathbb{R}^{n\times m})$ to be the addition of two convex sets (Minkowksi addition), where $A +_c B = \{a + b : a \in A, b \in B\}$. Likewise, we denote $\cdot_c : \mathbb{R} \times \mathbf{Co}(\mathbb{R}^{n\times m}) \to \mathbf{Co}(\mathbb{R}^{n\times m})$ where $v \cdot_c A = \{v \cdot a : a \in A\}$ and $/_c : \mathbf{Co}(\mathbb{R}^{n\times m}) \times \mathbb{R}_{\neq 0} \to \mathbf{Co}(\mathbb{R}^{n\times m})$ where $A /_c v = \{\frac{1}{v} \cdot a : a \in A\}$ to be convex scalar multiplication and division, respectively. Lastly, as mentioned, we let $\mathbf{co} : \mathcal{P}(\mathbb{R}^{n\times m}) \to \mathbf{Co}(\mathbb{R}^{n\times m})$ be the convex hull operator that takes the convex hull of a set of matrices. We now detail the rules of Fig. 2.4.

Variables and Constants

The Clarke Jacobian of a single variable x_i is $\{1\}$, but if we compute it with respect to $\mathbf{x_0} \in \mathbb{R}^m$, the Clarke Jacobian will be m-dimensional, hence denoted as $\{\mathbf{e}_i\}$, where the i^{th} entry of $\mathbf{e}_i$ is 1 and all other $m - 1$ entries are 0. Equivalently, $f \triangleq x_i$ can be thought of as a projection function going from $\mathbb{R}^m \to \mathbb{R}$ that takes the i^{th} component of $\mathbf{x_0}$. Similarly, the Clarke Jacobian of any constant is just the vector of m zeroes, denoted as $\{\mathbf{0}\}$.

Cartesian Product

The Clarke Jacobian of the Cartesian product of two functions (defined over the same input) is a *subset* of the matrix concatenation of each function's respective Clarke Jacobians. This follows directly from proposition 2.6.2 (e) of [39].

Addition

The Clarke Jacobian does not obey exact linearity; however, the Clarke Jacobian of the sum of two functions is *contained* in the Minkowski sum of each function's respective Clarke Jacobian. This rule follows directly from Proposition 2.3.3 of [39].

Multiplication and Division

The Clarke Jacobian follows both a product and quotient rule, but as with the other rules, the relationship is of containment instead of strict equality. These follow directly from Propositions 2.3.13 and 2.3.14 of [39].

Composition

To ensure our language is fully compositional, we can exploit the fact that the Clarke Jacobian follows a chain rule, when the outermost function is C^1-smooth. This result follows directly from Theorems 2.3.9 and 2.6.6 of [39].

C^1 Functions

For the C^1 primitive functions, the Clarke Jacobian reduces to the standard Jacobian $\mathbf{J}(C^1, \cdot)$. Furthermore, one can catch erroneous behavior by leveraging knowledge about the known domain of each C^1 primitive function. For instance, if one tries to evaluate $\partial_c(\text{sqrt}(-ReLU(x_i)), 1)$, this would require evaluating $\mathbf{J}(\text{sqrt}, -1)$. However, since -1 lies outside the domain of sqrt, this result is undefined; thus, the expression will evaluate to $\top$, and errors will be caught at this step. One can then propagate $\top$ up the remainder of a function's expression tree, since all other rules in Fig. 2.4 first check if any sub-expression evaluates to $\top$ (in which case they will also evaluate to $\top$).

Branching

Branching is absent in Clarke's original formulation. Furthermore, λ_S [38] and [44] also do not support a branching primitive. This is because a branch can introduce the possibility of encoding a discontinuous function, such as $f(x) = x > 0\ ?\ 1 : 0$. As a branch can be thought of as "splitting" a function into two separate pieces, we need to ensure that the entire function is still locally Lipschitz continuous (on the region of interest containing $\mathbf{x_0}$) for it to have a well-defined Clarke Jacobian at $\mathbf{x_0}$. Thus, to formally establish the necessary conditions for the well-definedness of the Clarke Jacobian of a branching function, we use results from the theory of piecewise differentiable functions [43, 64]. We first state a useful lemma.

Lemma 2.1. Let $f : \mathbb{R}^m \to \mathbb{R}^n$ be a function expressible using the syntax of Figure 2.3 that does not contain branches. For any $\mathbf{x_0}$ where $\mathbf{J}(f, \mathbf{x_0})$ is defined, we have that $\partial_c(f, \mathbf{x_0}) = \{\mathbf{J}(f, \mathbf{x_0})\}$.

Proof. (Sketch) Since a function that does not have branches is a constant or only uses addition, composition with a C^1 function, multiplication, division, or the Cartesian product, one can directly compute $\mathbf{J}(f, \mathbf{x_0})$ using linearity, chain rule, product or quotient rule, or the direct computation of the derivative (for C^1 functions and constants), provided $\mathbf{J}(f, \mathbf{x_0})$ is well-defined (e.g., there is no division by 0). Furthermore, by Proposition 2.2.4 of Clarke [39], if a function has a well-defined Jacobian at a point, the Clarke Jacobian reduces to that value. QED.

We now formally define piecewise differentiability.

Definition 2.5. (Piecewise Differentiability [64]) A function $f : \mathbb{R}^m \to \mathbb{R}^n$ is piecewise differentiable on an open set $\mathcal{X} \subseteq \mathbb{R}^m$ if f is continuous on $\mathcal{X}$ and for every $x \in \mathcal{X}$ there exists an open neighborhood $O \subseteq \mathcal{X}$ and a finite number of differentiable functions, denoted $\{g_1, ..., g_k\}$, such that for any $x_o \in O$, $f(x_o) \in \{g_1(x_o), ..., g_k(x_o)\}$. We will refer to the set of differentiable functions $\{g_1, ..., g_k\}$ as the selection set.

Any function f that is differentiable on a set $\mathcal{X} \subseteq \mathbb{R}^m$ (meaning $\mathbf{J}(f, \cdot)$ exists) is trivially piecewise differentiable on $\mathcal{X}$, albeit with a single piece. A piecewise differentiable function has a well-defined Clarke Jacobian that can be given in terms of the convex hull of the standard Jacobian of constituent pieces, provided an *active set* is known a priori [64]. If f has the selection set of differentiable functions $\{g_1, ..., g_k\}$, then (by Proposition 4.3.1 of [64]):

$$\partial_c(f, \mathbf{x_0}) = \mathbf{co}\{\mathbf{J}(g_i, \mathbf{x_0}) \mid i \in A(f, \mathbf{x_0})\} \tag{2.14}$$

where $A(f, \mathbf{x_0}) \subseteq \{1, ..., k\}$ is the *active set*, which denotes at $\mathbf{x_0}$ which of the k selection functions satisfy $g_i(\mathbf{x_0}) = f(\mathbf{x_0})$. To adapt this to our setting, we start with the simplest branching function $f \triangleq f_0 > c \ ? \ f_1 : f_2$ where f_1, f_2 are branch-free (meaning they are compositions, sums, or products of C^1 functions). Since f_1 and f_2 do not contain branches, one can compute $\mathbf{J}(f_i, \mathbf{x_0})$ directly in accordance with Lemma 2.1. Hence, f_1 and f_2 *are* the selection set, as they are differentiable (by assumption) and the value of $f_0 > c \ ? \ f_1 : f_2$ for some x_0 will necessarily be in $\{f_1(x_0), f_2(x_0)\}$. Therefore, when writing a branch, the selection set is known *by construction*. So all that remains for f to be piecewise differentiable (and have a well-defined ∂_c) is to ensure that it is continuous, which is true provided $f_1(x) = f_2(x)$ for $\{x : f_0(x) = c\}$.

We do not need to restrict to cases where f_1 and f_2 are branch-free. We can nest branches arbitrarily deeply, provided they agree on the decision boundary $\{x : f_0(x) = c\}$, as all this does is increase the number of selection functions (or pieces) by finitely many. This is because we do not allow infinite recursion or while loops that would permit expressing countably infinite possible branches. Intuitively, we recursively "unroll" the nested branches into all possible innermost functions (which themselves will no longer contain branches). This ultimately yields a finite set of branch-free functions f_i for which we can compute $\mathbf{J}(f_i, \cdot)$ directly (as in Lemma 2.1). Using these notions, we can now formally describe necessary conditions for the Clarke Jacobian of the branching primitive to be well-defined.

Theorem 2.1. (Well-Definedness of the Clarke Jacobian of a Branch) The function given by the branch $f_0 > c \ ? \ f_1 : f_2$ is piecewise differentiable on an open set $\mathcal{X} \subseteq \mathbb{R}^m$ if f_1 is piecewise differentiable on $\{x \in \mathcal{X} : f_0(x) \geq c\}$, f_2 is piecewise differentiable on $\{x \in \mathcal{X} : f_0(x) \leq c\}$, and $f_1(x_0) = f_2(x_0)$ for all $x_0 \in \{x \in \mathcal{X} : f_0(x) = c\}$.

Proof. (Sketch) Since f_1 is piecewise differentiable on $\{x \in \mathcal{X} : f_0(x) \geq c\}$, it has some selection set $\{g_1, ..., g_k\}$; hence, $f_0 > c \ ? \ f_1 : f_2$ is also piecewise differentiable on $\{x \in \mathcal{X} : f_0(x) \geq c\}$, with the same selection set on that region. Likewise, since f_2 is piecewise differentiable on $\{x \in \mathcal{X} : f_0(x) \leq c\}$, it has some selection set $\{h_1, ..., h_l\}$; thus, $f_0 > c \ ? \ f_1 : f_2$ is also piecewise differentiable on $\{x \in \mathcal{X} : f_0(x) \leq c\}$, with the same selection set. Furthermore, since $\{x \in \mathcal{X} : f_0(x) = c\} \subseteq \{x \in \mathcal{X} : f_0(x) \geq c\}$ and $\{x \in \mathcal{X} : f_0(x) = c\} \subseteq \{x \in \mathcal{X} : f_0(x) \leq c\}$, the selection set for $\{x \in \mathcal{X} : f_0(x) = c\}$ is $\{g_1, ..., g_k, h_1, ..., h_l\}$. Lastly, since $x_0 \in \{x \in \mathcal{X} : f_0(x) = c\}$ implies $f_1(x_0) = f_2(x_0)$, then $f_0 > c \ ? \ f_1 : f_2$ is continuous on its entire domain. QED.

We now present the rule for computing the Clarke Jacobian for a branching function, using the notions of piecewise differentiability. Formally, for $f_0 : \mathbb{R}^m \to \mathbb{R}$, $f_1, f_2 : \mathbb{R}^m \to \mathbb{R}^n$,

and $\mathbf{x_0} \in \mathbb{R}^m$:

$$\partial_c(f_0 > c \ ? \ f_1 : f_2, \mathbf{x_0}) \subseteq \begin{cases} \partial_c(f_1, \mathbf{x_0}) & \text{if } f_0(\mathbf{x_0}) > c \\ \partial_c(f_2, \mathbf{x_0}) & \text{if } f_0(\mathbf{x_0}) < c \\ \mathbf{co}\Big(\partial_c(f_1, \mathbf{x_0}), \partial_c(f_2, \mathbf{x_0})\Big) & \text{otherwise} \end{cases} \tag{2.15}$$

When $f_0(\mathbf{x_0}) > c$, the active set is $A(f_0, \mathbf{x_0}) = \{f_1\}$; similarly, when $f_0(\mathbf{x_0}) < c$, the active set is $A(f_0, \mathbf{x_0}) = \{f_2\}$. Along the decision boundary, both f_1 and f_2 are in the active set, hence the Clarke Jacobian is the convex hull, **co**, of both of their respective Clarke Jacobians. In the rule shown in Eq. 2.15, we evaluate $\partial_c(f_1, \mathbf{x_0})$ and $\partial_c(f_2, \mathbf{x_0})$ instead of $\mathbf{J}(f_1, \mathbf{x_0})$ and $\mathbf{J}(f_2, \mathbf{x_0})$ as in Eq. 2.14. This recursive definition allows us to capture the notion of unrolling a nested branch, as once f_1 and f_2 are themselves branch free, $\partial_c(f_i, \mathbf{x_0})$ and $\mathbf{J}(f_i, \mathbf{x_0})$ become equivalent (by Lemma 2.1); thus, this equation would coincide with Eq. 2.14.

Example 2.2. We can encode $\text{ReLU}(x) \triangleq x > 0 \ ? \ x : 0$, and hence $\partial_c(x > 0 \ ? \ x : 0, 0) = \mathbf{co}(\partial_c(x, 0), \partial_c(0, 0)) = \mathbf{co}(\{1, 0\}) = [0, 1]$.

Despite the elegant theory of [64], which allows us to characterize the conditions and well-definedness of the Clarke Jacobian of a piecewise differentiable function (or branch in our language), the problem of *statically checking* that these conditions are satisfied is undecidable. This is because even the smaller problem of ensuring that a branch is continuous along the decision boundary is known to be undecidable [44, 65, 66]. As those works do not target the exact framework we focus on, we offer a short self-contained result:

Lemma 2.2. (Undecidability of Ensuring Piecewise Differentiability) Let $f_1 : \mathbb{R}^m \to \mathbb{R}^n$ and $f_2 : \mathbb{R}^m \to \mathbb{R}^n$ be arbitrary functions in the language, and let $\mathcal{X} \subseteq \mathbb{R}^m$ be an arbitrary set on which f_1 and f_2 are both defined. Checking if $f_1(x) = f_2(x)$ for all $x \in \mathcal{X}$ where $f_0(x) = c$ is undecidable.

Proof. (Sketch) Reduction using Richardson's Theorem [67]. QED.

Existing works can heuristically check for this condition using SMT solvers [66], or restrict the language to only pre-specified operators (e.g., max, min, ReLU and abs) [38, 44, 68]. Despite the difficulties introduced by a branching primitive, it will prove immensely useful in making the static analysis more precise, particularly for functions such as bilinear interpolation.

2.5 INTERVAL CLARKE JACOBIAN

Having now defined the syntax and the standard Clarke Jacobian, we now formalize a computable, sound over-approximation: the Interval Clarke Jacobian, ∂^{Int}. We will then show how to scalably implement our abstraction with an equivalent formulation that leverages a sound abstraction of forward-mode automatic differentiation.

2.5.1 Interval Domain

To soundly overapproximate the Clarke Jacobian, we use the classic interval domain [28] as it is fully computable and can soundly abstract the convex set of matrices corresponding to the original Clarke Generalized Jacobian, since interval matrices are convex sets.

Preliminaries

Denote the set of real-valued intervals of the form $[a, b]$ where $a, b \in \mathbb{R} \cup \{\pm\infty\}$ and $a \leq b$ as $\mathbf{I}\mathbb{R}$. The set of $n \times m$ matrices of intervals is denoted as $\mathbf{I}\mathbb{R}^{n\times m}$. We will use the notation $\widehat{\mathbf{x_0}}$ instead of $\mathbf{x_0}$ to distinguish matrices and vectors whose entries are intervals instead of scalars. Similarly, to denote the evaluation of a function f where all of its operations are lifted to interval arithmetic, we will write $f(\widehat{\mathbf{x_0}})$. To denote the lower and upper bounds of an interval $\widehat{\mathbf{x_0}} = [a, b]$, we will write $lb(\widehat{\mathbf{x_0}})$ and $ub(\widehat{\mathbf{x_0}})$, respectively. We denote $+_{\mathbf{I}\mathbb{R}}$, $\cdot_{\mathbf{I}\mathbb{R}}$, and $/_{\mathbf{I}\mathbb{R}}$ as the interval arithmetic versions of addition, multiplication, and division, respectively. Likewise, we denote $\sqcup : \mathbf{I}\mathbb{R} \times \mathbf{I}\mathbb{R} \to \mathbf{I}\mathbb{R}$ to be the standard interval join, which returns the smallest interval enclosing both arguments. We may also apply $\sqcup$ element-wise to matrices in $\mathbf{I}\mathbb{R}^{n\times m}$. Lastly, we denote $\top = \mathbf{I}\mathbb{R}^{n\times m}$, which is the entire space of $n \times m$ interval matrices. We now define the Interval Clarke Jacobian, ∂^{Int}.

Definition 2.6. The Interval Clarke Jacobian $\partial^{Int} : \left(\left(\mathbb{R}^m \to \mathbb{R}^n\right) \times \mathbf{I}\mathbb{R}^m\right) \to \mathbf{I}\mathbb{R}^{n\times m}$ for a function $f : \mathbb{R}^m \to \mathbb{R}^n$ and interval vector $\widehat{\mathbf{x}} \in \mathbf{I}\mathbb{R}^m$ is denoted $\partial^{Int}(f, \widehat{\mathbf{x_0}})$ and is given by the rules of Fig. 2.5.

The interpretation of the Jacobian of a function f is now an interval matrix that over-approximates the convex set of matrices corresponding to the original Clarke Jacobian. This will be necessary to be able to define the local neighborhood of points for which we want to abstractly analyze or compute a Lipschitz constant. We now detail each abstract transformer.

$$\partial^{Int} : \left(\left(\mathbb{R}^m \to \mathbb{R}^n\right) \times \mathbf{I}\mathbb{R}^m\right) \to \mathbf{I}\mathbb{R}^{n \times m}$$

$$\begin{aligned}
\partial^{Int}(f_1 \times f_2, \widehat{\mathbf{x_0}}) &= \begin{bmatrix}\partial^{Int}(f_1, \widehat{\mathbf{x_0}}) \\ \partial^{Int}(f_2, \widehat{\mathbf{x_0}})\end{bmatrix} \\
&\quad \text{if } f_1 \;\&\; f_2 \text{ Lipschitz on } \widehat{\mathbf{x_0}}, \text{ else } \top \\
\partial^{Int}(f_1 + f_2, \widehat{\mathbf{x_0}}) &= \partial^{Int}(f_1, \widehat{\mathbf{x_0}}) +_{\mathbf{I}\mathbb{R}} \partial^{Int}(f_2, \widehat{\mathbf{x_0}}) \\
&\quad \text{if } f_1 \;\&\; f_2 \text{ Lipschitz on } \widehat{\mathbf{x_0}}, \text{ else } \top \\
\partial^{Int}(f_1 \cdot f_2, \widehat{\mathbf{x_0}}) &= f_1(\widehat{\mathbf{x_0}}) \cdot_{\mathbf{I}\mathbb{R}} \partial^{Int}(f_2, \widehat{\mathbf{x_0}}) +_{\mathbf{I}\mathbb{R}} f_2(\widehat{\mathbf{x_0}}) \cdot_{\mathbf{I}\mathbb{R}} \partial^{Int}(f_1, \widehat{\mathbf{x_0}}) \\
&\quad \text{if } f_1 \;\&\; f_2 \text{ Lipschitz on } \widehat{\mathbf{x_0}}, \text{ else } \top \\
\partial^{Int}(1/f_1, \widehat{\mathbf{x_0}}) &= -\partial^{Int}(f_1, \widehat{\mathbf{x_0}}) /_{\mathbf{I}\mathbb{R}} f_1(\widehat{\mathbf{x_0}})^2 \\
&\quad \text{if } f_1 \text{ Lipschitz on } \widehat{\mathbf{x_0}} \text{ and } 0 \notin f_1(\widehat{\mathbf{x_0}}), \text{ else } \top \\
\partial^{Int}(C^1 \circ f_1, \widehat{\mathbf{x_0}}) &= \mathbf{J}^{Int}(C^1, f_1(\widehat{\mathbf{x_0}})) \cdot_{\mathbf{I}\mathbb{R}} \partial^{Int}(f_1, \widehat{\mathbf{x_0}}) \\
&\quad \text{if } f_1 \text{ Lipschitz on } \widehat{\mathbf{x_0}} \text{ and } C^1 \text{ differentiable on } f_1(\widehat{\mathbf{x_0}}), \text{ else } \top \\
\partial^{Int}(f_0 > c \;?\; f_1 : f_2, \widehat{\mathbf{x_0}}) &= \begin{cases} \partial^{Int}(f_1, [\![f_0 > c]\!](\widehat{\mathbf{x_0}})) & lb(f_0(\widehat{\mathbf{x_0}})) > c \\ \partial^{Int}(f_2, [\![f_0 < c]\!](\widehat{\mathbf{x_0}})) & ub(f_0(\widehat{\mathbf{x_0}})) < c \\ \partial^{Int}(f_1, [\![f_0 \geq c]\!](\widehat{\mathbf{x_0}})) \bigsqcup \partial^{Int}(f_2, [\![f_0 \leq c]\!](\widehat{\mathbf{x_0}})) & \text{otherwise} \end{cases} \\
&\quad \text{if } f_1 \;\&\; f_2 \text{ Lipschitz on } \widehat{\mathbf{x_0}} \text{ and} \\
&\quad f_1(x) = f_2(x) \text{ for } x \in \{x' : f_0(x') = c \wedge x' \in \widehat{\mathbf{x_0}}\}, \text{ else } \top \\
\partial^{Int}(x_i, \widehat{\mathbf{x_0}}) &= \widehat{\mathbf{e}}_i \\
\partial^{Int}(c, \widehat{\mathbf{x_0}}) &= \widehat{\mathbf{0}}
\end{aligned}$$

Figure 2.5: Interval Clarke Jacobian Rules

Constants and Variables

The Interval Clarke Jacobian of a constant function is nearly identical to the respective Clarke Jacobian (in that ∂^{Int} of a constant is 0); it is lifted to become the constant interval $[0, 0]$ of the same dimension as $\widehat{\mathbf{x_0}}$, which we denote as $\widehat{\mathbf{0}} \in \mathbf{I}\mathbb{R}^m$. Likewise, the Interval Clarke Jacobian of a single variable x_i is also the constant vector where the i^{th} component is 1 and all other $m-1$ components are 0, again lifted to become a constant interval, which we denote as $\widehat{\mathbf{e}}_i \in \mathbf{I}\mathbb{R}^m$.

Cartesian Product

The Interval Clarke Jacobian of a Cartesian product of functions is just the concatenation of their respective Interval Clarke Jacobians.

Addition

The Interval Clarke Jacobian obeys linearity exactly.

Multiplication and Division

The Interval Clarke Jacobian satisfies a lifted interval arithmetic version of the product and quotient rules.

Composition

For compositions with C^1-smooth functions, the Interval Clarke Jacobian satisfies a chain rule as well, where $\mathbf{J}^{Int}$ is the classical Jacobian of the C^1-smooth function, just interpreted using interval arithmetic instead of ordinary arithmetic.

Example 2.3. $\mathbf{J}^{Int}(\sin, [x_0^\ell, x_0^u]) = \cos([x_0^\ell, x_0^u])$ and in a similar fashion, $\mathbf{J}^{Int}(\tanh, [x_0^\ell, x_0^u]) = [1,1] -_{\mathbb{IR}} [\tanh(x_0^\ell), \tanh(x_0^u)]^2$.

Just as with ∂_c, primitive C^1 functions are where we check if the Jacobian is undefined, in which case ∂^{Int} evaluates to $\top$ which we propagate up the function's expression tree.

Branching

If the lower bound of $f_0(\widehat{\mathbf{x_0}})$ is larger than c, we definitively know that f_1 is the only possible function of the active set. Similarly, if the upper bound of $f_0(\widehat{\mathbf{x_0}})$ is smaller than c, we definitively know that f_2 is the only possible function of the active set. If $c \in f_0(\widehat{\mathbf{x_0}})$, then it is possible that both f_1 and f_2 are in the active set, thus we have to consider both possibilities.

In either case, we can also refine our information about $\widehat{\mathbf{x_0}}$. For example, when evaluating ∂^{Int} of the function $f(x) \triangleq x > 0 \ ? \ f_1 : f_2$ on the interval $\widehat{\mathbf{x_0}} = [-1, 1]$, conditional upon entering the true branch, we should only evaluate $\partial^{Int}(f_1, [0, 1])$. Likewise, conditional upon entering the false branch, we should only evaluate $\partial^{Int}(f_2, [-1, 0])$. We denote the refinement of $\widehat{\mathbf{x_0}}$ for a Boolean guard of the form $f_0 > c$ as $[\![f_0 > c]\!](\widehat{\mathbf{x_0}})$. However, the general problem of determining how to optimally refine the input $\widehat{\mathbf{x_0}}$, conditioned on the information of the branch $f_0 > c$, is undecidable. Therefore, we follow the approach of Miné [28] and refine $\widehat{\mathbf{x_0}}$ when the function f_0 in the Boolean guard has a simple form: a single variable x_i, and then use the fallback transformer (the identity function) for all other cases (which can then be

simplified to get the rule from Fig. 2.5):

$$[\![f_0 \ op \ c]\!](\widehat{\mathbf{x_0}}) = \begin{cases} \widehat{\mathbf{x_0}} \cap \big([\text{-}\infty, \infty] \times ... \times [\text{-}\infty, c] \times ... \times [\text{-}\infty, \infty]\big) & \text{if } f_0 = x_i \ \wedge \ op \in \{<, \leq\} \\ \widehat{\mathbf{x_0}} \cap \big([\text{-}\infty, \infty] \times ... \times [c, \infty] \times ... \times [\text{-}\infty, \infty]\big) & \text{if } f_0 = x_i \ \wedge \ op \in \{>, \geq\} \\ \widehat{\mathbf{x_0}} & \text{otherwise} \end{cases}$$

2.5.2 Soundness of the Abstraction

To state the soundness of ∂^{Int} we first define the concretization γ.

Definition 2.7. Define the concretization $\gamma : \mathbf{I}\mathbb{R}^{n\times m} \to \mathbf{Co}\big(\mathbb{R}^{n\times m}\big)$ for an interval matrix $S \in \mathbf{I}\mathbb{R}^{n\times m}$ as follows:

$$\gamma(S) = \{s \in \mathbb{R}^{n\times m} \mid s \in S\}$$

This definition results from the fact that an interval matrix is already by definition a convex set of matrices.

We are now ready to prove the soundness of ∂^{Int}. To prove this soundness result, we first state the following corollaries that follow from Definition 2.7.

Corollary 2.1. γ is monotonic with respect to $\subseteq$.

Corollary 2.2. For any $c \in \mathbb{R}$ and $S \in \mathbf{I}\mathbb{R}^{n\times m}$, $c \cdot \gamma(S) = \gamma(cS)$ and $\gamma(S^T) = \gamma(S)^T$.

Corollary 2.3. For any $S_1, S_2 \in \mathbf{I}\mathbb{R}^{n\times m}$, $\gamma(S_1 \times S_2) = \gamma(S_1) \times \gamma(S_2)$. Also, $\mathbf{co}(\gamma(S_1), \gamma(S_2)) \subseteq \gamma(S_1 \sqcup S_2)$, $\gamma(S_1) +_c \gamma(S_2) = \gamma(S_1 +_{\mathbf{I}\mathbb{R}} S_2)$ and $\gamma(S_1) \cdot_c \gamma(S_2) = \gamma(S_1 \cdot_{\mathbf{I}\mathbb{R}} S_2)$.

The proof of soundness is now given case-wise for each primitive in the language (where each case is given as a separate theorem). The final soundness theorem for the entire language is stated in Theorem 2.10.

Constants

Given that our language is inductively defined, we first prove the soundness of the interval domain over-approximation of the Clarke Jacobian for the base cases, the first one being for constant functions $f = c$ for any $c \in \mathbb{R}$.

Theorem 2.2. (Soundness of ∂^{Int} for Constants) Let $c \in \mathbb{R}$ and $\mathbf{x_0} \in \mathbb{R}^m$, $\widehat{\mathbf{x_0}} \in \mathbf{I}\mathbb{R}^m$ with $\mathbf{x_0} \in \widehat{\mathbf{x_0}}$.

$$\partial^{Int}(c, \widehat{\mathbf{x_0}}) = \widehat{\mathbf{0}} = [0, 0] \times ... \times [0, 0]$$

Further, by the rules for ∂_c, we know explicitly that

$$\partial_c(c, \mathbf{x_0}) = \mathbf{0} = \{(0, ..., 0)\}$$

Thus

$$\gamma(\partial^{Int}(c, \widehat{\mathbf{x_0}})) = \gamma([0,0] \times ... \times [0,0]) = \{(0, ..., 0)\} = \partial_c(c, \mathbf{x_0})$$

Hence, because of the equality, the weaker statement follows:

$$\partial_c(c, \mathbf{x_0}) \subseteq \gamma(\partial^{Int}(c, \widehat{\mathbf{x_0}}))$$

Variables

The next base case for which we must prove the soundness of the interval domain over-approximation of the Clarke Jacobian is for functions of a single variable $f = x_i$.

Theorem 2.3. (Soundness of ∂^{Int} for Variables) Let $f = x_i$ and $\mathbf{x_0} \in \mathbb{R}^m$, $\widehat{\mathbf{x_0}} \in \mathbf{I}\mathbb{R}^m$ with $\mathbf{x_0} \in \widehat{\mathbf{x_0}}$.

$$\partial^{Int}(x_i, \widehat{\mathbf{x_0}}) = \widehat{\mathbf{e}}_i = [0,0] \times ... \times [0,0] \times [1,1] \times [0,0]...$$

$$\partial_c(x_i, \mathbf{x_0}) = \mathbf{e}_i = \{(0, ...0, 1, 0...)\}$$

Hence

$$\gamma(\partial^{Int}(c, \widehat{\mathbf{x_0}})) = \gamma([0,0] \times ... \times [0,0] \times [1,1] \times [0,0]...) = \{(0, ...0, 1, 0...)\} = \partial_c(x_i, \mathbf{x_0})$$

Again, because of the equality, the weaker statement follows:

$$\partial_c(x_i, \mathbf{x_0}) \subseteq \gamma(\partial^{Int}(x_i, \widehat{\mathbf{x_0}}))$$

Addition

Having proven the soundness of the interval domain over-approximation of the Clarke Jacobian for the base cases of constant functions and functions of a single variable, we can now inductively prove the soundness of ∂^{Int} for the sum of functions.

Theorem 2.4. (Soundness of ∂^{Int} for Addition of Functions (Linearity)) Let $f_1, f_2 : \mathbb{R}^m \to \mathbb{R}$ and $\mathbf{x_0} \in \mathbb{R}^m$, $\widehat{\mathbf{x_0}} \in \mathbf{I}\mathbb{R}^m$ with $\mathbf{x_0} \in \widehat{\mathbf{x_0}}$. By the inductive assumption,

$$\partial_c(f_1, \mathbf{x_0}) \subseteq \gamma(\partial^{Int}(f_1, \widehat{\mathbf{x_0}}))$$

$$\partial_c(f_2, \mathbf{x_0}) \subseteq \gamma(\partial^{Int}(f_2, \widehat{\mathbf{x_0}}))$$

Thus

$$\partial_c(f_1, \mathbf{x_0}) +_c \partial_c(f_2, \mathbf{x_0}) \subseteq \gamma(\partial^{Int}(f_1, \widehat{\mathbf{x_0}})) +_c \gamma(\partial^{Int}(f_2, \widehat{\mathbf{x_0}}))$$

However, when the arguments are interval matrices, $+_c$ and $+_{\mathbf{I}\mathbb{R}}$ coincide exactly. Then, since $\gamma(\partial^{Int}(f_1, \widehat{\mathbf{x_0}}))$ and $\gamma(\partial^{Int}(f_2, \widehat{\mathbf{x_0}}))$ are interval matrices, we have

$$\partial_c(f_1, \mathbf{x_0}) +_c \partial_c(f_2, \mathbf{x_0}) \subseteq \gamma(\partial^{Int}(f_1, \widehat{\mathbf{x_0}})) +_{\mathbf{I}\mathbb{R}} \gamma(\partial^{Int}(f_2, \widehat{\mathbf{x_0}}))$$

Since γ factors over $+_{\mathbf{I}\mathbb{R}}$, we get:

$$\partial_c(f_1, \mathbf{x_0}) +_c \partial_c(f_2, \mathbf{x_0}) \subseteq \gamma(\partial^{Int}(f_1, \widehat{\mathbf{x_0}}) +_{\mathbf{I}\mathbb{R}} \partial^{Int}(f_2, \widehat{\mathbf{x_0}}))$$

By the definition of $\partial^{Int}(f_1 + f_2, \widehat{\mathbf{x_0}})$, we get:

$$\partial_c(f_1, \mathbf{x_0}) +_c \partial_c(f_2, \mathbf{x_0}) \subseteq \gamma(\partial^{Int}(f_1 + f_2, \widehat{\mathbf{x_0}}))$$

Lastly, since $\partial_c(f_1 + f_2, \mathbf{x_0}) \subseteq \partial_c(f_1, \mathbf{x_0}) +_c \partial_c(f_2, \mathbf{x_0})$, we get:

$$\partial_c(f_1 + f_2, \mathbf{x_0}) \subseteq \gamma(\partial^{Int}(f_1 + f_2, \widehat{\mathbf{x_0}}))$$

Multiplication

Proving the soundness of the interval domain over-approximation of the Clarke Jacobian for the multiplication of two functions (i.e., ensuring the correctness of the interval domain-lifted product rule) also requires the inductive assumption.

Theorem 2.5. (Soundness of ∂^{Int} for Multiplication of Functions (Lifted Product Rule)) Let $f_1, f_2 : \mathbb{R}^m \to \mathbb{R}$ and $\mathbf{x_0} \in \mathbb{R}^m$, $\widehat{\mathbf{x_0}} \in \mathbf{I}\mathbb{R}^m$ with $\mathbf{x_0} \in \widehat{\mathbf{x_0}}$. By the inductive assumption,

$$\partial_c(f_1, \mathbf{x_0}) \subseteq \gamma(\partial^{Int}(f_1, \widehat{\mathbf{x_0}}))$$

$$\partial_c(f_2, \mathbf{x_0}) \subseteq \gamma(\partial^{Int}(f_2, \widehat{\mathbf{x_0}}))$$

Furthermore, $f_1(\mathbf{x_0})$ and $f_2(\mathbf{x_0})$ are just vectors in $\mathbb{R}^n$, hence

$$f_2(\mathbf{x_0}) \cdot_c \partial_c(f_1, \mathbf{x_0}) \subseteq f_2(\mathbf{x_0}) \cdot_c \gamma(\partial^{Int}(f_1, \widehat{\mathbf{x_0}}))$$

$$f_1(\mathbf{x_0}) \cdot_c \partial_c(f_2, \mathbf{x_0}) \subseteq f_1(\mathbf{x_0}) \cdot_c \gamma(\partial^{Int}(f_2, \widehat{\mathbf{x_0}}))$$

Because γ distributes over scalar multiplication,

$$f_2(\mathbf{x_0}) \cdot_c \partial_c(f_1, \mathbf{x_0}) \subseteq \gamma(f_2(\mathbf{x_0}) \cdot_c \partial^{Int}(f_1, \widehat{\mathbf{x_0}}))$$

$$f_1(\mathbf{x_0}) \cdot_c \partial_c(f_2, \mathbf{x_0}) \subseteq \gamma(f_1(\mathbf{x_0}) \cdot_c \partial^{Int}(f_2, \widehat{\mathbf{x_0}}))$$

Thus we now have

$$f_2(\mathbf{x_0}) \cdot_c \partial_c(f_1, \mathbf{x_0}) +_c f_1(\mathbf{x_0}) \cdot_c \partial_c(f_2, \mathbf{x_0}) \subseteq \gamma(f_2(\mathbf{x_0}) \cdot_c \partial^{Int}(f_1, \widehat{\mathbf{x_0}})) +_c \gamma(f_1(\mathbf{x_0}) \cdot_c \partial^{Int}(f_2, \widehat{\mathbf{x_0}}))$$

Again, when the arguments are interval matrices, $+_c$ and $+_{\mathbb{IR}}$ coincide exactly, so because $\gamma(f_2(\mathbf{x_0}) \cdot_c \partial^{Int}(f_1, \widehat{\mathbf{x_0}}))$ and $\gamma(f_1(\mathbf{x_0}) \cdot_c \partial^{Int}(f_2, \widehat{\mathbf{x_0}}))$ are interval matrices, we have

$$f_2(\mathbf{x_0}) \cdot_c \partial_c(f_1, \mathbf{x_0}) +_c f_1(\mathbf{x_0}) \cdot_c \partial_c(f_2, \mathbf{x_0}) \subseteq \gamma(f_2(\mathbf{x_0}) \cdot_c \partial^{Int}(f_1, \widehat{\mathbf{x_0}})) +_{\mathbb{IR}} \gamma(f_1(\mathbf{x_0}) \cdot_c \partial^{Int}(f_2, \widehat{\mathbf{x_0}}))$$

Likewise, since $f_1(\mathbf{x_0})$ and $f_2(\mathbf{x_0})$ are vectors, $\cdot_c$ and $\cdot_{\mathbb{IR}}$ coincide giving

$$f_2(\mathbf{x_0}) \cdot_c \partial_c(f_1, \mathbf{x_0}) +_c f_1(\mathbf{x_0}) \cdot_c \partial_c(f_2, \mathbf{x_0}) \subseteq \gamma(f_2(\mathbf{x_0}) \cdot_{\mathbb{IR}} \partial^{Int}(f_1, \widehat{\mathbf{x_0}})) +_{\mathbb{IR}} \gamma(f_1(\mathbf{x_0}) \cdot_{\mathbb{IR}} \partial^{Int}(f_2, \widehat{\mathbf{x_0}}))$$

Furthermore, by the soundness of interval arithmetic,

$$f_1(\mathbf{x_0}) \in f_1(\widehat{\mathbf{x_0}})$$

$$f_2(\mathbf{x_0}) \in f_2(\widehat{\mathbf{x_0}})$$

Hence

$$f_2(\mathbf{x_0}) \cdot_{\mathbb{IR}} \partial^{Int}(f_1, \widehat{\mathbf{x_0}}) \subseteq f_2(\widehat{\mathbf{x_0}}) \cdot_{\mathbb{IR}} \partial^{Int}(f_1, \widehat{\mathbf{x_0}})$$

$$f_1(\mathbf{x_0}) \cdot_{\mathbb{IR}} \partial^{Int}(f_2, \widehat{\mathbf{x_0}}) \subseteq f_1(\widehat{\mathbf{x_0}}) \cdot_{\mathbb{IR}} \partial^{Int}(f_2, \widehat{\mathbf{x_0}})$$

By the monotonicity of γ,

$$\gamma(f_2(\mathbf{x_0}) \cdot_{\mathbb{IR}} \partial^{Int}(f_1, \widehat{\mathbf{x_0}})) \subseteq \gamma(f_2(\widehat{\mathbf{x_0}}) \cdot_{\mathbb{IR}} \partial^{Int}(f_1, \widehat{\mathbf{x_0}}))$$

$$\gamma(f_1(\mathbf{x_0}) \cdot_{\mathbb{IR}} \partial^{Int}(f_2, \widehat{\mathbf{x_0}})) \subseteq \gamma(f_1(\widehat{\mathbf{x_0}}) \cdot_{\mathbb{IR}} \partial^{Int}(f_2, \widehat{\mathbf{x_0}}))$$

Hence

$$\gamma(f_2(\mathbf{x_0}) \cdot_{\mathbb{IR}} \partial^{Int}(f_1, \widehat{\mathbf{x_0}})) +_{\mathbb{IR}} \gamma(f_1(\mathbf{x_0}) \cdot_{\mathbb{IR}} \partial^{Int}(f_2, \widehat{\mathbf{x_0}}))$$
$$\subseteq \gamma(f_2(\widehat{\mathbf{x_0}}) \cdot_{\mathbb{IR}} \partial^{Int}(f_1, \widehat{\mathbf{x_0}})) +_{\mathbb{IR}} \gamma(f_1(\widehat{\mathbf{x_0}}) \cdot_{\mathbb{IR}} \partial^{Int}(f_2, \widehat{\mathbf{x_0}}))$$

Therefore

$$f_2(\mathbf{x_0}) \cdot_c \partial_c(f_1, \mathbf{x_0}) +_c f_1(\mathbf{x_0}) \cdot_c \partial_c(f_2, \mathbf{x_0}) \subseteq \gamma(f_2(\widehat{\mathbf{x_0}}) \cdot_{\mathbf{I}\mathbb{R}} \partial^{Int}(f_1, \widehat{\mathbf{x_0}})) +_{\mathbf{I}\mathbb{R}} \gamma(f_1(\widehat{\mathbf{x_0}}) \cdot_{\mathbf{I}\mathbb{R}} \partial^{Int}(f_2, \widehat{\mathbf{x_0}}))$$

Since γ distributes over $+_{\mathbf{I}\mathbb{R}}$, we get

$$f_2(\mathbf{x_0}) \cdot_c \partial_c(f_1, \mathbf{x_0}) +_c f_1(\mathbf{x_0}) \cdot_c \partial_c(f_2, \mathbf{x_0}) \subseteq \gamma(f_2(\widehat{\mathbf{x_0}}) \cdot_{\mathbf{I}\mathbb{R}} \partial^{Int}(f_1, \widehat{\mathbf{x_0}}) +_{\mathbf{I}\mathbb{R}} f_1(\widehat{\mathbf{x_0}}) \cdot_{\mathbf{I}\mathbb{R}} \partial^{Int}(f_2, \widehat{\mathbf{x_0}}))$$

By the definition of $\partial^{Int}(f_1 \cdot f_2, \widehat{\mathbf{x_0}})$, we have

$$f_2(\mathbf{x_0}) \cdot_c \partial_c(f_1, \mathbf{x_0}) +_c f_1(\mathbf{x_0}) \cdot_c \partial_c(f_2, \mathbf{x_0}) \subseteq \gamma(\partial^{Int}(f_1 \cdot f_2, \widehat{\mathbf{x_0}}))$$

Lastly, by the definition of $\partial_c(f_1 \cdot f_2, \widehat{\mathbf{x_0}})$, we get

$$\partial_c(f_1 \cdot f_2, \widehat{\mathbf{x_0}}) \subseteq \gamma(\partial^{Int}(f_1 \cdot f_2, \widehat{\mathbf{x_0}}))$$

Division

We now detail the soundness of the interval domain over-approximation of the Clarke Jacobian for the division of two functions, thus ensuring the correctness of the lifted quotient rule. This also requires the inductive assumption.

Theorem 2.6. (Soundness of ∂^{Int} for Division of Functions (Lifted Quotient Rule)) Let $f_1 : \mathbb{R}^m \to \mathbb{R}$ and $\mathbf{x_0} \in \mathbb{R}^m$, $\widehat{\mathbf{x_0}} \in \mathbf{I}\mathbb{R}^m$ with $\mathbf{x_0} \in \widehat{\mathbf{x_0}}$. By the inductive assumption,

$$\partial_c(f_1, \mathbf{x_0}) \subseteq \gamma(\partial^{Int}(f_1, \widehat{\mathbf{x_0}}))$$

Since $-1/f_1(\mathbf{x_0})^2$ is a constant, $/_c$ simply scales each element of both sets by this constant, which still preserves the inequality

$$-\partial_c(f_1, \mathbf{x_0})/_c f_1(\mathbf{x_0})^2 \subseteq -\gamma(\partial^{Int}(f_1, \widehat{\mathbf{x_0}}))/_c f_1(\mathbf{x_0})^2$$

Since γ distributes over scaling,

$$-\partial_c(f_1, \mathbf{x_0})/_c f_1(\mathbf{x_0})^2 \subseteq \gamma(-\partial^{Int}(f_1, \widehat{\mathbf{x_0}})/_{\mathbf{I}\mathbb{R}} f_1(\mathbf{x_0}))^2$$

Then, by the soundness of interval arithmetic,

$$f_1(\mathbf{x_0})^2 \in f_1(\widehat{\mathbf{x_0}})^2$$

Hence

$$-\partial^{Int}(f_1, \widehat{\mathbf{x_0}})/_{\mathbf{I}\mathbb{R}} f_1(\mathbf{x_0})^2 \subseteq -\partial^{Int}(f_1, \widehat{\mathbf{x_0}})/_{\mathbf{I}\mathbb{R}} f_1(\widehat{\mathbf{x_0}})^2$$

Thus by the monotonicity of γ, we have

$$-\partial_c(f_1, \mathbf{x_0})/_c f_1(\mathbf{x_0})^2 \subseteq \gamma(-\partial^{Int}(f_1, \widehat{\mathbf{x_0}})/_{\mathbf{I}\mathbb{R}} f_1(\widehat{\mathbf{x_0}})^2)$$

Which by definition

$$\partial_c(1/f_1, \mathbf{x_0}) \subseteq \gamma(\partial^{Int}(1/f_1, \widehat{\mathbf{x_0}}))$$

Composition

We now detail the soundness of the interval domain over-approximation of the Clarke Jacobian for compositions with C^1 functions, thus ensuring the correctness of the lifted chain rule.

Theorem 2.7. (Soundness of ∂^{Int} for Compositions with C^1 functions (Lifted Chain Rule)) Let $f_1 : \mathbb{R}^m \to \mathbb{R}$ and $\mathbf{x_0} \in \mathbb{R}^m$, $\widehat{\mathbf{x_0}} \in \mathbf{I}\mathbb{R}^m$ with $\mathbf{x_0} \in \widehat{\mathbf{x_0}}$. By the inductive assumption,

$$\partial_c(f_1, \mathbf{x_0}) \subseteq \gamma\big(\partial^{Int}(f_1, \widehat{\mathbf{x_0}})\big)$$

Since each C^1 function maps $\mathbb{R} \to \mathbb{R}$, this means that $\mathbf{J}(C^1, f_1(\mathbf{x_0}))$ is a constant, hence we can scale both sides by this constant and preserve the $\subseteq$ inequality:

$$\mathbf{J}(C^1, f_1(\mathbf{x_0})) \cdot \partial_c(f_1, \mathbf{x_0}) \subseteq \mathbf{J}(C^1, f_1(\mathbf{x_0})) \cdot \gamma\big(\partial^{Int}(f_1, \widehat{\mathbf{x_0}})\big)$$

Since $\mathbf{J}(C^1, f_1(\mathbf{x_0}))$ is a constant, we can move it inside γ, giving

$$\mathbf{J}(C^1, f_1(\mathbf{x_0})) \cdot \partial_c(f_1, \mathbf{x_0}) \subseteq \gamma\big(\mathbf{J}(C^1, f_1(\mathbf{x_0})) \cdot \partial^{Int}(f_1, \widehat{\mathbf{x_0}})\big)$$

By the soundness of the interval arithmetic,

$$\mathbf{J}(C^1, f_1(\mathbf{x_0})) \in \mathbf{J}^{Int}(C^1, f_1(\widehat{\mathbf{x_0}}))$$

Hence

$$\mathbf{J}(C^1, f_1(\mathbf{x_0})) \cdot \partial^{Int}(f_1, \widehat{\mathbf{x_0}}) \subseteq \mathbf{J}^{Int}(C^1, f_1(\widehat{\mathbf{x_0}})) \cdot_{\mathbf{I}\mathbb{R}} \partial^{Int}(f_1, \widehat{\mathbf{x_0}})$$

Then, by monotonicity of γ, we have

$$\mathbf{J}(C^1, f_1(\mathbf{x_0}))\cdot\partial_c(f_1, \mathbf{x_0}) \subseteq \gamma\big(\mathbf{J}(C^1, f_1(\mathbf{x_0}))\cdot\partial^{Int}(f_1, \widehat{\mathbf{x_0}})\big) \subseteq \gamma\big(\mathbf{J}^{Int}(C^1, f_1(\widehat{\mathbf{x_0}}))\cdot_{\mathbf{I}\mathbb{R}}\partial^{Int}(f_1, \widehat{\mathbf{x_0}})\big)$$

By definition of $\partial_c(C^1 \circ f_1, \mathbf{x_0})$, we have

$$\partial_c(C^1 \circ f_1, \mathbf{x_0}) \subseteq \mathbf{J}(C^1, f_1(\mathbf{x_0})) \cdot \partial_c(f_1, \mathbf{x_0}) \subseteq \gamma\big(\mathbf{J}^{Int}(C^1, f_1(\widehat{\mathbf{x_0}})) \cdot_{\mathbf{I}\mathbb{R}} \partial^{Int}(f_1, \widehat{\mathbf{x_0}})\big)$$

Lastly, by the definition of $\partial^{Int}(C^1 \circ f_1, \widehat{\mathbf{x_0}})$, we obtain

$$\partial_c(C^1 \circ f_1, \mathbf{x_0}) \subseteq \gamma\big(\mathbf{J}^{Int}(C^1, f_1(\widehat{\mathbf{x_0}})) \cdot_{\mathbf{I}\mathbb{R}} \partial^{Int}(f_1, \widehat{\mathbf{x_0}})\big) = \gamma\big(\partial^{Int}(C^1 \circ f_1, \widehat{\mathbf{x_0}})\big)$$

Cartesian Product

We now describe the soundness of the interval domain over-approximation of the Clarke Jacobian for the Cartesian product of two functions, which also requires the inductive assumption.

Theorem 2.8. (Soundness of ∂^{Int} for Cartesian Products) Let $f_1, f_2 : \mathbb{R}^m \to \mathbb{R}^n$ and $\mathbf{x_0} \in \mathbb{R}^m$, $\widehat{\mathbf{x_0}} \in \mathbf{I}\mathbb{R}^m$ with $\mathbf{x_0} \in \widehat{\mathbf{x_0}}$. By the inductive assumption,

$$\partial_c(f_1, \mathbf{x_0}) \subseteq \gamma(\partial^{Int}(f_1, \widehat{\mathbf{x_0}}))$$

$$\partial_c(f_2, \mathbf{x_0}) \subseteq \gamma(\partial^{Int}(f_2, \widehat{\mathbf{x_0}}))$$

Starting from the definition of $\partial_c(f_1 \times f_2, \mathbf{x_0})$ gives:

$$\partial_c(f_1 \times f_2, \mathbf{x_0}) \subseteq \begin{bmatrix}\partial_c(f_1, \mathbf{x_0}) \\ \partial_c(f_2, \mathbf{x_0})\end{bmatrix}$$

By the inductive assumption,

$$\begin{bmatrix}\partial_c(f_1, \mathbf{x_0}) \\ \partial_c(f_2, \mathbf{x_0})\end{bmatrix} \subseteq \begin{bmatrix}\gamma(\partial^{Int}(f_1, \widehat{\mathbf{x_0}})) \\ \gamma(\partial^{Int}(f_2, \widehat{\mathbf{x_0}}))\end{bmatrix}$$

However, γ commutes over blocks in a block matrix, hence

$$\begin{bmatrix}\gamma(\partial^{Int}(f_1, \widehat{\mathbf{x_0}})) \\ \gamma(\partial^{Int}(f_2, \widehat{\mathbf{x_0}}))\end{bmatrix} = \gamma\left(\begin{bmatrix}\partial^{Int}(f_1, \widehat{\mathbf{x_0}}) \\ \partial^{Int}(f_2, \widehat{\mathbf{x_0}})\end{bmatrix}\right) = \gamma(\partial^{Int}(f_1 \times f_2, \widehat{\mathbf{x_0}}))$$

Branching

Finally, we describe the soundness of the interval domain over-approximation of the Clarke Jacobian for the branching primitive.

Theorem 2.9. (Soundness of ∂^{Int} for Branching) Let $f_1, f_2 : \mathbb{R}^m \to \mathbb{R}^n$ and $\mathbf{x_0} \in \mathbb{R}^m$, $\widehat{\mathbf{x_0}} \in \mathbf{I}\mathbb{R}^m$ with $\mathbf{x_0} \in \widehat{\mathbf{x_0}}$.

Case 2.1. If $f_0(\mathbf{x_0}) > c$, then

$$\partial_c(f_0 > c?\ f_1 : f_2, \mathbf{x_0}) = \partial_c(f_1, \mathbf{x_0})$$

and either $lb(f_0(\widehat{\mathbf{x_0}})) > c$ or $c \in f_0(\widehat{\mathbf{x_0}})$. If $lb(f_0(\widehat{\mathbf{x_0}})) > c$, then

$$\partial^{Int}(f_0 > c?\ f_1 : f_2, \widehat{\mathbf{x_0}}) = \partial^{Int}(f_1, [\![f_0 > c]\!](\widehat{\mathbf{x_0}}))$$

Thus all that remains to be shown is that $\mathbf{x_0} \in [\![f_0 > c]\!](\widehat{\mathbf{x_0}})$; we can invoke the inductive assumption.

If $f_0 \neq x_i$, then $[\![f_0 > c]\!](\widehat{\mathbf{x_0}}) = \widehat{\mathbf{x_0}}$, and by assumption $\mathbf{x_0} \in \widehat{\mathbf{x_0}}$, thus trivially,

$$\mathbf{x_0} \in [\![f_0 > c]\!](\widehat{\mathbf{x_0}})$$

If $f_0 = x_i$, then $f_0(\mathbf{x_0}) > c \implies \mathbf{x_0}[i] > c$, thus

$$\mathbf{x_0} \in [-\infty, \infty] \times ... \times [c, \infty] \times ... \times [-\infty, \infty]$$

But by assumption $\mathbf{x_0} \in \widehat{\mathbf{x_0}}$, thus by the definition of $[\![x_i > c]\!]$,

$$\mathbf{x_0} \in \widehat{\mathbf{x_0}} \cap [-\infty, \infty] \times ... \times [c, \infty] \times ... \times [-\infty, \infty]$$

$$\implies \mathbf{x_0} \in [\![f_0 > c]\!](\widehat{\mathbf{x_0}})$$

Thus by the inductive assumption,

$$\partial_c(f_1, \mathbf{x_0}) \subseteq \gamma(\partial^{Int}(f_1, [\![f_0 > c]\!](\widehat{\mathbf{x_0}})))$$

Hence, given the assumptions that $f_0(\mathbf{x_0}) > c \wedge lb(f_0(\widehat{\mathbf{x_0}})) > c$,

$$\partial_c(f_0 > c?\ f_1 : f_2, \mathbf{x_0}) = \partial_c(f_1, \mathbf{x_0}) \subseteq$$

$$\gamma(\partial^{Int}(f_1, [\![f_0 > c]\!](\widehat{\mathbf{x_0}}))) = \gamma(\partial^{Int}(f_0 > c?\ f_1 : f_2, \widehat{\mathbf{x_0}}))$$

$$f_0(\mathbf{x_0}) > c \wedge lb(f_0(\mathbf{x_0})) > c \implies \partial_c(f_0 > c?\ f_1 : f_2, \mathbf{x_0}) \subseteq \gamma(\partial^{Int}(f_0 > c?\ f_1 : f_2, \widehat{\mathbf{x_0}}))$$

Now, assuming $c \in f_0(\widehat{\mathbf{x_0}})$, then

$$\partial^{Int}(f_0 > c?\ f_1 : f_2, \widehat{\mathbf{x_0}}) = \partial^{Int}(f_1, [\![f_0 \geq c]\!](\widehat{\mathbf{x_0}})) \sqcup \partial^{Int}(f_2, [\![f_0 \leq c]\!](\widehat{\mathbf{x_0}}))$$

However, by the same logic, regardless of if $f_0 = x_i$ or $f_0 \neq x_i$, since $f_0(\mathbf{x_0}) > c$,

$$\implies \mathbf{x_0} \in [\![f_0 \geq c]\!](\widehat{\mathbf{x_0}})$$

Again invoking the inductive assumption,

$$\implies \partial_c(f_1, \mathbf{x_0}) \subseteq \gamma(\partial^{Int}(f_1, [\![f_0 \geq c]\!](\widehat{\mathbf{x_0}})))$$

However, by the monotonocity of γ,

$$\gamma(\partial^{Int}(f_1, [\![f_0 > c]\!](\widehat{\mathbf{x_0}}))) \subseteq \gamma(\partial^{Int}(f_1, [\![f_0 \geq c]\!](\widehat{\mathbf{x_0}})) \sqcup \partial^{Int}(f_2, [\![f_0 \leq c]\!](\widehat{\mathbf{x_0}})))$$

which means

$$\partial_c(f_1, \mathbf{x_0}) \subseteq \gamma(\partial^{Int}(f_1, [\![f_0 > c]\!](\widehat{\mathbf{x_0}}))) \subseteq \gamma(\partial^{Int}(f_1, [\![f_0 \geq c]\!](\widehat{\mathbf{x_0}})) \sqcup \partial^{Int}(f_2, [\![f_0 \leq c]\!](\widehat{\mathbf{x_0}}))$$

Thus substituting in, gives:

$$\partial_c(f_0 > c?\ f_1 : f_2, \mathbf{x_0}) \subseteq \gamma(\partial^{Int}(f_0 > c?\ f_1 : f_2, \widehat{\mathbf{x_0}}))$$

Case 2.2. If $f_0(\mathbf{x_0}) \geq c$, the proof proceeds exactly the same albeit replacing $>$ with $\geq$.

Case 2.3. If $f_0(\mathbf{x_0}) < c$, then $-f_0(\mathbf{x_0}) \geq -c$, which would be covered by the previous case.

Case 2.4. Likewise, if $f_0(\mathbf{x_0}) \leq c$, then $-f_0(\mathbf{x_0}) > -c$, which is covered by the first case.

Case 2.5. If $f_0(\mathbf{x_0}) = c$, then

$$\partial_c(f_0 > c?\ f_1 : f_2, \mathbf{x_0}) = \mathbf{co}(\partial_c(f_1, \mathbf{x_0}), \partial_c(f_2, \mathbf{x_0}))$$

which also implies $c \in f_0(\widehat{\mathbf{x_0}})$. Thus

$$\partial^{Int}(f_0 > c?\ f_1 : f_2, \widehat{\mathbf{x_0}}) = \partial^{Int}(f_1, [\![f_0 \geq c]\!](\widehat{\mathbf{x_0}})) \sqcup \partial^{Int}(f_2, [\![f_0 \leq c]\!](\widehat{\mathbf{x_0}}))$$

If $f_0 \neq x_i$, then

$$\partial^{Int}(f_1, [\![f_0 \geq c]\!](\widehat{\mathbf{x_0}})) \sqcup \partial^{Int}(f_2, [\![f_0 \leq c]\!](\widehat{\mathbf{x_0}})) = \partial^{Int}(f_1, \widehat{\mathbf{x_0}}) \sqcup \partial^{Int}(f_2, \widehat{\mathbf{x_0}})$$

But by the inductive assumption ,

$$\partial_c(f_1, \mathbf{x_0}) \subseteq \gamma(\partial^{Int}(f_1, \widehat{\mathbf{x_0}}))$$

$$\partial_c(f_2, \mathbf{x_0}) \subseteq \gamma(\partial^{Int}(f_2, \widehat{\mathbf{x_0}}))$$

If $f_0 = x_i$, then $\mathbf{x_0}[i] = c$ and

$$[\![f_0 \leq c]\!](\widehat{\mathbf{x_0}}) = \widehat{\mathbf{x_0}} \cap [-\infty, \infty] \times ... \times [-\infty, c] \times ... \times [-\infty, \infty]$$

$$[\![f_0 > c]\!](\widehat{\mathbf{x_0}}) = \widehat{\mathbf{x_0}} \cap [-\infty, \infty] \times ... \times [c, \infty] \times ... \times [-\infty, \infty]$$

However, because both interval vectors contain c in their i^{th} component interval,

$$\mathbf{x_0} \in [\![f_0 \leq c]\!](\widehat{\mathbf{x_0}})$$

$$\mathbf{x_0} \in [\![f_0 \geq c]\!](\widehat{\mathbf{x_0}})$$

Thus we can again invoke the inductive assumption:

$$\partial_c(f_1, \mathbf{x_0}) \subseteq \gamma(\partial^{Int}(f_1, [\![f_0 \geq c]\!](\widehat{\mathbf{x_0}}))$$

$$\partial_c(f_2, \mathbf{x_0}) \subseteq \gamma(\partial^{Int}(f_2, [\![f_0 \leq c]\!](\widehat{\mathbf{x_0}}))$$

Because **co** is monotonic,

$$\mathbf{co}(\partial_c(f_1, \mathbf{x_0}), \partial_c(f_2, \mathbf{x_0})) \subseteq \mathbf{co}(\gamma(\partial^{Int}(f_1, [\![f_0 \geq c]\!](\widehat{\mathbf{x_0}})), \gamma(\partial^{Int}(f_2, [\![f_0 \leq c]\!](\widehat{\mathbf{x_0}})))$$

By Corollary 2.3

$$\mathbf{co}(\gamma(\partial^{Int}(f_1, [\![f_0 \geq c]\!](\widehat{\mathbf{x_0}})), \gamma(\partial^{Int}(f_2, [\![f_0 \leq c]\!](\widehat{\mathbf{x_0}}))$$

$$\subseteq \gamma(\partial^{Int}(f_1, [\![f_0 \geq c]\!](\widehat{\mathbf{x_0}})) \sqcup \partial^{Int}(f_2, [\![f_0 \leq c]\!](\widehat{\mathbf{x_0}})))$$

Hence by transitivity:

$$\mathbf{co}(\partial_c(f_1, \mathbf{x_0}), \partial_c(f_2, \mathbf{x_0})) \subseteq \gamma(\partial^{Int}(f_1, [\![f_0 \geq c]\!](\widehat{\mathbf{x_0}})) \sqcup \partial^{Int}(f_2, [\![f_0 \leq c]\!](\widehat{\mathbf{x_0}})))$$

Substituting in the definitions:

$$\partial_c(f_0 > c?\ f_1 : f_2, \mathbf{x_0}) \subseteq \gamma(\partial^{Int}(f_0 > c?\ f_1 : f_2, \widehat{\mathbf{x_0}}))$$

Thus in all cases (when $f_0(\mathbf{x_0})$ is greater than, less than, or equal to c) and sub-cases (when f_0 is either the trivial function x_i or not), the following result holds:

$$\partial_c(f_0 > c?\ f_1 : f_2, \mathbf{x_0}) \subseteq \gamma(\partial^{Int}(f_0 > c?\ f_1 : f_2, \widehat{\mathbf{x_0}}))$$

Final Soundness Result. Having now proven Theorems 2.2 through 2.9, we can conclude by stating the full soundness result for *any* constructible function in our syntax:

Theorem 2.10. (Soundness) Let $f : \mathbb{R}^m \to \mathbb{R}^n$ be any function constructible according to Fig. 2.3 and $\mathbf{x_0} \in \mathbb{R}^m$, $\widehat{\mathbf{x_0}} \in \mathbf{I}\mathbb{R}^m$ with $\mathbf{x_0} \in \widehat{\mathbf{x_0}}$. Then

$$\partial_c(f, \mathbf{x_0}) \subseteq \gamma(\partial^{Int}(f, \widehat{\mathbf{x_0}}))$$

Proof. By structural induction on the language expressions in Theorems 2.2 through 2.9, we have covered all possible cases. QED.

2.5.3 Obtaining a Lipschitz Constant

Having proven that ∂^{Int} soundly over-approximates ∂_c, we next show how to use ∂^{Int} to bound the Lipschitz constant, which is a task we will experimentally evaluate in Section 2.8. As ∂^{Int} evaluates to an interval matrix, we first define the norm of interval matrices.

Definition 2.8. (Operator Norm of Interval Matrices) Let $\|\cdot\|_{\alpha,\beta}$ be an induced operator norm for $\mathbb{R}^{n\times m}$. For an interval matrix $M \in \mathbf{I}\mathbb{R}^{n\times m}$, we define $\widehat{\|M\|}_{\alpha,\beta}$ as

$$\widehat{\|M\|}_{\alpha,\beta} = sup_{m\in M}\|m\|_{\alpha,\beta}$$

As with the standard Jacobian, one can recover the Lipschitz constant from the Clarke Generalized Jacobian, thanks to a result by Jordan et al. [42].

Theorem 2.11. (Jordan et al. Theorem 1 [42]) Let $\|\cdot\|_\alpha$ and $\|\cdot\|_\beta$ be arbitary convex norms over $\mathbb{R}^m$ and $\mathbb{R}^n$, respectively, and let $f : \mathbb{R}^m \to \mathbb{R}^n$ be Lipschitz continuous over an open set $O \subseteq \mathbb{R}^m$. The following equality holds:

$$K^{(\alpha,\beta)}(f, O) = sup_{G\in\partial_c(f,O)}\|G^T\|_{\alpha,\beta}$$

where $\partial_c(f, O) = \{G \in \partial_c(f, x) \mid x \in O\}$ and $\|M\|_{\alpha,\beta} = sup_{\|v\|_\alpha \leq 1} \|Mv\|_\beta$.

Thankfully, for certain α, β, we can compute this value exactly, as we detail below.

Theorem 2.12. (Definition 7.1 [69], presented using our notation) For an interval matrix $M \in \mathbf{I}\mathbb{R}^{n \times m}$,

$$\widehat{\|M\|}_{1,1} = \max_{1 \leq j \leq m} \left(\sum_{i=1}^{n} \max \left(|lb(M_{i,j})|, |ub(M_{i,j})| \right) \right)$$

$$\widehat{\|M\|}_{\infty,\infty} = \max_{1 \leq i \leq n} \left(\sum_{j=1}^{m} \max \left(|lb(M_{i,j})|, |ub(M_{i,j})| \right) \right)$$

We can now relate this to the Lipschitz constant by combining it with Theorem 2.11.

Theorem 2.13. For a function $f : \mathbb{R}^m \to \mathbb{R}^n$ and an open set $\mathcal{X} \subseteq \mathbb{R}^m$ such that $\mathcal{X} \subseteq X$ where $X \in \mathbf{I}\mathbb{R}^m$, the local Lipschitz constant on $\mathcal{X}$ satisfies

$$K^{\alpha,\beta} \leq \|\widehat{\partial^{Int}(f, X)}\|_{\alpha,\beta}$$

Proof. By Theorem 2.11, we know that

$$K^{\alpha,\beta} = sup_{G \in \partial_c(f,\mathcal{X})} \|G\|_{\alpha,\beta}$$

But by the soundness of the over-approximation (Thm. 2.10), we know that $\partial_c(f, \mathcal{X}) \subseteq \gamma(\partial^{Int}(f, X))$. But $\gamma(\partial^{Int}(f, X)) = \partial^{Int}(f, X)$, hence

$$sup_{G \in \partial_c(f,\mathcal{X})} \|G\|_{\alpha,\beta} \leq sup_{G \in \partial^{Int}(f,X)} \|G\|_{\alpha,\beta}$$

And the RHS is just the definition of $\|\widehat{\partial^{Int}(f, X)}\|_{\alpha,\beta}$. QED.

2.6 DUAL INTERVAL DOMAIN

While we could compute ∂^{Int} recursively, a key contribution of our work is to provide an equivalent, forward-mode version of the static analysis based on dual numbers.

2.6.1 Dual Interval Arithmetic

Definition 2.9. The set of dual intervals, denoted as $\mathbf{I}\mathbb{D}$, are tuples of the form $[a, b] + [c, d]\epsilon$, where $a, b, c, d \in \mathbb{R} \cup \{\pm\infty\}$, $a \leq b$ and $c \leq d$. Intuitively, a dual interval represents a *set*

of dual numbers where the real part is within $[a, b]$ and the dual part is within $[c, d]$. To access the real part $[a, b]$ of a dual interval, we will write $fst([a, b] + [c, d]\epsilon)$; to access the coefficients of the dual part $[c, d]$, we will write $snd([a, b] + [c, d]\epsilon)$. We will denote the set of m-dimensional vectors of dual intervals as $\mathbb{ID}^m$ and the set of $n \times m$ dimensional matrices as $\mathbb{ID}^{n\times m}$. Lastly, we denote $\top = \mathbb{ID}^{n\times m}$ (the entire space of dual interval matrices).

We can lift the ordinary arithmetic operators to dual intervals. We define dual interval addition, $+_{\mathbb{ID}} : \mathbb{ID} \times \mathbb{ID} \to \mathbb{ID}$ as follows:

$$\big([a, b] + [c, d]\epsilon\big) +_{\mathbb{ID}} \big([e, f] + [g, h]\epsilon\big) = \big([a, b] +_{\mathbb{IR}} [e, f]\big) + \big([c, d] +_{\mathbb{IR}} [g, h]\big)\epsilon \tag{2.16}$$

We define dual interval multiplication, $\cdot_{\mathbb{ID}} : \mathbb{ID} \times \mathbb{ID} \to \mathbb{ID}$ as:

$$\big([a, b] + [c, d]\epsilon\big) \cdot_{\mathbb{ID}} \big([e, f] + [g, h]\epsilon\big) = \big([a, b] \cdot_{\mathbb{IR}} [e, f]\big) + \big([a, b] \cdot_{\mathbb{IR}} [g, h] +_{\mathbb{IR}} [c, d] \cdot_{\mathbb{IR}} [e, f]\big)\epsilon \tag{2.17}$$

And dual interval division, $/_{\mathbb{ID}} : \mathbb{ID} \times \mathbb{ID} \to \mathbb{ID}$ as:

$$\big([a, b] + [c, d]\epsilon\big) /_{\mathbb{ID}} \big([e, f] + [g, h]\epsilon\big) = \big([a, b] /_{\mathbb{IR}} [e, f]\big) + \Big(\big([c, d] \cdot_{\mathbb{IR}} [e, f] -_{\mathbb{IR}} [a, b] \cdot_{\mathbb{IR}} [g, h]\big) /_{\mathbb{IR}} [e, f]^2\Big)\epsilon \tag{2.18}$$

It will also be useful to define a join for dual intervals, $\sqcup_{\mathbb{ID}} : \mathbb{ID} \times \mathbb{ID} \to \mathbb{ID}$ as:

$$\big([a, b] + [c, d]\epsilon\big) \sqcup_{\mathbb{ID}} \big([e, f] + [g, h]\epsilon\big) = \big([a, b] \sqcup [e, f]\big) + \big([c, d] \sqcup [g, h]\big)\epsilon \tag{2.19}$$

2.6.2 Forward-Mode Abstract Evaluation with Dual Intervals

We now describe how to perform a forward-mode abstract evaluation where all operations are lifted to operate on dual intervals using the abstract interpreter $Eval_{\mathbb{ID}}$.

Definition 2.10. The abstract interpreter $Eval_{\mathbb{ID}} : \big(\big(\mathbb{R}^m \to \mathbb{R}^n\big) \times \mathbb{ID}^m\big) \to \mathbb{ID}^n$ takes a real-valued function $f : \mathbb{R}^m \to \mathbb{R}^n$ and evaluates it abstractly by lifting the interpretation of all operations to dual interval arithmetic, as shown in Fig. 2.6.

Because our language is functional, there is no "state." However, the second argument to $Eval_{\mathbb{ID}}$ is the input, which serves the same purpose. We now detail the rules of Fig. 2.6.

$$Eval_{\mathbf{ID}} : \left(\left(\mathbb{R}^m \to \mathbb{R}^n\right) \times \mathbf{ID}^m\right) \to \mathbf{ID}^n$$

$$\begin{aligned}
Eval_{\mathbf{ID}}(f_1 \times f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{y_0}}\epsilon) &= Eval_{\mathbf{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{y_0}}\epsilon) \times Eval_{\mathbf{ID}}(f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{y_0}}\epsilon) \\
&\quad \text{if } f_1 \text{ \& } f_2 \text{ Lipschitz on } \widehat{\mathbf{x_0}}, \text{ else } \top \\
Eval_{\mathbf{ID}}(f_1 + f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{y_0}}\epsilon) &= Eval_{\mathbf{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{y_0}}\epsilon) +_{\mathbf{ID}} Eval_{\mathbf{ID}}(f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{y_0}}\epsilon) \\
&\quad \text{if } f_1 \text{ \& } f_2 \text{ Lipschitz on } \widehat{\mathbf{x_0}}, \text{ else } \top \\
Eval_{\mathbf{ID}}(f_1 \cdot f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{y_0}}\epsilon) &= Eval_{\mathbf{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{y_0}}\epsilon) \cdot_{\mathbf{ID}} Eval_{\mathbf{ID}}(f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{y_0}}\epsilon) \\
&\quad \text{if } f_1 \text{ \& } f_2 \text{ Lipschitz on } \widehat{\mathbf{x_0}}, \text{ else } \top \\
Eval_{\mathbf{ID}}(1/f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{y_0}}\epsilon) &= \left([1,1] + [0,0]\epsilon\right) /_{\mathbf{ID}} Eval_{\mathbf{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{y_0}}\epsilon) \\
&\quad \text{if } f_1 \text{ Lipschitz on } \widehat{\mathbf{x_0}}, \text{ else } \top \\
Eval_{\mathbf{ID}}(C^1 \circ f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{y_0}}\epsilon) &= Eval_{\mathbf{ID}}(C^1, Eval_{\mathbf{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{y_0}}\epsilon)) \\
&\quad \text{if } f_1 \text{ Lipschitz on } \widehat{\mathbf{x_0}} \text{ and } C^1 \text{ differentiable on } f_1(\widehat{\mathbf{x_0}}), \text{ else } \top \\
Eval_{\mathbf{ID}}(f_0 > c \ ? \ f_1 : f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{y_0}}\epsilon) &= \begin{cases} Eval_{\mathbf{ID}}(f_1, [\![f_0 > c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{y_0}}\epsilon) & lb(fst(Eval_{\mathbf{ID}}(f_0, \widehat{\mathbf{x_0}} + \widehat{\mathbf{y_0}}\epsilon))) > c \\ Eval_{\mathbf{ID}}(f_2, [\![f_0 < c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{y_0}}\epsilon) & ub(fst(Eval_{\mathbf{ID}}(f_0, \widehat{\mathbf{x_0}} + \widehat{\mathbf{y_0}}\epsilon))) < c \\ Eval_{\mathbf{ID}}(f_1, [\![f_0 \geq c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{y_0}}\epsilon) & \text{otherwise} \\ \quad \sqcup_{\mathbf{ID}} Eval_{\mathbf{ID}}(f_2, [\![f_0 \leq c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{y_0}}\epsilon) & \end{cases} \\
Eval_{\mathbf{ID}}(x_i, \widehat{\mathbf{x_0}} + \widehat{\mathbf{y_0}}\epsilon) &= \widehat{\mathbf{x_0}}[i] + \widehat{\mathbf{y_0}}[i]\epsilon \\
Eval_{\mathbf{ID}}(c, \widehat{\mathbf{x_0}} + \widehat{\mathbf{y_0}}\epsilon) &= [c,c] + [0,0]\epsilon \\
Eval_{\mathbf{ID}}(C^1, [x_0^\ell, x_0^u] + [y_0^\ell, y_0^u]\epsilon) &= C^1([x_0^\ell, x_0^u]) + \left(\mathbf{J}^{Int}(C^1, [x_0^\ell, x_0^u]) \cdot_{\mathbb{IR}} [y_0^\ell, y_0^u]\right)\epsilon \\
&\quad \text{if } C^1 \text{ differentiable on } [x_0^\ell, x_0^u], \text{ else } \top
\end{aligned}$$

Figure 2.6: Dual Interval Forward Mode Abstract Evaluation

Constants and Variables

The abstract evaluation of a constant is that constant, albeit abstracted to a (degenerate) dual interval. Likewise, the evaluation of a single variable x_i is the i^{th} element of the input dual interval (denoted by the $[i]$ accessor).

Cartesian Product

To abstractly evaluate the Cartesian product of two functions f_1 and f_2 on a given input $\widehat{\mathbf{x_0}} + \widehat{\mathbf{y_0}}\epsilon$, we abstractly evaluate each one, then take the Cartesian product of the results.

Addition

To abstractly evaluate the sum of two functions, we abstractly evaluate each on the given input $\widehat{\mathbf{x_0}} + \widehat{\mathbf{y_0}}\epsilon$, then take the dual interval sum $+_{\mathbb{ID}}$ of the respective results.

Multiplication and Division

As with addition, to abstractly evaluate the product of two functions or their quotient, we first abstractly evaluate the individual functions on the input, then use the dual interval forms of multiplication $\cdot_{\mathbb{ID}}$ and division $/_{\mathbb{ID}}$.

Composition

The abstract evaluation of the composition of any function f_1 with a C^1 function for an input dual interval $\widehat{\mathbf{x_0}} + \widehat{\mathbf{y_0}}\epsilon$, is the composition of the successive abstract evaluations: f_1 is abstractly evaluated on the input, then the C^1 function is abstractly evaluated on that result.

Branching

When evaluating a branch abstractly, *we only use the real part* of the dual interval, denoted $fst(Eval_{\mathbf{ID}}(f_0, \widehat{\mathbf{x_0}} + \widehat{\mathbf{y_0}}\epsilon))$, to select which branch to take (or whether to abstractly evaluate both). While this may seem strange, one will note that in Eq. 2.15, the Clarke Jacobian is only computed for the piece that is ultimately chosen – f_1 or f_2 (possibly both) – and not the threshold function f_0. In fact, f_0 is only used to select which branch to take, hence its derivative information (which corresponds to its dual part) is unnecessary. If the lower bound $lb(fst(Eval_{\mathbf{ID}}(f_0, \widehat{\mathbf{x_0}} + \widehat{\mathbf{y_0}}\epsilon)))$ is larger than c, we definitively know to take only the true branch. Conversely, if the upper bound of the real part $ub(fst(Eval_{\mathbf{ID}}(f_0, \widehat{\mathbf{x_0}} + \widehat{\mathbf{y_0}}\epsilon)))$ is less than c, we only take the false branch. Otherwise, we abstractly evaluate both branches then take their join $\sqcup_{\mathbb{ID}}$. In all cases, we can refine the information of the real part $\widehat{\mathbf{x_0}}$ but not the dual part $\widehat{\mathbf{y_0}}$, as there is no way of knowing which sub-regions of $\widehat{\mathbf{x_0}}$ correspond to which sub-regions in $\widehat{\mathbf{y_0}}$, since the interval domain is non-relational.

2.6.3 Equivalence of $Eval_{\mathbf{ID}}$ and ∂^{Int}

We now give the proof that forward mode dual interval evaluation, $Eval_{\mathbf{ID}}$, yields the same answer as the recursively defined ∂^{Int}.

Theorem 2.14. (Equivalence of ∂^{Int} and $Eval_{\mathbf{I}\mathbb{D}}$) Let $f : \mathbb{R}^m \to \mathbb{R}^n$ and let $\widehat{\mathbf{x_0}} \in \mathbf{I}\mathbb{R}^m$. Then

$$\partial^{Int}(f, \widehat{\mathbf{x_0}}) = snd\Big(Eval_{\mathbf{I}\mathbb{D}}(f, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)^T \times ... \times Eval_{\mathbf{I}\mathbb{D}}(f, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)^T\Big)$$

Proof. We start with the base cases.

Case 2.6. Base Cases - Constants. The first base case is when $f = c$. We start with

$$\partial^{Int}(c, \widehat{\mathbf{x_0}}) = \widehat{\mathbf{0}}$$

But $Eval_{\mathbf{I}\mathbb{D}}(c, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_i\epsilon)^T = [c, c] + [0, 0]\epsilon$ for any $i \in \{1, .., m\}$, thus

$$snd\Big(Eval_{\mathbf{I}\mathbb{D}}(c, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)^T \times ... \times Eval_{\mathbf{I}\mathbb{D}}(c, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)^T\Big) = [0, 0] \times ... \times [0, 0]$$

But by our notation, $\widehat{\mathbf{0}} \triangleq [0, 0] \times ... \times [0, 0]$. Hence

$$\partial^{Int}(c, \widehat{\mathbf{x_0}}) = \widehat{\mathbf{0}} = snd\Big(Eval_{\mathbf{I}\mathbb{D}}(c, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)^T \times ... \times Eval_{\mathbf{I}\mathbb{D}}(c, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)^T\Big)$$

Case 2.7. Base Cases - Variables. We next analyze the base case where $f = x_i$. We start with

$$\partial^{Int}(x_i, \widehat{\mathbf{x_0}}) = \widehat{\mathbf{e}}_i$$

where $\widehat{\mathbf{e}}_i = [0, 0], \times ... \times [0, 0] \times [1, 1] \times [0, 0]...$ (the $[1, 1]$ term is in the i^{th} index). We also know that

$$Eval_{\mathbf{I}\mathbb{D}}(x_i, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_i\epsilon) = \widehat{\mathbf{x_0}}[i] + \widehat{\mathbf{e}}_i[i]\epsilon = \widehat{\mathbf{x_0}}[i] + [1, 1]\epsilon$$

Thus

$$snd\big(Eval_{\mathbf{I}\mathbb{D}}(x_i, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_i\epsilon)\big) = [1, 1] = snd\big(Eval_{\mathbf{I}\mathbb{D}}(x_i, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_i\epsilon)^T\big)$$

And for any $j \neq i$,

$$Eval_{\mathbf{I}\mathbb{D}}(x_i, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_j\epsilon) = \widehat{\mathbf{x_0}}[i] + \widehat{\mathbf{e}}_j[i]\epsilon = \widehat{\mathbf{x_0}}[i] + [0, 0]\epsilon$$

Thus

$$snd\big(Eval_{\mathbf{I}\mathbb{D}}(x_i, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_j\epsilon)\big) = [0, 0] = snd\big(Eval_{\mathbf{I}\mathbb{D}}(x_i, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_j\epsilon)^T\big)$$

Hence

$$snd\Big(Eval_{\mathbf{I}\mathbb{D}}(x_i, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)^T \times ... \times Eval_{\mathbf{I}\mathbb{D}}(x_i, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)^T\Big) = [0, 0] \times ...[0, 0] \times [1, 1] \times [0, 0]...$$

$$= \widehat{\mathbf{e}}_i = \partial^{Int}(x_i, \widehat{\mathbf{x_0}})$$

We now proceed to the inductive cases.

Case 2.8. Addition. We require both $f_1, f_2 : \mathbb{R}^m \to \mathbb{R}^n$. We start with

$$\partial^{Int}(f_1 + f_2, \widehat{\mathbf{x_0}}) = \partial^{Int}(f_1, \widehat{\mathbf{x_0}}) +_{\mathbf{IR}} \partial^{Int}(f_2, \widehat{\mathbf{x_0}})$$

By the inductive hypothesis,

$$= snd\Big(Eval_{\mathbf{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)^T \times ... \times Eval_{\mathbf{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)^T\Big)$$

$$+_{\mathbf{IR}} snd\Big(Eval_{\mathbf{ID}}(f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)^T \times ... \times Eval_{\mathbf{ID}}(f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)^T\Big)$$

Distributing the *snd* then grouping terms element-wise,

$$= \Big(snd\Big(Eval_{\mathbf{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)^T\Big) +_{\mathbf{IR}} snd\Big(Eval_{\mathbf{ID}}(f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)^T\Big)\Big) \times ...$$

$$... \times \Big(snd\Big(Eval_{\mathbf{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)^T\Big) +_{\mathbf{IR}} snd\Big(Eval_{\mathbf{ID}}(f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)^T\Big)\Big)$$

By the definition of dual interval arithmetic evaluation of addition,

$$= snd\Big(Eval_{\mathbf{ID}}(f_1 + f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)^T\Big) \times ...$$

$$... \times snd\Big(Eval_{\mathbf{ID}}(f_1 + f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)^T\Big)$$

Undistributing the *snd*,

$$= snd\Big(Eval_{\mathbf{ID}}(f_1 + f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)^T \times ... \times Eval_{\mathbf{ID}}(f_1 + f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)^T\Big)$$

Case 2.9. Multiplication. We require both $f_1, f_2 : \mathbb{R}^m \to \mathbb{R}^n$. We start with

$$\partial^{Int}(f_1 \cdot f_2, \widehat{\mathbf{x_0}}) = f_2(\widehat{\mathbf{x_0}}) \cdot_{\mathbf{IR}} \partial^{Int}(f_1, \widehat{\mathbf{x_0}}) +_{\mathbf{IR}} f_1(\widehat{\mathbf{x_0}}) \cdot_{\mathbf{IR}} \partial^{Int}(f_2, \widehat{\mathbf{x_0}})$$

By the inductive hypothesis,

$$= f_2(\widehat{\mathbf{x_0}}) \cdot_{\mathbf{IR}} snd\Big(Eval_{\mathbf{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)^T \times ... \times Eval_{\mathbf{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)^T\Big)$$

$$+_{\mathbf{IR}} f_1(\widehat{\mathbf{x_0}}) \cdot_{\mathbf{IR}} snd\Big(Eval_{\mathbf{ID}}(f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)^T \times ... \times Eval_{\mathbf{ID}}(f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)^T\Big)$$

Distributing the snd then distributing f_1 and f_2 by the definition of $\cdot_{\mathbb{IR}}$,

$$= \Big(f_2(\widehat{\mathbf{x_0}}) \cdot_{\mathbb{IR}} snd(Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)^T) \times ... \times \big(f_2(\widehat{\mathbf{x_0}}) \cdot_{\mathbb{IR}} snd(Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)^T)\Big)$$

$$+_{\mathbb{IR}}\Big(f_1(\widehat{\mathbf{x_0}}) \cdot_{\mathbb{IR}} snd\Big(Eval_{\mathbb{ID}}(f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)^T\Big) \times ... \times f_1(\widehat{\mathbf{x_0}}) \cdot_{\mathbb{IR}} snd\Big(Eval_{\mathbb{ID}}(f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)^T\Big)\Big)$$

Because $f_2(\widehat{\mathbf{x_0}}) = fst\Big(Eval_{\mathbb{ID}}(f_2, \widehat{\mathbf{x_0}} + \mathbf{e_i}\epsilon)\Big)$ for any $i \in \{1, ..., m\}$, and for $f_1(\widehat{\mathbf{x_0}})$, we have

$$= \Big(fst\Big(Eval_{\mathbb{ID}}(f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)\Big) \cdot_{\mathbb{IR}} snd\Big(Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)^T\Big) \times ...$$

$$... \times fst\Big(Eval_{\mathbb{ID}}(f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)\Big) \cdot_{\mathbb{IR}} snd\Big(Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)^T\Big)\Big)$$

$$+_{\mathbb{IR}}\Big(fst\Big(Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)\Big) \cdot_{\mathbb{IR}} snd\Big(Eval_{\mathbb{ID}}(f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)^T\Big) \times ...$$

$$... \times fst\Big(Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)\Big) \cdot_{\mathbb{IR}} snd\Big(Eval_{\mathbb{ID}}(f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)^T\Big)\Big)$$

By the definition of $+_{\mathbb{IR}}$, we can group terms element-wise:

$$= \Big(fst\Big(Eval_{\mathbb{ID}}(f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)\Big) \cdot_{\mathbb{IR}} snd\Big(Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)^T\Big)$$

$$+_{\mathbb{IR}} fst\Big(Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)\Big) \cdot_{\mathbb{IR}} snd\Big(Eval_{\mathbb{ID}}(f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)^T\Big)\Big) \times ...$$

$$... \times \Big(fst\Big(Eval_{\mathbb{ID}}(f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)\Big) \cdot_{\mathbb{IR}} snd\Big(Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)^T\Big)$$

$$+_{\mathbb{IR}} fst\Big(Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)\Big) \cdot_{\mathbb{IR}} snd\Big(Eval_{\mathbb{ID}}(f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)^T\Big)\Big)$$

By the definition of $\cdot_{\mathbb{IR}}$,

$$= snd\Big(Eval_{\mathbb{ID}}(f_1 \cdot f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)^T\Big) \times ... \times snd\Big(Eval_{\mathbb{ID}}(f_1 \cdot f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)^T\Big)$$

Undistributing the snd,

$$= snd\Big(Eval_{\mathbb{ID}}(f_1 \cdot f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)^T \times ... \times Eval_{\mathbb{ID}}(f_1 \cdot f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)^T\Big)$$

Case 2.10. Division. We require $f_1 : \mathbb{R}^m \to \mathbb{R}$. We start from the desired LHS:

$$snd\Big(Eval_{\mathbb{ID}}(1/f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)^T \times ... \times Eval_{\mathbb{ID}}(1/f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)^T\Big)$$

Distributing the snd,

$$= snd\Big(Eval_{\mathbb{ID}}(1/f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)^T\Big) \times ... \times snd\Big(Eval_{\mathbb{ID}}(1/f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)^T\Big)$$

Since $1/f$ has one dimensional output, $Eval_{\mathbb{ID}}(1/f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)^T$ is a (dual-interval) scalar. Hence, $Eval_{\mathbb{ID}}(1/f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_i\epsilon)^T = Eval_{\mathbb{ID}}(1/f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_i\epsilon)$, and thus we can drop all transposes, giving:

$$= snd\Big(Eval_{\mathbb{ID}}(1/f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)\Big) \times ... \times snd\Big(Eval_{\mathbb{ID}}(1/f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)\Big)$$

By the definition of $Eval_{\mathbb{ID}}$,

$$= snd\Big([1,1] + [0,0]\epsilon/_{\mathbb{ID}} Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)\Big) \times ... \times snd\Big([1,1] + [0,0]\epsilon/_{\mathbb{ID}} Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)\Big)$$

By the definition of dual interval division,

$$= snd\Big([1,1]/_{\mathbb{IR}} fst\Big(Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)\Big) - snd\Big(Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)\Big)/_{\mathbb{IR}} fst\Big(Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)\Big)^2\epsilon\Big) \times ...$$

$$... \times snd\Big([1,1]/_{\mathbb{IR}} fst\Big(Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)\Big) - snd\Big(Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)\Big)/_{\mathbb{IR}} fst\Big(Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)\Big)^2\epsilon\Big)$$

Applying the outermost snd operator element-wise,

$$= -snd\Big(Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)\Big)/_{\mathbb{IR}} fst\Big(Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)\Big)^2 \times ...$$

$$... \times -snd\Big(Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)\Big)/_{\mathbb{IR}} fst\Big(Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)\Big)^2$$

However, since $fst\Big(Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_i\epsilon)\Big) = f_1(\widehat{\mathbf{x_0}})$ for any i, we have:

$$= -snd\Big(Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)\Big)/_{\mathbb{IR}} f_1(\widehat{\mathbf{x_0}})^2 \times ...$$

$$... \times -snd\Big(Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)\Big)/_{\mathbb{IR}} f_1(\widehat{\mathbf{x_0}})^2$$

We can pull out the $f_1(\widehat{\mathbf{x_0}})^2$ term since it is a scalar common to each dimension,

$$= \Big(-snd\Big(Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)\Big) \times ... \times -snd\Big(Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)\Big)\Big)/_{\mathbb{IR}} f_1(\widehat{\mathbf{x_0}})^2$$

But by the inductive hypothesis, the numerator just reduces to $-\partial^{Int}(f_1, \widehat{\mathbf{x_0}})$, hence giving

$$= -\partial^{Int}(f_1, \widehat{\mathbf{x_0}})/_{\mathbb{IR}} f_1(\widehat{\mathbf{x_0}})^2$$

$$= \partial^{Int}(1/f_1, \widehat{\mathbf{x_0}})$$

Case 2.11. Branching.

If $lb(f_0(\widehat{\mathbf{x_0}})) > c$, then we have

$$\partial^{Int}(f_0 > c \ ? \ f_1 : f_2, \widehat{\mathbf{x_0}}) = \partial^{Int}(f_1, [\![f_0 > c]\!](\widehat{\mathbf{x_0}}))$$

However, $lb(f_0(\widehat{\mathbf{x_0}})) > c$ also means that for any $i \in \{1, ..., m\}$,

$$Eval_{\mathbf{ID}}(f_0 > c \ ? \ f_1 : f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_i\epsilon) = Eval_{\mathbf{ID}}(f_1, [\![f_0 > c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{e}}_i\epsilon)$$

Hence, by the inductive hypothesis:

$$\partial^{Int}(f_1, [\![f_0 > c]\!](\widehat{\mathbf{x_0}})) = snd\Big(Eval_{\mathbf{ID}}(f_1, [\![f_0 > c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{e}}_1\epsilon)^T \times ... \times Eval_{\mathbf{ID}}(f_1, [\![f_0 > c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{e}}_m\epsilon)^T\Big)$$

Thus substituting back in, we have that $lb(f_0(\widehat{\mathbf{x_0}})) > c$ implies

$$\partial^{Int}(f_0 > c \ ? \ f_1 : f_2, \widehat{\mathbf{x_0}}) =$$

$$snd\Big(Eval_{\mathbf{ID}}(f_0 > c \ ? \ f_1 : f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)^T \times ... \times Eval_{\mathbf{ID}}(f_0 > c \ ? \ f_1 : f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)^T\Big)$$

The case where $ub(f_0(\widehat{\mathbf{x_0}})) < c$ proceeds exactly the same by symmetry.

The case where $c \in f_0(\widehat{\mathbf{x_0}})$, then

$$\partial^{Int}(f_0 > c \ ? \ f_1 : f_2, \widehat{\mathbf{x_0}}) = \partial^{Int}(f_1, [\![f_0 \geq c]\!](\widehat{\mathbf{x_0}})) \sqcup \partial^{Int}(f_2, [\![f_0 \leq c]\!](\widehat{\mathbf{x_0}})$$

But if $c \in f_0(\widehat{\mathbf{x_0}})$, then for any $i \in \{1, ..., m\}$,

$$Eval_{\mathbf{ID}}(f_0 > c \ ? \ f_1 : f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_i\epsilon) = Eval_{\mathbf{ID}}(f_1, [\![f_0 \geq c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{e}}_i\epsilon) \sqcup_{\mathbf{ID}} Eval_{\mathbf{ID}}(f_2, [\![f_0 \leq c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{e}}_i\epsilon)$$

Furthermore, the transpose T distributes over $\sqcup_{\mathbf{ID}}$:

$$\Big(Eval_{\mathbf{ID}}(f_1, [\![f_0 \geq c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{e}}_i\epsilon) \sqcup_{\mathbf{ID}} Eval_{\mathbf{ID}}(f_2, [\![f_0 \leq c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{e}}_i\epsilon)\Big)^T$$

$$= Eval_{\mathbf{ID}}(f_1, [\![f_0 \geq c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{e}}_i\epsilon)^T \sqcup_{\mathbf{ID}} Eval_{\mathbf{ID}}(f_2, [\![f_0 \leq c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{e}}_i\epsilon)^T$$

Thus starting from the desired RHS:

$$snd\Big(Eval_{\mathbf{ID}}(f_0 > c\ ?\ f_1 : f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)^T \times ... \times Eval_{\mathbf{ID}}(f_0 > c\ ?\ f_1 : f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)^T\Big)$$

$$= snd\Big(Eval_{\mathbf{ID}}(f_1, [\![f_0 \geq c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{e}}_1\epsilon)^T \sqcup_{\mathbf{ID}} Eval_{\mathbf{ID}}(f_2, [\![f_0 \leq c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{e}}_1\epsilon)^T \times ...$$

$$... \times Eval_{\mathbf{ID}}(f_1, [\![f_0 \geq c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{e}}_m\epsilon)^T \sqcup_{\mathbf{ID}} Eval_{\mathbf{ID}}(f_2, [\![f_0 \leq c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{e}}_m\epsilon)^T\Big)$$

We distribute *snd* to each term:

$$= snd\Big(Eval_{\mathbf{ID}}(f_1, [\![f_0 \geq c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{e}}_1\epsilon)^T \sqcup_{\mathbf{ID}} Eval_{\mathbf{ID}}(f_2, [\![f_0 \leq c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{e}}_1\epsilon)^T\Big) \times ...$$

$$... \times snd\Big(Eval_{\mathbf{ID}}(f_1, [\![f_0 \geq c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{e}}_m\epsilon)^T \sqcup_{\mathbf{ID}} Eval_{\mathbf{ID}}(f_2, [\![f_0 \leq c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{e}}_m\epsilon)^T\Big)$$

And since $snd(A \sqcup_{\mathbf{ID}} B) = snd(A) \sqcup snd(B)$, we get (for any i):

$$snd\Big(Eval_{\mathbf{ID}}(f_1, [\![f_0 \geq c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{e}}_i\epsilon)^T \sqcup_{\mathbf{ID}} Eval_{\mathbf{ID}}(f_2, [\![f_0 \leq c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{e}}_i\epsilon)^T\Big)$$

$$= snd\Big(Eval_{\mathbf{ID}}(f_1, [\![f_0 \geq c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{e}}_i\epsilon)^T\Big) \sqcup snd\Big(Eval_{\mathbf{ID}}(f_2, [\![f_0 \leq c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{e}}_i\epsilon)^T\Big)$$

Thus applying this to each term gives:

$$= snd\Big(Eval_{\mathbf{ID}}(f_1, [\![f_0 \geq c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{e}}_1\epsilon)^T\Big) \sqcup snd\Big(Eval_{\mathbf{ID}}(f_2, [\![f_0 \leq c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{e}}_1\epsilon)^T\Big) \times ...$$

$$... \times snd\Big(Eval_{\mathbf{ID}}(f_1, [\![f_0 \geq c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{e}}_m\epsilon)^T\Big) \sqcup snd\Big(Eval_{\mathbf{ID}}(f_2, [\![f_0 \leq c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{e}}_m\epsilon)^T\Big)$$

But this is just $\sqcup$ applied element-wise, hence we can rewrite it as

$$= snd\Big(Eval_{\mathbf{ID}}(f_1, [\![f_0 \geq c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{e}}_1\epsilon)^T\Big) \times ... \times snd\Big(Eval_{\mathbf{ID}}(f_1, [\![f_0 \geq c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{e}}_m\epsilon)^T\Big)$$

$$\sqcup$$

$$snd\Big(Eval_{\mathbf{ID}}(f_2, [\![f_0 \leq c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{e}}_1\epsilon)^T\Big) \times ... \times snd\Big(Eval_{\mathbf{ID}}(f_2, [\![f_0 \leq c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{e}}_m\epsilon)^T\Big)$$

Undistributing the *snd* gives

$$= snd\Big(Eval_{\mathbf{ID}}(f_1, [\![f_0 \geq c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{e}}_1\epsilon)^T \times ... \times Eval_{\mathbf{ID}}(f_1, [\![f_0 \geq c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{e}}_m\epsilon)^T\Big)$$

$$\sqcup$$

$$snd\Big(Eval_{\mathbf{ID}}(f_2, [\![f_0 \leq c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{e}}_1\epsilon)^T \times ... \times Eval_{\mathbf{ID}}(f_2, [\![f_0 \leq c]\!](\widehat{\mathbf{x_0}}) + \widehat{\mathbf{e}}_m\epsilon)^T\Big)$$

By the inductive hypothesis, this is just

$$= \partial^{Int}(f_1, [\![f_0 \geq c]\!](\widehat{\mathbf{x_0}})) \sqcup \partial^{Int}(f_2, [\![f_0 \leq c]\!](\widehat{\mathbf{x_0}}))$$

which is of course just equal to

$$= \partial^{Int}(f_0 > c \ ? \ f_1 : f_2, \widehat{\mathbf{x_0}})$$

Case 2.12. Composition. We require $f_1 : \mathbb{R}^m \to \mathbb{R}$. We start with the desired RHS:

$$snd\Big(Eval_{\mathbb{ID}}(C^1(f_1), \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)^T \times ... \times Eval_{\mathbb{ID}}(C^1(f_1), \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)^T\Big)$$

By the definition of $Eval_{\mathbb{ID}}$,

$$= snd\Big(Eval_{\mathbb{ID}}(C^1, Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon))^T \times ... \times Eval_{\mathbb{ID}}(C^1, Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon))^T\Big)$$

Distributing the snd element-wise:

$$= snd\Big(Eval_{\mathbb{ID}}(C^1, Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon))^T\Big) \times ... \times snd\Big(Eval_{\mathbb{ID}}(C^1, Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon))^T\Big)$$

Applying the definition of $Eval_{\mathbb{ID}}(C^1, \cdot)$ and then applying snd:

$$= J^{Int}\Big(C^1, fst\Big(Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)\Big)\Big) \cdot_{\mathbb{IR}} snd\Big(Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)\Big) \times ...$$

$$... \times J^{Int}\Big(C^1, fst\Big(Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)\Big)\Big) \cdot_{\mathbb{IR}} snd\Big(Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)\Big)$$

Simplifying each $fst\Big(Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_i\epsilon)\Big)$ term:

$$= J^{Int}\Big(C^1, f_1(\widehat{\mathbf{x_0}})\Big) \cdot_{\mathbb{IR}} snd\Big(Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)\Big) \times ...$$

$$... \times J^{Int}\Big(C^1, f_1(\widehat{\mathbf{x_0}})\Big) \cdot_{\mathbb{IR}} snd\Big(Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)\Big)$$

Since $J^{Int}\Big(C^1, f_1(\widehat{\mathbf{x_0}})\Big)$ is common to each element, it can be factored out:

$$= J^{Int}\Big(C^1, f_1(\widehat{\mathbf{x_0}})\Big) \cdot_{\mathbb{IR}} \Big(snd\Big(Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)\Big) \times ... \times snd\Big(Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)\Big)\Big)$$

But by the inductive assumption, the rightmost term reduces to $\partial^{Int}(f_1, \widehat{\mathbf{x_0}})$, giving:

$$= J^{Int}(C^1, f_1(\widehat{\mathbf{x_0}})) \cdot_{\mathbb{IR}} \partial^{Int}(f_1, \widehat{\mathbf{x_0}})$$

$$= \partial^{Int}(C^1(f_1), \widehat{\mathbf{x_0}})$$

Case 2.13. Cartesian Product. We start with the desired LHS:

$$snd\Big(Eval_{\mathbb{ID}}(f_1 \times f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)^T \times ... \times Eval_{\mathbb{ID}}(f_1 \times f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)^T\Big)$$

Applying the definition of $Eval_{\mathbb{ID}}(f_1 \times f_2, \cdot)$,

$$= snd\Big(\Big(Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon) \times Eval_{\mathbb{ID}}(f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)\Big)^T \times ...$$

$$... \times \Big(Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon) \times Eval_{\mathbb{ID}}(f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)\Big)^T\Big)$$

By properties of transpose T:

$$= snd\Big(\begin{bmatrix} Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)^T \\ Eval_{\mathbb{ID}}(f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)^T \end{bmatrix} \times ... \times \begin{bmatrix} Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)^T \\ Eval_{\mathbb{ID}}(f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)^T \end{bmatrix}\Big)$$

By properties of $\times$:

$$= snd\Big(\begin{bmatrix} Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)^T \times ... \times Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)^T \\ Eval_{\mathbb{ID}}(f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)^T \times ... \times Eval_{\mathbb{ID}}(f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)^T \end{bmatrix}\Big)$$

Distributing *snd*,

$$= \begin{bmatrix} snd\Big(Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)^T \times ... \times Eval_{\mathbb{ID}}(f_1, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)^T\Big) \\ snd\Big(Eval_{\mathbb{ID}}(f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_1\epsilon)^T \times ... \times Eval_{\mathbb{ID}}(f_2, \widehat{\mathbf{x_0}} + \widehat{\mathbf{e}}_m\epsilon)^T\Big) \end{bmatrix}$$

By the inductive assumption:

$$= \begin{bmatrix} \partial^{Int}(f_1, \widehat{\mathbf{x_0}}) \\ \partial^{Int}(f_2, \widehat{\mathbf{x_0}}) \end{bmatrix}$$

$$= \partial^{Int}(f_1 \times f_2, \widehat{\mathbf{x_0}})$$

QED.

2.6.4 Complexity

The key insight of Theorem 2.14 is that, as with standard forward-mode AD, the dual interval abstraction requires as many forward passes as there are inputs. Therefore, for $f : \mathbb{R}^m \to \mathbb{R}^n$, we require m independent passes of $Eval_{\mathbb{ID}}$ in order to compute the full interval over-approximation of the Clarke Jacobian. However, they can be computed in parallel (as we do). Hence, as with standard forward-mode AD, our method is better suited for functions where the input dimension is small. Additionally, the real part for each pass will be the same, and need only be computed once.

Furthermore, for the basic arithmetic functions, dual interval arithmetic requires more primitive operations than the same operation defined over the reals. For example, multiplication of two dual intervals as shown in Section 2.6.1 requires 12 primitive multiplications to perform all the interval arithmetic multiplications for both the real and dual parts, as well as 2 primitive additions. However, for all arithmetic operations $\{+, -, \cdot, /\}$, the amount of additional primitive operations required for the dual interval version (vs. the real-valued version) is still just a constant factor more.

While it has been established that for standard AD, the amount of primitive operations of a single pass of forward-mode evaluation with dual numbers is at most a constant amount (typically $2 - 3$x) more than the number of operations to evaluate the function itself [5], this is not the case for our analysis. A key complexity issue arises due to our support of branching. As the abstract evaluator $Eval_{\mathbb{ID}}$ could potentially evaluate both branches of a conditional, we may have *exponentially* many more evaluations compared to evaluating the function with ordinary real numbers. Thus, if $OPS(f)$ denotes the number of primitive arithmetic operations to evaluate f (not its Jacobian) over real numbers, then $\mathcal{O}(2^{OPS(f)})$ is the upper bound on the number of primitive arithmetic operations required to evaluate a single pass of the dual interval lifted version of f. For the evaluation, the ReLU function as well as the contrast variation and rotation perturbations employ branching. Yet, despite the theoretically worst-case exponential operation complexity, in practice, DeepJ's implementation of $Eval_{\mathbb{ID}}$ was still quite fast. Lastly, each term will always have only two intervals associated to it (the real and dual parts). Hence, unlike affine interval arithmetic [70] or tools like Rosa [71], the number of intervals we must track per variable does *not* grow as a function of the expression size.

2.7 METHODOLOGY

We next describe the experimental setup, including the perturbation functions and networks used in our experiments. We also detail the training procedures and accuracies of these networks. We ran our experiments on a 2.20 GHz 14 core Intel Xeon Gold 5120 CPU with 256 GB of main memory.

Implementation

We implemented our analysis in a tool called DeepJ, written in C++. DeepJ employs operator and function overloading to allow dual interval inputs to be propagated through arbitrary perturbation functions and neural networks. Even though our formalization is from a purely functional view, one can easily use an ANF conversion [72] to produce imperative code, since our programs do not have recursions or while loops. All code, neural networks, and results are available at `https://github.com/uiuc-arc/DeepJ`.

Perturbation Functions

We consider three previously studied image perturbations affecting pixel intensity and image geometry, as well as compositions of these perturbations. We assume that the images' pixel values are in the range $[0, 1]$.

1. **Haze** [61]: The intensity x_i of the i^{th} pixel in the original image is perturbed to $(1 - \alpha)x_i + \alpha$, where $\alpha = [0, \alpha_{max}] + [1, 1]\epsilon$; the parameter $\alpha_{max} \in [0, 1]$ represents the degree of haze.

2. **Contrast** [61]: The intensity x_i of the i^{th} pixel in the original image is perturbed to $\max(0, \min(1, \frac{x_i - 0.5 \cdot \alpha}{1-\alpha}))$, where $\alpha = [0, \alpha_{max}] + [1, 1]\epsilon$; the parameter $\alpha_{max} \in [0, 1]$ represents the degree of contrast.

3. **Rotation** [57]: We analyze image rotations with bilinear interpolation within a range of angles θ, where $\theta = [-\theta_{max}, \theta_{max}] + [1, 1]\epsilon$; the parameter $\theta_{max} \in \mathbb{R}_{\geq 0}$ represents the rotation angle in radians.

4. **Composition:** We look at three ways of composing the above functions – haze followed by rotation, contrast followed by rotation, and contrast followed by haze.

The contrast, rotation, and composite perturbations described above are non-differentiable but Lipschitz continuous, and cannot be handled by prior work [15, 40, 41, 42]. In our

experiments, we consider $\alpha_{max} \leq 0.63$ for the haze and contrast perturbations, and $\theta_{max} \leq 0.32$ radians (which corresponds to approximately $\pm 18°$) for rotation.

Network Architectures

In total we trained 6 networks: three ReLU networks each for the CIFAR10 and MNIST datasets. The first network (FFNN) is a 7-layer fully-connected architecture from RecurJac [15]. The other networks are convolutional networks from [56] – ConvMed features two convolutional and two fully-connected layers, while ConvBig features four convolutional and three fully-connected layers. Details on these networks are shown below. We denote Conv(o, k, s, p) as a convolutional layer with o output channels, k x k kernel width and height, stride s, and padding p; we denote Fc(o) as a fully-connected layer that outputs o neurons. ReLU(l) indicates that a ReLU activation is applied element-wise to the outputs of layer l. The networks are as follows:

1. FFNN: Seven-layer fully-connected network with 1024, 512, 256, 128, 64, and 32 hidden neurons and a ReLU activation after every layer (including the final layer).
2. ConvMed: ReLU(Conv(16, 4, 2, 1)) $\rightarrow$ ReLU(Conv(32, 4, 2, 1)) $\rightarrow$ ReLU(Fc(100)) $\rightarrow$ Fc(10)
3. ConvBig: ReLU(Conv(32, 3, 1, 1)) $\rightarrow$ ReLU(Conv(32, 4, 2, 1)) $\rightarrow$ ReLU(Conv(64, 3, 1, 1)) $\rightarrow$ ReLU(Conv(64, 4, 2, 1)) $\rightarrow$ ReLU(Fc(512)) $\rightarrow$ ReLU(Fc(512)) $\rightarrow$ Fc(10)

Our largest network is the CIFAR ConvBig network containing $>$ 62 K neurons. These network sizes are comparable to those of other state-of-the-art verification techniques [53]. Furthermore, for the local optimization landscape experiment, we trained 7 fully-connected networks on the MNIST dataset, varying the total number of layers from 3 to 9. Each hidden layer contains 30 neurons, and a ReLU activation is applied after every layer (including the final layer).

Data Transformation

For all MNIST networks, we transformed the training set so that for each image, we padded it by 4 and took a random 28x28 crop of the resulting 32x32 image. For the CIFAR10 networks, we introduced random cropping with padding 4 *and* randomly flipping each image horizontally with probability 0.5. Afterwards, we normalized the images using $\mu =$

$0.1307, \sigma = 0.3081$ for MNIST and $\mu = (0.4914, 0.4822, 0.4465), \sigma = (0.2023, 0.1994, 0.2010)$ for CIFAR10.

Training Hyperparameters

For the 7 networks trained for the local optimization experiment, we used the Adam optimizer [73] with a learning rate of 10^{-4} and L2-regularization with $\lambda = 0.001$; we trained with a batch size of 500 for 60 epochs, using 6,000 images from the training set for validation. For all other networks, we used the Adam optimizer with a learning rate of 10^{-3} on MNIST and 10^{-4} on CIFAR10. We trained the MNIST networks with a batch size of 500 for 30 epochs, using 6,000 images from the training set for validation; we trained the CIFAR10 networks with a batch size of 64 for 60 epochs, using 5,000 images from the training set for validation. In all cases, we then picked the model that attained the highest validation accuracy out of all epochs.

Network Accuracies

The fully-connected networks trained for the local optimization landscape experiment all attain test accuracy of at least 92%. The classification accuracies for the networks in the Lipschitz experiments are shown in Table 2.1.

Table 2.1: Classification accuracy on test set for our networks.

	CIFAR10	MNIST
FFNN	56.71	98.34
ConvMed	67.26	98.95
ConvBig	79.58	99.50

2.8 EXPERIMENTAL EVALUATION

We evaluate the effectiveness of our approach on two tasks: (i) Lipschitz robustness certification and (ii) optimization landscape analysis. Both tasks are defined across a variety of datasets, perturbation functions, and neural network architectures, demonstrating our language's flexibility and scalability.

2.8.1 Lipschitz Robustness

We compute an upper bound on the Lipschitz constant of functions of the form $f \circ n \circ p$, where p denotes one or more perturbation functions, n is an input normalization function (which simply rescales images by a constant), and f is a neural network. The correctness of bounding the Lipschitz constant via the Interval Clarke Jacobian is proved in Section 2.5.3. We use the ℓ_∞-norm for the calculation of all Lipschitz constants. For both individual and composite perturbations, we consider five versions of our tool: DeepJ with the given range for the input perturbation parameter(s) and four versions of DeepJ where the input intervals are subdivided uniformly, denoted DeepJ kx, where k represents how many subintervals are used *per input parameter*. When using splitting, we compute the upper bound on the Lipschitz constant separately for each split and take the maximum constant across the splits. Each pass of our analysis per image and split is completely independent of one another, and is thus parallelized in our implementation. Since no existing work can handle functions of the form $f \circ n \circ p$ considered in our evaluation, we employ a baseline combining global and local Lipschitz analysis. The baseline computes $L_{loc}(p) \cdot L_{global}(n) \cdot L_{global}(f)$, where L_{loc} and L_{global} are the local and global Lipschitz constants of their corresponding functions. As p is not globally Lipschitz, we obtain $L_{loc}(p)$ with DeepJ. We compute $L_{global}(n)$ as the reciprocal of the standard deviation (MNIST) or the reciprocal of the smallest standard deviation of the three RGB channels (CIFAR10) used for normalizing the input. We calculate $L_{global}(f)$ by multiplying the norm of each layer's weights using the tool from [51].

Results for Individual Perturbations Figure 2.7 shows the Lipschitz constant results for single perturbations on our larger convolutional architectures for both MNIST and CIFAR10. The results for the remaining networks are shown in Fig. 2.8. The x-axis for the haze and contrast perturbations shows the value of α_{max} used for defining the input range $\alpha = [0, \alpha_{max}] + [1, 1]\epsilon$, while for rotation, the x-axis shows θ_{max} used for defining the input range $\theta = [-\theta_{max}, \theta_{max}] + [1, 1]\epsilon$. The y-axis shows the upper bound on the Lipschitz constant computed with different methods. Both axes use logarithmic scales. Each data point is the average over 100 correctly classified images, selected by taking the first 10 correctly classified test-set images from each output category. The same set of images are used per dataset for each experiment. We use $\alpha_{max} \in \{10^{-k/4} \cdot 2 \mid k \in [2, 18]\}$ for haze and contrast and $\theta_{max} \in \{10^{-k/4} \mid k \in [2, 18]\}$ for rotation. For the rotation perturbation, the input to our function is in radians and covers the same range of interval sizes as the other two perturbations, but we convert the units to degrees for clearer presentation.

In all cases, DeepJ is more precise than the baseline, often obtaining Lipschitz bounds

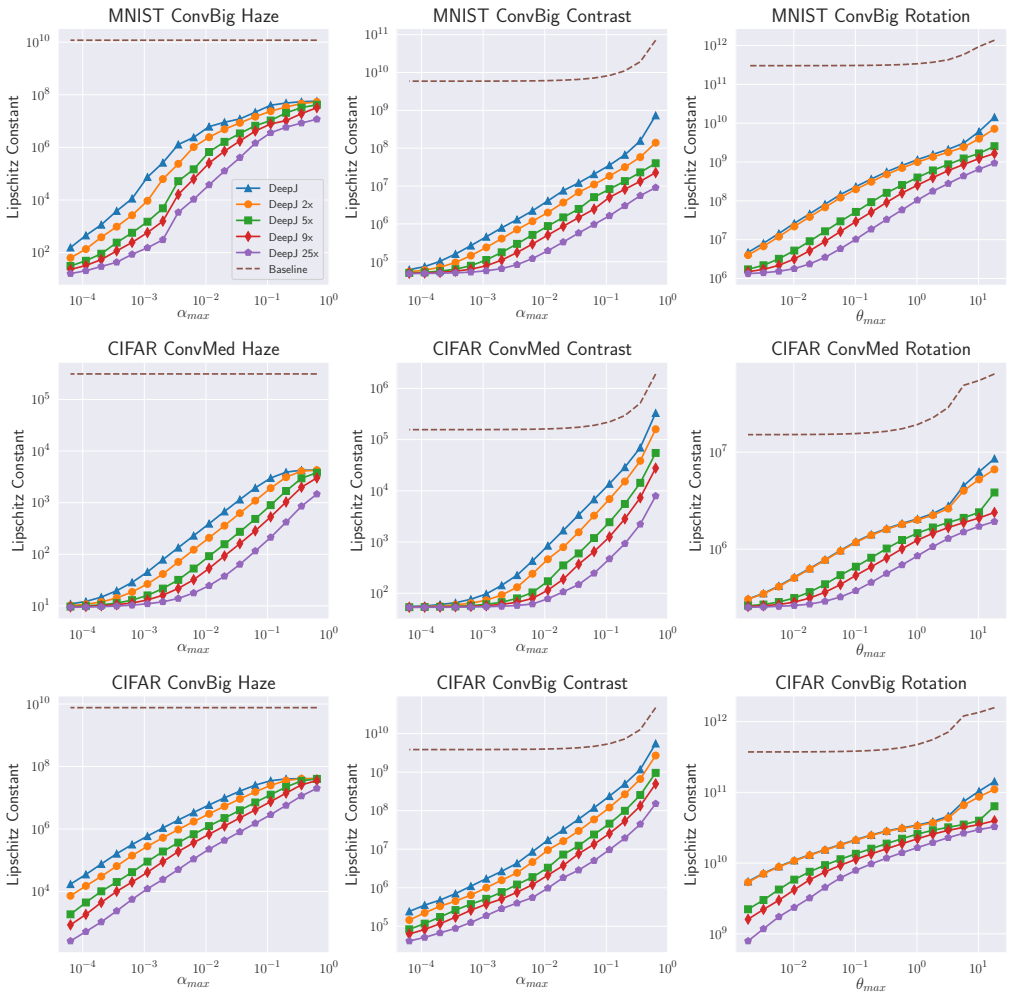


Figure 2.7: Upper bounds on the Lipschitz constants with respect to individual perturbations for MNIST ConvBig, CIFAR ConvMed, and CIFAR ConvBig networks.

orders of magnitude smaller than the baseline. The computed bounds become larger as the perturbation size is increased. It can also be seen that increased splitting leads to more precise results, with DeepJ 25x achieving much lower bounds than vanilla DeepJ (which does not do splitting).

Table 2.2 shows statistics on the runtime of the different methods for the same networks from Figure 2.7 and 2.8. We consider all values of the parameters on the x-axis shown in Fig. 2.7. We report the minimum, median, and maximum runtimes across the 100 images for each function. DeepJ 25x takes the longest to run due to more splits, while the baseline usually runs the fastest, except for rotation on large angles. In some cases, splitting may be faster than the baseline or vanilla DeepJ, since without splitting, intervals can become too over-approximate; evaluating conditionals (for ReLUs) on over-approximate intervals often requires evaluating both branches, which may lead to the exponential blowup phenomenon

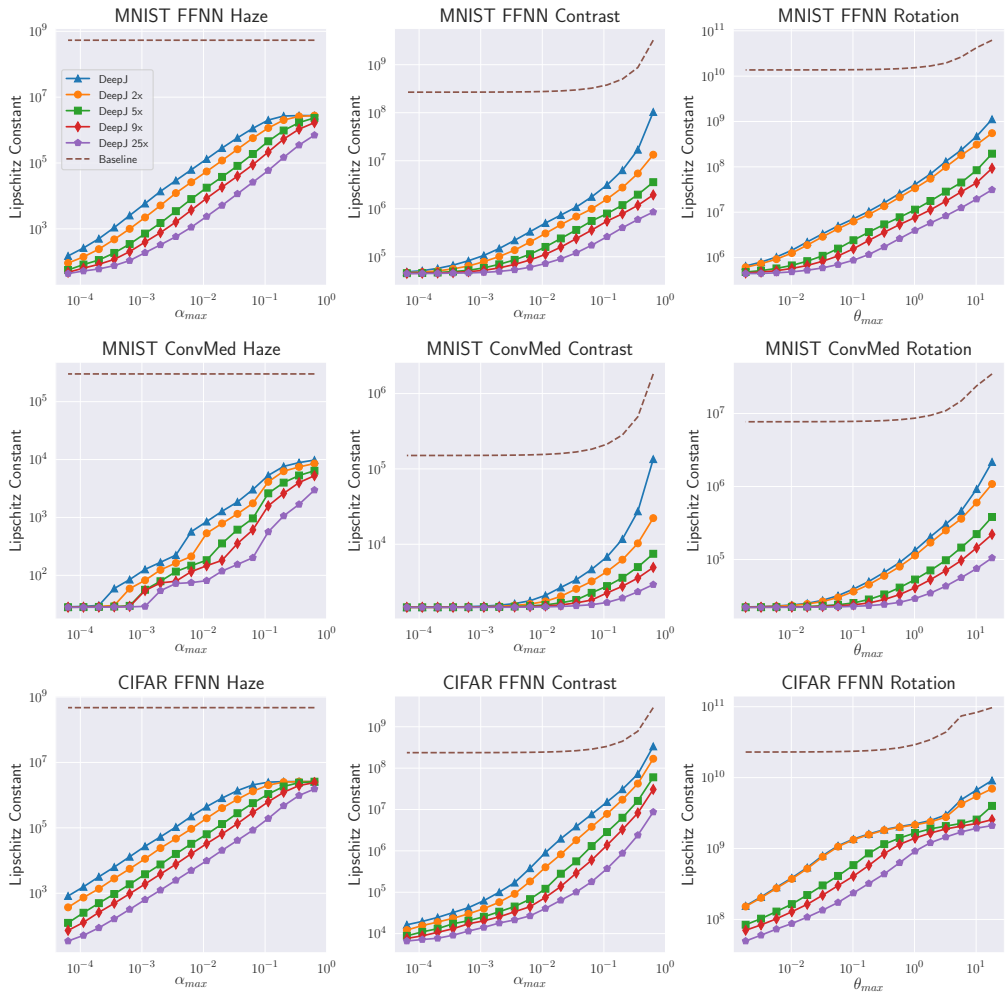


Figure 2.8: Upper bounds on the Lipschitz constants with respect to individual perturbations for the MNIST FFNN, MNIST ConvMed, and CIFAR FFNN networks.

discussed in Section 3.2. For rotation with bilinear interpolation, we empirically observed this beyond ± 0.1 radians.

The computation of $L_{loc}(p)$ via our method contributes the most to the runtime of the baseline. For all versions of DeepJ, the rotation perturbation has the highest runtime. For a given perturbation type, the runtime increases with the size of the network. On the most expensive rotation perturbation with the CIFAR ConvBig network, the median time for DeepJ to finish is under 1.4 min per 100 images. Finally, the precision of our analysis can be improved by increasing the number of splits (as seen in Figure 2.7) at the cost of additional runtime (as seen in Table 2.2).

Results for Composite Perturbations Figure 2.9 shows the upper bounds on the Lipschitz constant computed for compositions of perturbations on the same networks as in

Table 2.2: Runtimes in seconds to compute Interval Clarke Jacobians for individual perturbations over 100 images. For each network, the six rows represent the times for DeepJ vanilla, 2x, 5x, 9x, 25x, and Baseline.

	Haze			Contrast			Rotation		
	Min	Med	Max	Min	Med	Max	Min	Med	Max
MNIST ConvBig	10.4	10.9	11.7	10.3	10.5	10.9	24.5	24.8	130.5
	20.6	21.1	21.9	20.4	20.7	21.4	48.9	49.2	129.8
	51.1	51.8	57.4	50.7	51.1	53.6	79.4	80.1	127.4
	91.5	101.5	391.8	91.4	93.1	359.9	134.4	141.9	311.5
	254.0	258.8	397.2	252.7	282.0	489.5	352.3	379.7	689.5
	0.05	0.06	0.07	0.04	0.06	0.08	13.6	14.0	114.9
CIFAR ConvMed	0.9	0.9	1.0	0.9	0.9	0.9	69.9	70.0	715.6
	1.7	1.7	1.8	1.7	1.7	1.8	139.9	140.1	614.2
	4.1	4.2	4.3	4.1	4.2	4.3	142.5	143.4	550.2
	7.3	7.4	7.6	7.3	7.4	7.5	215.7	241.7	630.2
	20.1	21.8	88.7	20.3	21.6	191.7	503.1	531.6	865.5
	0.05	0.05	0.07	0.06	0.06	0.07	65.4	66.2	685.6
CIFAR ConvBig	14.3	14.7	14.9	14.4	14.7	15.0	83.3	83.7	734.7
	28.5	29.1	29.5	28.6	29.2	29.6	165.6	167.0	635.4
	70.8	71.9	73.3	70.1	72.0	72.9	208.3	211.3	475.4
	127.3	130.3	142.9	127.5	131.2	150.5	335.1	339.3	768.1
	350.7	360.9	757.5	351.2	357.6	510.9	840.5	880.6	1029.7
	0.05	0.06	0.08	0.07	0.07	0.08	65.5	66.2	685.6
MNIST FFNN	1.7	1.7	1.8	1.7	1.8	2.7	15.5	15.7	117.9
	3.4	3.4	3.5	3.3	3.4	3.5	30.9	31.0	107.3
	8.3	8.6	9.2	8.2	8.2	9.4	36.1	36.6	81.2
	15.0	15.1	15.3	14.8	14.9	93.9	57.3	57.9	96.5
	41.6	68.8	371.4	40.7	57.9	70.8	141.3	208.4	475.7
	0.05	0.06	0.07	0.04	0.06	0.08	13.6	14.0	114.9
MNIST ConvMed	0.6	0.6	0.6	0.6	0.6	0.7	14.9	14.9	120.5
	1.1	1.1	1.2	1.1	1.1	1.2	29.5	29.7	108.8
	2.7	2.7	2.8	2.7	2.7	2.7	31.2	31.3	77.4
	4.7	4.8	4.9	4.7	4.7	4.8	47.4	53.4	652.6
	13.0	13.4	22.3	12.8	13.0	22.0	112.4	150.2	210.4
	0.04	0.05	0.06	0.04	0.05	0.07	13.6	14.0	114.9
CIFAR FFNN	4.3	4.3	4.4	4.3	4.3	4.4	71.2	71.4	704.0
	8.5	8.6	10.2	8.5	8.6	9.1	140.6	142.2	599.6
	21.1	21.2	21.4	21.1	21.3	25.7	154.3	155.9	582.2
	38.4	38.6	38.8	38.4	38.7	198.7	246.8	265.0	470.9
	106.9	158.1	312.5	106.1	156.9	331.8	591.0	853.4	1104.1
	0.06	0.07	0.08	0.07	0.08	0.09	65.5	66.2	685.6

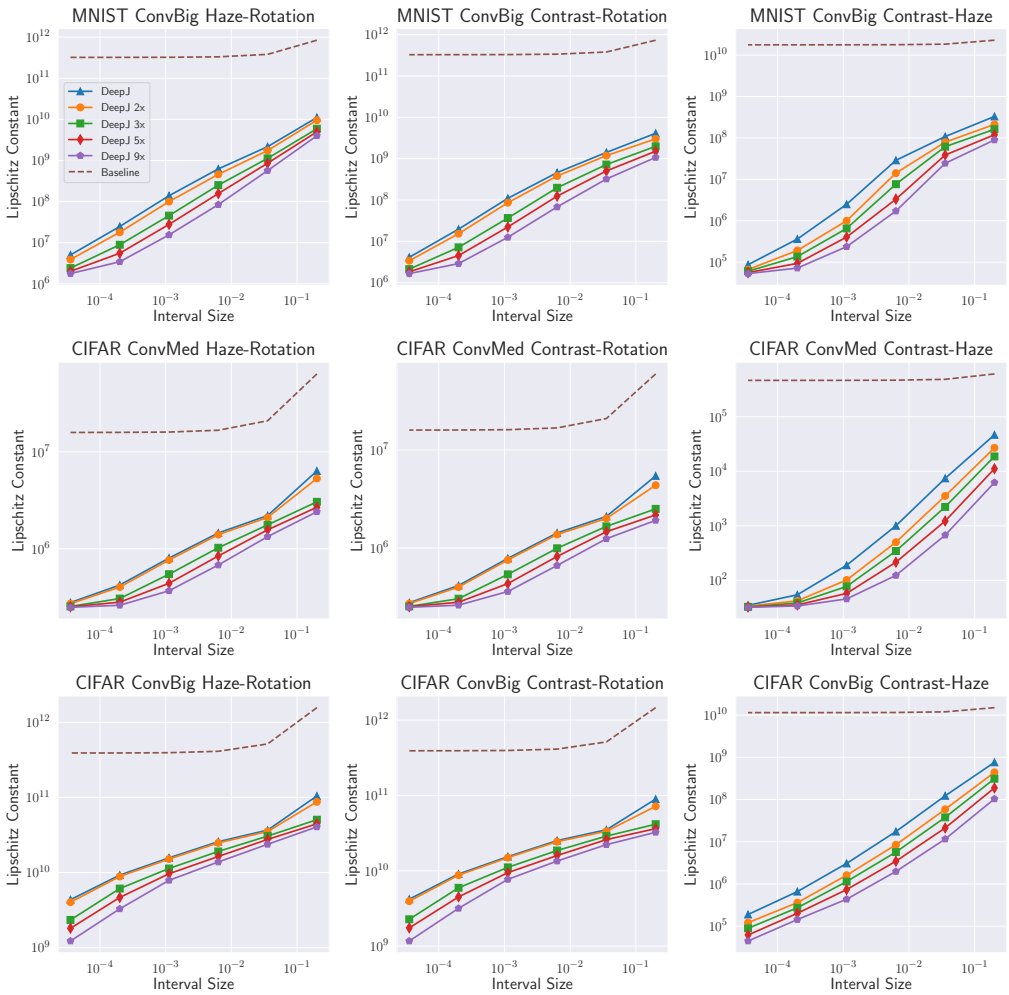


Figure 2.9: Upper bounds on the Lipschitz constants with respect to composite perturbations for the MNIST ConvBig, CIFAR ConvMed, and CIFAR ConvBig networks.

Figure 2.7. The results for the remaining networks are shown in Figure 2.10. Haze-Rotation denotes a haze perturbation composed with a rotation, in that order; the terminology is similar for the other composite perturbations. We use the same interval width when perturbing each parameter independently; the x-axis shows this width. For compositions that involve rotation, if the interval size on the x-axis is denoted s, we utilize the real interval $[-s/2, s/2]$ for rotation and the interval $[0, s]$ for the other perturbation. The y-axis shows the upper bound on the Lipschitz constant computed by each method. Again, both axes use logarithmic scales. Each data point is the average over 10 correctly classified images, taking the first correctly classified test-set image from each output category. We use $s \in \{10^{-k/4} \cdot 2 \mid k \in \{4, 7, 10, 13, 16, 19\}\}$ for all experiments.

As with individual perturbations, DeepJ is better than the baseline in all cases, obtaining upper bounds on the Lipschitz constants orders of magnitude smaller than the baseline, as

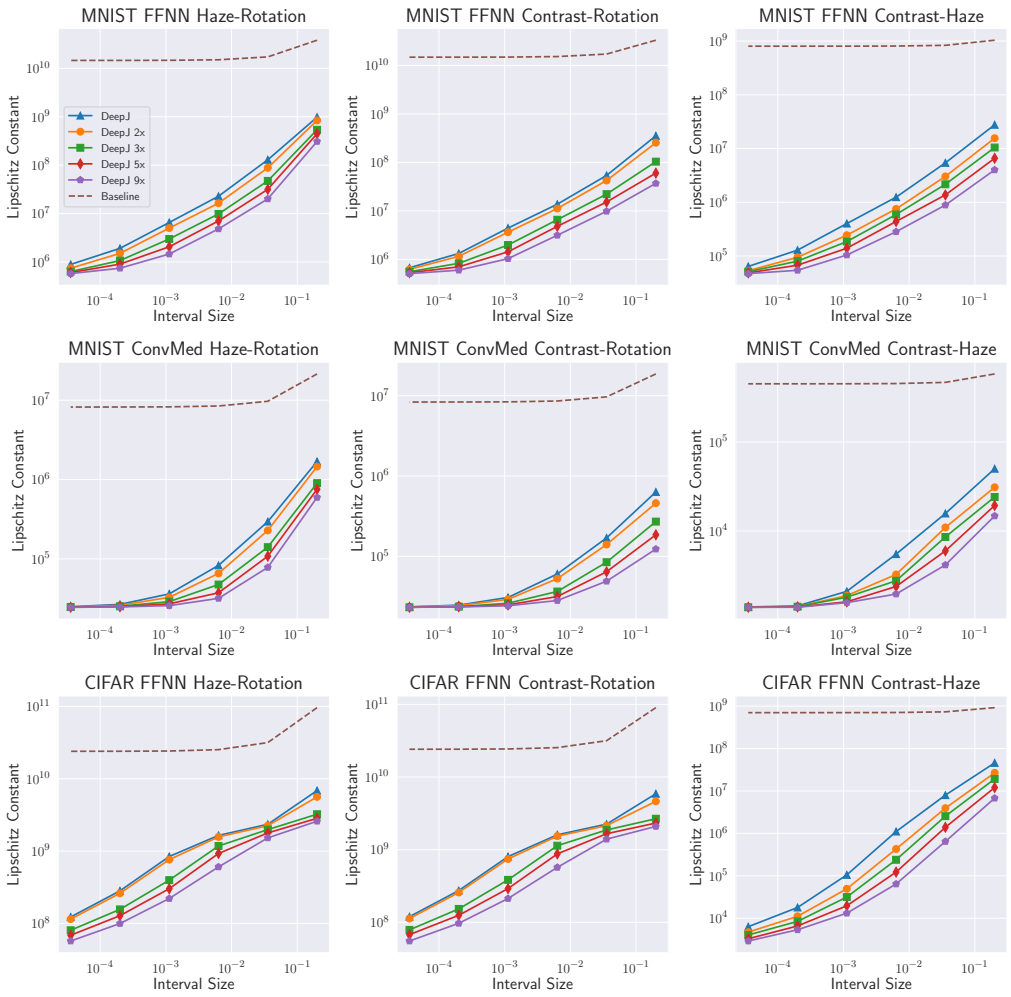


Figure 2.10: Upper bounds on the Lipschitz constants with respect to composite perturbations for the MNIST FFNN, MNIST ConvMed, and CIFAR FFNN networks.

seen in both Figure 2.9 and Figure 2.10. DeepJ 9x with the largest number of splits is the most precise. The Lipschitz constants for both the Haze-Rotation and Contrast-Rotation perturbations are nearly identical, since the entries in the Clarke Jacobian corresponding to the rotation variable have much higher magnitudes. Thus, these entries dominate the ℓ_∞-norm, overshadowing the effect of the other perturbation.

Table 2.3 shows the runtime statistics for the different methods in the same way as Table 2.2. We observe similar trends in the relative runtimes of the different methods as with individual perturbations. For all versions of DeepJ and for the baseline, the compositions that take the longest time to analyze are those containing rotations. On the most expensive composite perturbation (Haze-Rotation) with the CIFAR ConvBig network, the median time for DeepJ to finish is under 51 seconds per 10 images. As with individual perturba-

Table 2.3: Runtimes in seconds to compute Interval Clarke Jacobians for composite perturbations over 10 images. For each network, the six rows represent the times for DeepJ vanilla, 2x, 3x, 5x, 9x, and Baseline.

	Haze-Rotation			Contrast-Rotation			Contrast-Haze		
	Min	Med	Max	Min	Med	Max	Min	Med	Max
MNIST ConvBig	14.8	14.9	24.4	14.7	14.9	20.9	6.3	6.4	6.7
	58.7	59.6	72.2	58.8	59.1	70.9	24.8	25.2	26.0
	76.0	76.6	78.0	75.7	76.3	77.6	33.8	34.2	35.0
	102.5	103.4	107.6	102.3	102.5	105.7	52.3	52.8	55.4
	247.7	251.3	263.6	246.4	248.8	257.3	158.6	160.0	174.0
	8.2	8.2	13.9	8.2	8.2	13.8	0.06	0.07	0.08
CIFAR ConvMed	42.2	42.2	80.7	42.1	42.3	82.0	0.6	0.8	1.8
	167.2	167.6	239.2	167.9	173.0	251.3	2.0	2.1	2.2
	209.1	209.4	216.1	209.3	210.1	214.9	2.9	3.1	3.5
	247.4	247.9	264.1	247.4	247.6	265.3	4.3	4.4	4.8
	527.7	530.6	559.6	526.1	530.1	561.7	12.7	13.3	13.5
	39.4	39.4	79.1	39.4	39.4	76.1	0.05	0.05	0.07
CIFAR ConvBig	49.9	50.4	89.1	50.0	50.1	89.7	8.6	8.8	16.8
	198.9	199.8	270.9	198.8	199.3	269.7	34.2	34.7	35.2
	251.1	252.5	257.9	251.1	251.9	258.5	46.0	46.7	52.4
	313.9	315.0	328.3	314.2	315.9	327.8	72.7	73.7	74.7
	695.3	701.8	728.3	693.6	696.5	725.9	220.6	224.0	227.1
	39.4	39.4	79.1	39.4	39.4	76.1	0.06	0.06	0.08
MNIST FFNN	9.4	9.7	15.3	9.3	9.4	15.1	1.0	1.1	1.1
	37.4	42.2	48.5	37.2	37.3	48.0	4.0	4.0	4.1
	47.9	48.3	48.8	47.9	48.2	49.2	5.6	5.8	6.1
	58.6	58.6	65.3	58.6	58.8	61.2	8.7	9.0	9.2
	125.9	128.6	129.9	124.6	126.6	133.9	26.5	26.7	27.1
	8.2	8.2	13.9	8.2	8.2	13.8	0.06	0.07	0.08
MNIST ConvMed	9.0	9.5	15.4	9.1	9.4	15.3	0.4	0.5	1.6
	35.5	35.6	46.7	35.5	35.6	46.5	1.3	1.3	1.4
	44.8	45.3	45.7	44.5	44.8	47.7	2.0	2.2	2.5
	53.3	53.5	56.3	53.1	53.2	55.4	2.8	2.9	3.1
	113.5	114.9	119.1	113.2	113.8	118.1	8.2	9.3	9.5
	8.2	8.2	13.9	8.2	8.2	13.8	0.05	0.06	0.07
CIFAR FFNN	42.7	42.7	80.2	42.7	42.8	80.6	2.6	2.6	2.7
	171.5	172.3	239.7	171.4	174.0	240.8	11.6	16.8	49.4
	217.8	218.6	224.6	217.9	218.2	223.9	14.2	14.6	19.5
	262.4	263.0	284.6	262.8	263.2	275.8	22.3	22.4	22.8
	568.8	571.4	600.8	565.5	569.0	606.0	67.6	68.0	73.1
	39.4	39.4	79.1	39.4	39.4	76.1	0.06	0.07	0.08

Table 2.4: Error and overhead of floating-point sound computations for DeepJ. For each network, the two rows represent maximum relative error and maximum relative time overhead, respectively.

	Haze	Contrast	Rotation	Haze-Rotation	Contrast-Rotation	Contrast-Haze
MNIST ConvBig	2.24e-11 3.67x	9.34e-13 3.78x	1.23e-12 3.51x	1.24e-12 3.59x	1.25e-12 3.58x	9.77e-13 4.08x
CIFAR ConvMed	8.08e-12 3.58x	1.79e-12 3.68x	8.93e-13 2.83x	9.07e-13 3.28x	8.99e-13 3.25x	5.62e-12 3.45x
CIFAR ConvBig	2.84e-11 3.45x	2.81e-12 3.43x	1.30e-12 2.94x	1.36e-12 3.31x	1.36e-12 3.27x	6.85e-12 3.66x

tions, when analyzing composite perturbations, the precision of DeepJ can be improved by performing more splits at the cost of additional runtime.

Our tool, though implemented with floating-point precision, assumes ideal real arithmetic. To ensure this assumption does not lead to substantially different results, we also implemented a floating-point sound version of DeepJ using the techniques in [48]. We ran floating-point sound experiments for vanilla DeepJ. Table 2.4 shows that the difference in the computed Lipschitz constants is negligible ($< 10^{-10}$) in all cases. However, ensuring floating-point soundness adds up to 4.1x runtime overhead.

2.8.2 Local Optimization Landscape Analysis

Obtaining an Interval Clarke Jacobian allows us to study the local geometry of $f \circ n \circ p$. We focus on finding the largest input regions where no stationary point exists. To prove that a given region does not contain a stationary point, we check if there exists an interval entry $[l_{i,j}, u_{i,j}]$ in the Interval Clarke Jacobian such that $l_{i,j} > 0$ or $u_{i,j} < 0$. If this holds, then we can guarantee that the Clarke Jacobian matrix will not become zero, and therefore no stationary point exists within the given input region (which follows from Theorem 2.3.2 of [39]). Hence, similar to RecurJac [15], we study the relationship between network depth and the maximal interval size for which the absence of stationary points can still be guaranteed. However, RecurJac cannot handle functions of the form $f \circ n \circ p$ considered in our work.

We define the set of input intervals to analyze as $\{[0, 10^{-k/4} \cdot 2] \mid k \in [2, 21]\}$ for haze and contrast and $\{[-10^{-k/4}, 10^{-k/4}] \mid k \in [2, 21]\}$ for rotation. We consider 100 test images from MNIST that are correctly classified, using the first 10 correctly classified test images from each category. For each combination of perturbation type and network, we compute the largest input region where a stationary point does not exist. For each image, we uniformly split every input interval into 25 subintervals, computing the Interval Clarke Jacobians separately for each subinterval. Next, we identify the subintervals where the entry in the Interval

Table 2.5: Largest interval sizes that guarantee no stationary point exists for CIFAR10 (left) and MNIST (right) networks, averaged over 100 images.

	Haze	Contrast	Rotation
FFNN	6.2e-5	7.5e-5	3.6e-7
ConvMed	4.2e-3	4.0e-3	4.4e-5
ConvBig	4.4e-6	4.7e-6	2.3e-8

	Haze	Contrast	Rotation
FFNN	9.7e-4	2.0e-3	6.7e-5
ConvMed	7.8e-3	1.8e-2	1.3e-3
ConvBig	1.3e-4	7.6e-4	1.9e-5

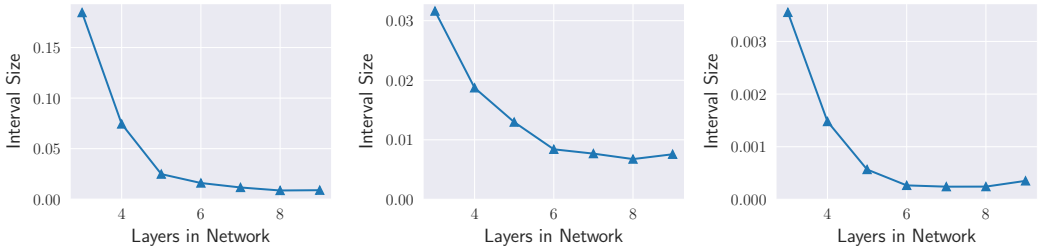


Figure 2.11: Largest interval size that guarantees no stationary point exists for Haze (left), Contrast (center), and Rotation (right), averaged over 100 images.

Clarke Jacobian corresponding to the correctly classified class is either strictly positive or negative. We sum the widths of all such subintervals, then compute the maximum of these sums across all candidate widths.

Figure 2.11 shows that as the depth of the fully-connected network increases, the maximal interval size that guarantees a non-zero Jacobian decreases for all perturbations. Further, we use the same procedure to obtain maximal interval sizes for the networks used in Section 2.8.1, as shown in Table 2.5. We observe that rotation requires smaller interval sizes as it is a more complex perturbation involving interpolation, whereas the other two are simpler pixel-wise operations. DeepJ can certify the absence of stationary points for larger regions on MNIST networks compared to CIFAR10, due to the former's smaller input dimension.

2.9 RELATED WORK

While there is considerable work on the static analysis of input-output properties of numerical programs and ML models specified via constraints on the inputs and outputs [52, 53, 54, 57, 74, 75, 76, 77, 78, 79, 80, 81, 82], such as for robustness and safety, much less work has been done on formally analyzing the Jacobian matrix.

From the programming languages (PL) and automatic differentiation (AD) literature, prior works [38, 43, 44, 46] have examined AD through the Clarke Jacobian [39]. However, many define their semantics for computing the Clarke Jacobian at a single point [43, 83], instead of an abstraction of points. Additionally, the works that can formally analyze Jaco-

bians for sets of input points are insufficient for our tasks. [44] restrict functions' domains to [-1,1] and require all Lipschitz constants be less than 1, thus their analysis cannot be used for local Lipschitz certification. The work by [47] is restricted to functions of a single variable with input domain on $[0, 1]$. Follow-up works [40, 45, 46] all suffer other restrictions, namely requiring the output dimension be one and only supporting limited arithmetic operations (e.g., no division), which render them unable to analyze both neural networks and our perturbations. Furthermore, these techniques provide only a theoretical discussion, with no implementation of their approaches. λ_S [38] presents semantics for the Clarke Jacobian for concrete input points, extended partially to intervals (described in their appendix) as compared to our approach, which presents a sound abstract interpretation of the Clarke Jacobian that we compute with a set of fully compositional dual interval domain transformers, whose behavior is specified exactly for all language primitives.

Another closely related PL work by Chaudhuri et al. [84] aims to establish Lipschitz robustness of programs, but they use the classical Jacobian and do not support AD. Follow-up works showed how to analyze Lipschitz robustness of non-differentiable programs by analyzing their smooth approximation [85], stemming from the fact that several PL works focus on differentiable approximations of non-differentiable programs and ML models [86, 87]. In contrast, DeepJ analyzes Lipschitz properties of programs with points of non-differentiability directly, by using the more general Clarke Jacobian.

In addition, while the practical problem of Lipschitz certification of neural networks has been studied [15, 42, 88, 89, 90], to the best of our knowledge, none of these works can bound local Lipschitz constants for composite, non-smooth perturbations. Both [41] and [42] use proof techniques that require the classifier function analyzed be piecewise linear, hence they cannot support arbitrary activations. Additionally, RecurJac [15] cannot support arbitrary arithmetic primitives like non-scalar multiplication or division. Hence, none of these works can reason about non-differentiable input perturbations, such as those generated by rotation [57] or contrast variation [61].

Our work also bears similarity with Rosa [71], as they also bound Jacobians to compute Lipschitz constants. However, unlike us, they do not support bounding Clarke Jacobians. Further, while their abstract domain tracks "ϵ-terms," the semantics of these terms do not correspond to first derivatives, but rather numerical round-off error effects.

2.10 SUMMARY

In this chapter, we developed a novel abstraction for bounding the Clarke Jacobian of a Lipschitz, but not necessarily differentiable function for local input regions. Our abstraction,

based upon dual numbers, soundly over-approximates all first derivatives needed to compute the Clarke Jacobian. We implemented our analysis in tool named DeepJ and showed that it can efficiently compute Lipschitz bounds and analyze the local geometry of multiple deep neural networks with respect to multiple non-differentiable input perturbations. Our work is the first to address the problem of local Lipschitz certification of non-smooth perturbations, such as haze, contrast variation, rotation with bilinear interpolation, and their compositions.

CHAPTER 3: A GENERAL CONSTRUCTION FOR ABSTRACT INTERPRETATION OF HIGHER-ORDER AUTOMATIC DIFFERENTIATION

We present a novel, general construction to abstractly interpret higher-order automatic differentiation (AD). Our construction allows one to instantiate an abstract interpreter for computing derivatives up to a chosen order. Furthermore, since our construction reduces the problem of abstractly reasoning about derivatives to abstractly reasoning about real-valued straight-line programs, it can be instantiated with almost any numerical abstract domain, both relational and non-relational. We formally establish the soundness of this construction.

We implement our technique by instantiating our construction with both the non-relational interval domain and the relational zonotope domain to compute both first and higher-order derivatives. In the latter case, we are the first to apply a relational domain to automatic differentiation for abstracting higher-order derivatives, and hence we are also the first abstract interpretation work to track correlations across not only different variables, but different orders of derivatives.

We evaluate these instantiations on multiple case studies, namely robustly explaining a neural network and more precisely computing a neural network's Lipschitz constant. For robust interpretation, first and second derivatives computed via zonotope AD are up to 4.76× and 6.98× more precise, respectively, compared to interval AD. For Lipschitz certification, we obtain bounds that are up to 11,850× more precise with zonotopes, compared to the state-of-the-art interval-based tool.

3.1 INTRODUCTION

Despite the widespread adoption of AD, as pointed out in [91], there is surprisingly little work on *formally verifying* that these programs compute correctly. While there have been several works [38, 44, 92] on proving correct the *concrete* semantics of AD, there is far less work on defining different *abstract* AD semantics for the purpose of verifying program properties expressed over derivatives.

While a few techniques exist, they remain significantly restricted: these techniques are either limited to the interval domain [7, 16, 34, 38] or do not support higher-order derivatives [34, 93]. Thus, precisely tracking relational information *across* higher derivatives is out of the realm of prior abstract interpreters, despite the fact that this precision is needed in tasks such as improving the performance of verified ODE solvers [94]. Further, most concrete semantics for computing higher-order derivatives are defined over complicated structures,

such as derivative towers [95] or lazily evaluated data structures as in [96] which poses additional difficulties for developing sound abstractions. Additionally, fundamental questions that arise when constructing an abstract semantics, such as how to systematically construct sound transformers for broad classes of functions or understanding the precision differences of various domains, have not been studied – especially for AD with higher-order derivatives.

Challenges This chapter focuses on developing a general framework for systematically constructing abstract semantics for AD with arbitrary order derivatives. The first key challenge is that we need to first define a concrete semantics of higher-order AD that lends itself to precise and scalable abstract interpretation – particularly with higher derivatives. This is challenging since the same abstract interpreter can have significantly different cost and precision for different syntactic representations of the same computation.

The second key challenge is ensuring that this abstract reasoning can then be done *generally* with expressive relational abstract domains and is not just constrained to the simpler interval domain, as in [7, 16, 34]. This problem is challenging since unlike in conventional programs, reasoning about higher-order AD requires precise abstract transformers for an exponential number of non-linear assignment statements which result from all the different partial derivative expressions. Designing these abstract transformers manually would require considerable expertise as well as substantial AD domain knowledge to know how the functional forms of different derivatives can be leveraged for precision.

Therefore, we need a generic construction that can systematically construct sound transformers for all higher-order derivatives by composing the transformers for a small number of primitive functions in our language. We also want to systematically leverage analytical properties of these derivatives to further refine the precision of the abstraction in a way that goes beyond naively composing existing numerical domains' abstract transformers together. Simultaneously addressing *all* of these challenges is beyond the scope of any existing work.

Our Work. This chapter presents a novel construction for abstractly interpreting higher-order AD to address these challenges. Our construction involves defining a concrete semantics for arbitrary order AD that semantically desugars the computation of derivatives into primitive operations on individual variables, and that also leverages data-dependence across derivatives in the program's syntactic representation. Thus, given only a small set of sound transformers for the elementary arithmetic primitives, we can immediately abstractly interpret these transformed AD programs, and still attain precision due to the dependency-aware syntactic representation of the AD program. Hence, we reduce the problem of formally reasoning about complicated AD semantics and having to manually design custom trans-

formers for all possible partial derivatives to reasoning about (straight-line) programs using standard abstract domains. This technique allows us generality in the abstraction – we can easily instantiate the construction with relational domains. We also show how to combine the abstract transformers with domain-specific knowledge about derivatives to further improve the precision of the analysis.

Contributions. The chapter makes the following contributions:

1. **Concrete AD Semantics:** We present a novel concrete semantics for higher-order AD. By designing our concrete semantics to be forward-mode and imperative, and by exposing each variable in a way that captures data dependence across derivatives, we enable one to intuitively define precise abstractions of these concrete semantics.

2. **Generic Abstract Interpretation of AD:** We provide the first generic construction to allow one to abstractly interpret forward-mode AD with arbitrary order derivatives. This construction is fully general and can be instantiated with both relational and non-relational abstract domains.

3. **Implementation:** We implement our formal construction into a practical tool and instantiate it with the interval and zonotope domains for first and higher derivatives, thus providing the first implementation of higher-order AD with relational abstract domains. Our implementation is publicly available at `https://github.com/uiuc-arc/AbstractAD` [97].

4. **Evaluation:** We empirically show the advantages of using a relational abstract domain and the benefits of abstractly interpreting higher-order derivatives through two case studies: (1) computing robust interpretations of regression neural networks and (2) bounding the Lipschitz constant of classification neural networks with respect to a semantic perturbation. For the robust interpretation task, we are the first to bound *both* first and second derivatives of a neural network with respect to an input *region*. First and second derivatives computed via zonotope AD are up to 4.76× and 6.98× more precise, respectively, compared to interval AD. For Lipschitz certification, we obtain bounds that are up to 11,850× more precise with zonotopes, compared to the state-of-the-art interval-based tool of [34].

3.2 EXAMPLE

We start with an illustrative example that highlights the generality of our construction.

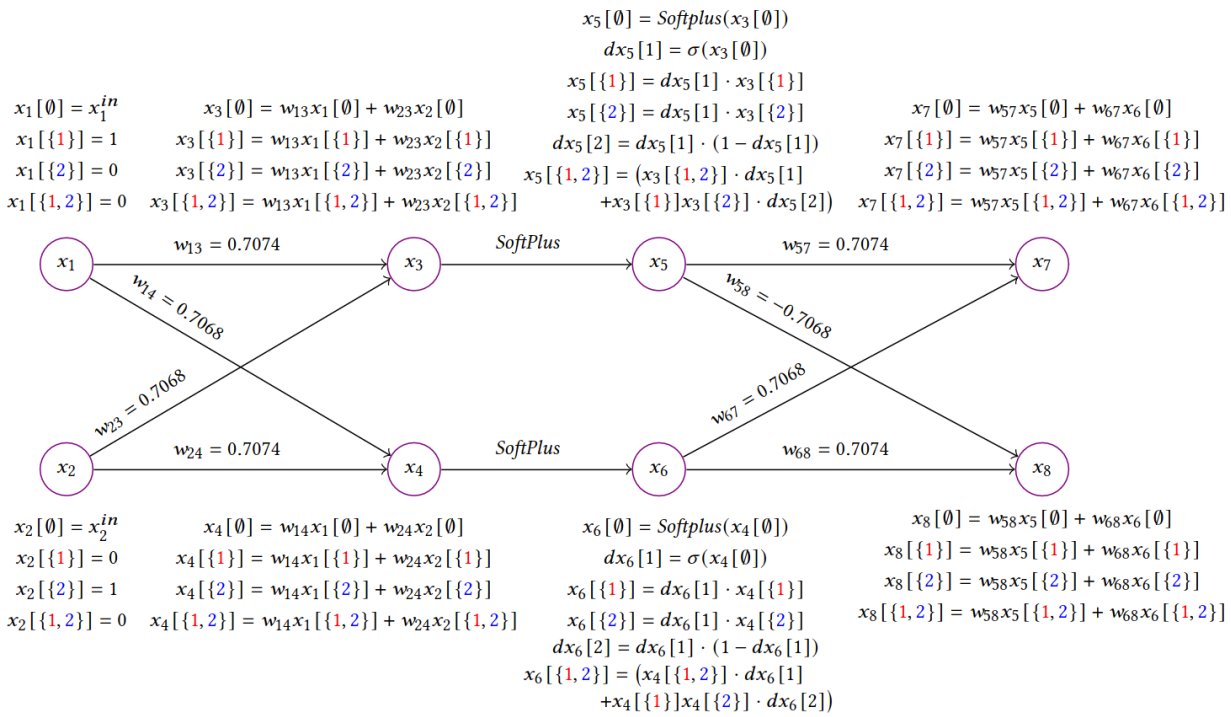


Figure 3.1: Concrete 2^{nd} order forward-mode AD evaluation

Running Example

Fig. 3.1 presents an example neural network. The network takes two inputs x_1 and x_2, propagates them through an affine layer to get the values of hidden neurons x_3 and x_4, applies a *SoftPlus* activation to get the values of x_5 and x_6, then passes those results through a final affine layer to get the outputs x_7 and x_8. Our goal is to compute second derivatives of the network's outputs (x_7 and x_8) with respect to the inputs (x_1 and x_2). These derivatives are useful for a variety of applications, such as if we want to explain the network by computing interactions between inputs (as in [19]) or if one wants to certify input regions where the function is locally concave or convex [16]. For this example, we study the impact of the interaction between x_1 and x_2 on the network output over a local region. The computation of the neural network output can be described by a straight-line imperative program, where each of the eight neurons is a program variable. However, when interpreting this program with forward-mode AD semantics, our interpreter will add additional augmented variables to keep track of all the derivatives (described in Section 3.4).

Second Derivative Analysis

While our prior work [34] can propagate 1^{st} derivatives through a neural network using the interval domain, for computing higher-order interactions of multiple variables, we need to compute *higher* derivatives with respect to the inputs, in this case 2^{nd} derivatives. To compute 2^{nd} derivatives, we instantiate a 2^{nd}-order instance of our construction. A key contribution of our approach is that it provides a general construction for giving both a concrete and abstract semantics for AD of arbitrary order.

For this example, each variable x_i in the original program will have *four* associated components in the forward-mode AD interpretation. The first of the four components is the "real part", denoted as $x_i[\emptyset]$, which intuitively corresponds to a "0^{th}" derivative. Equivalently, $x_i[\emptyset]$ is just the value of x_i, if the program were run under standard semantics instead of forward-mode AD semantics.

The next two components are x_i's first derivative terms, $x_i[\{1\}]$ and $x_i[\{2\}]$, which respectively index the first derivatives with respect to variable 1 and variable 2. Just as dual numbers (the canonical approach for first-order AD) only support differentiation of one variable at a time, our second-order analysis only supports differentiation of two variables at a time, which we explain formally in Theorem 3.2. In this example, variable 1 is just x_1 and variable 2 is just x_2, however this need not always be the case. Hence, for each x_i, $x_i[\{1\}] = \frac{\partial x_i}{\partial x_1}$ and $x_i[\{2\}] = \frac{\partial x_i}{\partial x_2}$. The last of the four components is the 2^{nd} derivative term $x_i[\{1,2\}]$, which tracks the derivative with respect to the first variable, then with respect to the second; equivalently, $x_i[\{1,2\}] = \frac{\partial^2 x_i}{\partial x_1 \partial x_2}$. Furthermore, because derivatives are symmetric (e.g., $\frac{\partial^2 x_i}{\partial x_1 \partial x_2} = \frac{\partial^2 x_i}{\partial x_2 \partial x_1}$), we do not need to worry about terms such as $x_i[\{2,1\}]$. In our example, we can think of the forward-mode AD semantics as propagating coefficients of a 2^{nd} degree Taylor polynomial that has been truncated to only include these terms. A key point is that our forward-mode AD semantics will automatically transform the original program to include additional variables to track these additional derivatives, which we will also call *augmented* variables.

These derivative will be evaluated not symbolically, but rather at specific points – in this example, all derivatives will be concretely evaluated at the scalar point $(x_1^{in}, x_2^{in}) \in \mathbb{R}^2$ shown in Fig. 3.1.

Lastly, to properly initialize the input state so that the concrete evaluation produces correct derivatives with respect to x_1 and x_2, we initialize the first derivative terms of the inputs as $x_1[\{1\}] = 1$ (since $\frac{dx_1}{dx_1} = 1$) and $x_2[\{2\}] = 1$ (since $\frac{dx_2}{dx_2} = 1$), while also setting $x_1[\{2\}] = x_2[\{1\}] = x_1[\{1,2\}] = x_2[\{1,2\}] = 0$, since $\frac{\partial x_1}{\partial x_2} = \frac{\partial x_2}{\partial x_1} = \frac{\partial^2 x_1}{\partial x_1 \partial x_2} = \frac{\partial^2 x_2}{\partial x_1 \partial x_2} = 0$. This initialization is the second-order analog of how one initializes states for 1^{st}-order forward-

mode AD with dual numbers.

Forward-Mode AD

The question then becomes how to correctly propagate these additional terms corresponding to first and second derivatives of each x_i through the differentiable program (in this example a neural network). For first derivatives, there are the canonical dual numbers which give a standard way to do this. However, for higher derivatives, we must lift all standard calculus rules to higher-order derivatives. For instance, the chain rule must be generalized using the Faa di Bruno formula [98], and the product rule must be generalized using the general Leibniz rule [99].

We next compute the values for x_3 and x_4. Because of the linearity of derivatives, both the first and second derivative terms for x_3 and x_4 are also linear combinations of the respective first and second derivative terms of x_1 and x_2. Hence $x_3[\{1\}] = w_{13}x_1[\{1\}] + w_{23}x_2[\{1\}]$, $x_3[\{1,2\}] = w_{13}x_1[\{1,2\}] + w_{23}x_2[\{1,2\}]$ and $x_4[\{2\}] = w_{14}x_1[\{2\}] + w_{24}x_2[\{2\}]$, $x_4[\{1,2\}] = w_{14}x_1[\{1,2\}] + w_{24}x_2[\{1,2\}]$.

Upon computing all real and derivative coefficients for x_3 and x_4, we then apply a *SoftPlus* activation to obtain x_5 and x_6, respectively. Computing the first derivative terms – $x_5[\{1\}]$, $x_5[\{2\}]$, $x_6[\{1\}]$, and $x_6[\{2\}]$ – is done virtually identically as with dual numbers. We compute the first derivative of the *SoftPlus* function which is the sigmoid function, $\sigma(x)$, and evaluate it at both $x_3[\emptyset]$ and $x_4[\emptyset]$, saving these results in unique intermediate variables, $dx_5[1]$ and $dx_6[1]$.

However, for first derivatives, the chain rule tells us that we need not only the derivatives of the sigmoidal function ($dx_5[1]$ and $dx_6[1]$) we are composing with, but also the first derivative of the inputs ($x_3[\{1\}]$ and $x_4[\{1\}]$), hence why we must also evaluate $x_5[\{1\}] = dx_5[1] \cdot x_3[\{1\}]$, $x_5[\{2\}] = dx_5[1] \cdot x_3[\{2\}]$ as well as $x_6[\{1\}] = dx_6[1] \cdot x_4[\{1\}]$ and $x_6[\{2\}] = dx_6[1] \cdot x_4[\{2\}]$.

Computing the higher-derivative terms for composition with the *SoftPlus* function is more involved. While computing the higher-derivative terms for affine combinations of variables is easy due to linearity, higher-derivatives of functional composition require Faa di Bruno's formula. In this example, we need to compute the value of the second derivative of the *SoftPlus* function, which is $\sigma(x) \cdot (1 - \sigma(x))$. Since we have already computed $\sigma(x_3[\emptyset])$ and $\sigma(x_4[\emptyset])$, we can reuse these results (respectively stored in $dx_5[1]$ and $dx_6[1]$), hence why the second derivatives are computed as $dx_5[2] = dx_5[1] \cdot (1 - dx_5[1])$ and $dx_6[2] = dx_6[1] \cdot (1 - dx_6[1])$. We can then use both $dx_5[1]$ and $dx_5[2]$ as well as $dx_6[1]$ and $dx_6[2]$ to compute $x_5[\{1,2\}]$ and $x_6[\{1,2\}]$. It is important to note that there is a data-dependence *across derivatives*: for instance, $dx_6[2]$ has data-dependence on $dx_6[1]$. This data-dependence

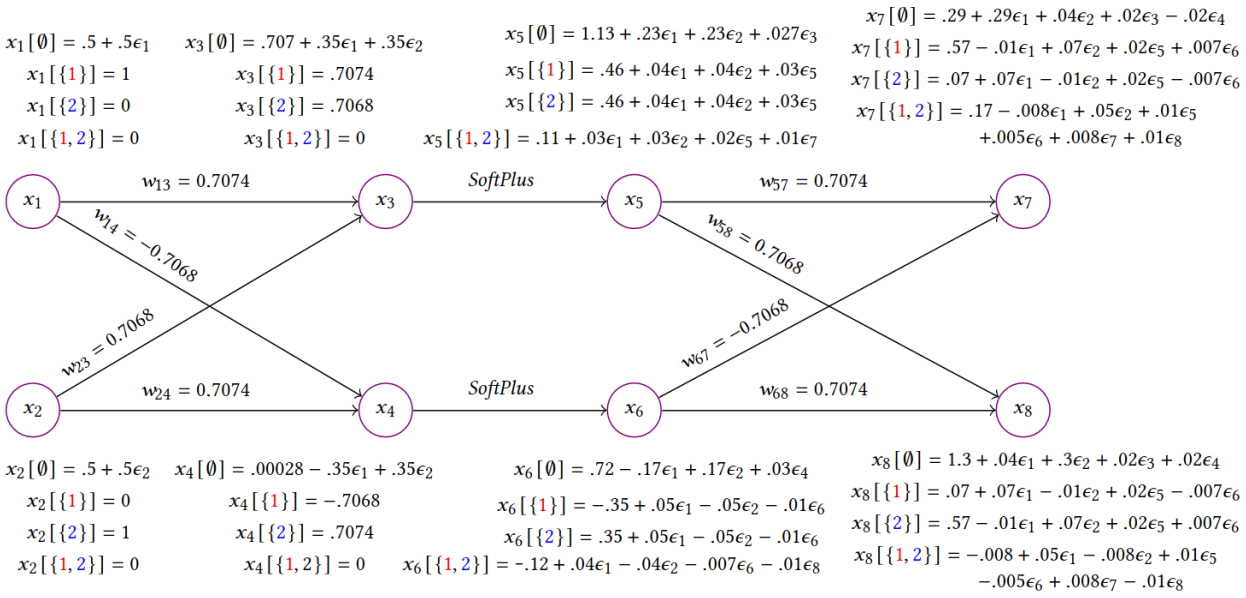


Figure 3.2: Abstract 2^{nd} order forward-mode AD evaluation

is a direct result of how we have set up our concrete AD semantics, as one could have naively had the transformed program recompute terms instead of producing a program that exposes data dependence between lower and higher-order derivatives as we do. We will later see in Section 3.7 that transforming the program using this technique improves the precision when performing abstract interpretation with a relational domain.

Lastly, we again apply an affine layer to get the final output of the network. As before, we take linear combinations of not just the "real" parts ($x_5[\emptyset]$ and $x_6[\emptyset]$) but also their derivative parts, since derivatives still follow linearity. The final outputs are now the derivatives with respect to the original inputs, x_1 and x_2. Thus, $x_7[\{1\}] = \frac{\partial x_7}{\partial x_1}\Big|_{(x_1^{in}, x_2^{in}) \in \mathbb{R}^2}$, $x_7[\{2\}] = \frac{\partial x_7}{\partial x_2}\Big|_{(x_1^{in}, x_2^{in}) \in \mathbb{R}^2}$ and $x_7[\{1,2\}] = \frac{\partial^2 x_7}{\partial x_1 \partial x_2}\Big|_{(x_1^{in}, x_2^{in}) \in \mathbb{R}^2}$, where the notation $\frac{\partial x_i}{\partial x_j, \ldots, \partial x_k}\Big|_{(x_1, \ldots, x_m) \in \mathbb{R}^m}$ denotes the partial derivative of some variable (x_i) with respect to other variables $(x_j, ..., x_k)$ evaluated at some concrete point $(x_1, ..., x_m) \in \mathbb{R}^m$. Likewise, we also know that $x_8[\{1\}] = \frac{\partial x_8}{\partial x_1}\Big|_{(x_1^{in}, x_2^{in}) \in \mathbb{R}^2}$, $x_8[\{2\}] = \frac{\partial x_8}{\partial x_2}\Big|_{(x_1^{in}, x_2^{in}) \in \mathbb{R}^2}$ and $x_8[\{1,2\}] = \frac{\partial^2 x_8}{\partial x_1 \partial x_2}\Big|_{(x_1^{in}, x_2^{in}) \in \mathbb{R}^2}$.

Abstract Interpretation

Having shown how the *concrete* semantics are interpreted and expose each derivative explicitly as a separate variable in memory (indexed by a set), we can think about how we might *abstractly* interpret these semantics to get formal bounds on all of these derivatives inside a region. Getting bounds on these derivatives is useful as (1) derivatives are canonically used for explaining the behavior (e.g. 1^{st}-order feature attribution or 2^{nd}-order interactions)

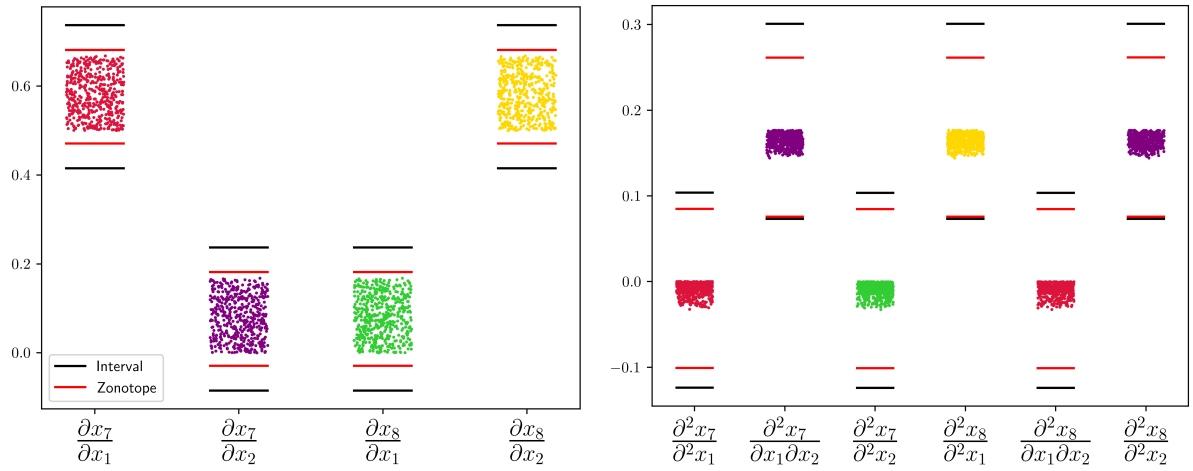
of a neural network and (2) prior work has shown that such explanations evaluated at scalar points are not robust [12], hence why recent work [100] has focused on computing *provably robust* explanations using interval abstractions. Abstractly interpreting the AD used to compute the derivatives for these explanations gives us these provable bounds.

For our example, we instantiate our construction with the zonotope abstract domain [101], which associates to each variable an affine form written as $c_0 + \sum_{i=1}^{K} c_i\epsilon_i$, where each noise term satisfies $\epsilon_i \in [-1, 1]$. Since multiple variables can share noise terms, dependencies across variables are captured; therefore, the zonotope domain is a *relational* domain.

For the purposes of this example, we want to compute provable bounds on the derivatives of all variables for the local region $(x_1^{in}, x_2^{in}) \in [0, 1] \times [0, 1]$. Since $x_1^{in} \in [0, 1]$, its affine form is $.5 + .5\epsilon_1$. Likewise since $x_2^{in} \in [0, 1]$, its affine form is $.5 + .5\epsilon_2$, noting that it has an independent noise term (ϵ_2 instead of ϵ_1) since the variables are not correlated. Even for the abstract analysis, we still set $x_1[\{1\}] = 1$ and $x_2[\{2\}] = 1$ if we wish to differentiate with respect to x_1 and x_2, hence why there are still constants (constants are represented exactly in the zonotope domain).

For AD, affine transformations on variables correspond to affine transformations on *all* of their derivatives. Thus, because we will be computing not just one, but 4 affine transformations per variable (one for each component), it is advantageous to have an abstract domain that is as precise as possible for affine transformations. The zonotope domain is *exact* for affine transformations, thus making it attractive for abstract interpretation of AD. For the "real" parts of x_3 and x_4 , we have $x_3[\emptyset] = .707 + .35\epsilon_1 + .35\epsilon_2$ and $x_4[\emptyset] = .00028 - 0.35\epsilon_1 + .35\epsilon_2$. Likewise, the first derivative terms of x_3 and x_4 are exactly an affine combination of their original values (0 and 1), giving $x_3[\{1\}] = .7074$, $x_3[\{2\}] = .7068$ and $x_4[\{1\}] = -.7068$, $x_4[\{2\}] = .7074$, respectively. We also note that the second derivative terms ($x_3[\{1, 2\}]$ and $x_4[\{1, 2\}]$) are still zero (which is represented exactly in the zonotope domain), as the second derivative of a linear function is necessarily zero.

Applying any non-linear function, such as a $\sigma(x)$ or $\mathit{SoftPlus}(x)$, to a zonotope or affine form is more challenging, as we must come up with an abstract transformer that employs a sound *linearization* of that function. Furthermore, for AD, we need a sound linearization for not just the activation function, but also for all of its derivatives. Hence, we need abstract transformers for $\mathit{SoftPlus}(x)$, $\sigma(x)$, and multiplication. In Section 3.6.2, we describe how to automatically construct zonotope transformers for functions like $\mathit{SoftPlus}(x)$ that other works [52, 93] cannot support, using the Chebyshev construction [102]. The core idea is to add new noise symbols for each application of a non-linear function. The resulting affine forms after applying the $\mathit{SoftPlus}(x)$ are shown above variable x_5 and below variable x_6 in Fig. 3.2. Since the $dx_i[1]$ and $dx_i[2]$ are merely intermediate variables for the computation



(a) Bounds on First Derivatives

(b) Bounds on Second Derivatives

Figure 3.3: Bounds on Jacobian and Hessian components. Black bounds were obtained via interval analysis; red bounds were obtained via zonotope analysis. The dots represent concretely evaluating the Jacobian/Hessian at scalar points sampled from the input region.

of each $x_i[\{1\}]$, $x_i[\{2\}]$, and $x_i[\{1,2\}]$, we omit them for simplicity. However, it is important to recall that because of how we set up the semantics earlier, there is a data dependence between $dx_i[1]$ and $dx_i[2]$. The zonotope domain will be able to leverage this dependence for improved precision.

We can now finally propagate the zonotope through the last affine layer. As mentioned, for the zonotope domain, this step is exact. The final affine forms (which collectively form the zonotope) for all derivatives are shown in Fig. 3.2. Furthermore, we can ultimately convert this output zonotope to interval bounds for each variable. Fig. 3.3a presents in red the bounds for all first derivatives computed from the zonotopes and likewise Fig. 3.3b shows in red the bounds on the second derivatives. We also show the result of performing the abstraction instantiated with the standard interval domain, denoted by the black bars in the respective plots. Lastly, we show the derivatives at randomly sampled points $(x_1, x_2) \in [0,1] \times [0,1]$ using only the concrete semantics (colored points). Both plots show that zonotopes are more precise for bounding both the first and second derivatives, and both abstractions soundly enclose the derivatives computed at scalar points. There is slightly more over-approximation for the bounds in Fig. 3.3b, as second derivatives require more computations, compounding the over-approximation inherent to the abstraction. For illustration simplicity, the example of Fig. 3.2 corresponds to a single pass of (abstract) forward-mode AD. However, to obtain *all* entries plotted in Figs. 3.3a and 3.3b, we need to rerun forward-mode AD for multiple passes. Theorem 3.2 describes how many separate passes of forward-mode AD are required.

By understanding bounds on the derivatives, we can interpret and explain the magnitude of the contribution of each variable (1^{st} derivatives) or the interactive combinations of two variables (2^{nd} derivatives) to the final output, and do this provably for an entire input region. We empirically evaluate this approach further in Section 3.7.2.

3.3 PRELIMINARIES

We now detail the necessary mathematical preliminaries to describe forward-mode automatic differentiation, particularly for higher derivatives.

3.3.1 Mathematical Definitions

As many of our operations involve working with sets we first detail our set-based notation. We let $\binom{n}{k}$ represent the scalar binomial coefficient $\frac{n!}{(n-k)!k!}$. We will let $\mathcal{P}(S)$ denote the *powerset* of a set S. Furthermore, we will write $|S|$ to denote the cardinality of S.

We will also write $\mathcal{P}_k(S)$ to denote the collection of all subsets of some set S that have a specific cardinality k, equivalently $\{A \in \mathcal{P}(S) \ : \ |A| = k\}$. For instance, $\mathcal{P}_1(\{1,2\}) = \{\{1\},\{2\}\}$. For *finite* sets of integers in a given range, we may write $\{1,...,D\}$, e.g. $\{1,...,4\} = \{1,2,3,4\}$. To denote the set of all possible partitions of a finite set of integers, we will write $Part(S)$.

3.3.2 Forward-Mode Automatic Differentiation

The most basic implementation of first-order forward-mode AD is operator overloading with dual numbers [103]. Intuitively, dual numbers are two-dimensional numbers that correspond to the value of a function at a point, and the value of the function's derivative at that point. Dual numbers are canonically defined as numbers of the form $a + b\epsilon$ with ϵ being an infinitesimal part, similar to the imaginary part of complex numbers. However, because our work later uses ϵ for denoting zonotope noise symbols, and because it is more intuitive when generalizing to higher derivatives, we present dual numbers simply as two-element tuples.

Definition 3.1. Dual numbers, are numbers of the form (a, b) where $a, b \in \mathbb{R}$ and for any real-valued differentiable function $f\colon \mathbb{R} \to \mathbb{R}$, we can lift the interpretation of f to dual numbers by defining $f\big((a,b)\big) = \big(f(a), f'(a) \cdot b\big)$.

One may also define all the arithmetic operations over dual numbers. For example for two dual numbers $x = (a_1, b_1)$ and $y = (a_2, b_2)$ one defines $x + y = (a_1 + a_2, b_1 + b_2)$, which

intuitively encodes the notion of linearity. Rules for multiplication and division are similar, implicitly encoding the product and quotient rules. Hence, given a set of primitive functions f_i whose derivatives f_i' we know analytically, we can propagate dual numbers through arbitrary compositions of these functions including ones involving arithmetic operations to compute first derivatives compositionally.

Higher-Order Derivatives. Just as forward-mode automatic differentiation can compute first derivatives using an alternate interpretation of functions and arithmetic, it can be extended to compute higher-order derivatives, as noted in [103]. However, higher-order derivatives are more challenging, because the formulas such as the product or chain rules must be lifted to their higher-order versions. For first derivatives there are 2 elements in the dual number tuple, hence one might expect that for D^{th} derivatives the generalization of a dual number will be a $\mathbb{R}^{2^D}$ tuple, which we will indeed show is the case. Intuitively, for a D^{th}-order derivative, each of the 2^D tuple elements represents a partial derivative with respect to a subset of the program variables. However, there is no loss of generality; indeed, we will show that this idea still allows one to differentiate with respect to the *same* variable multiple times.

This stems from the fact that in forward-mode AD, one is essentially propagating coefficients of a truncated D degree *multi-variate* Taylor polynomial, as the coefficients (without the integer factorial division) are precisely the values of the derivatives. We define this intuition formally.

Definition 3.2. For a variable x_o corresponding to some arithmetic function of variables x_1 through x_D, a truncated D-degree multi-variate Taylor polynomial is a 2^D length tuple of the form
$(x_o, \frac{\partial x_o}{\partial x_1}, ..., \frac{\partial x_o}{\partial x_D}, \frac{\partial^2 x_o}{\partial x_1 \partial x_2}, ..., \frac{\partial^D x_o}{\partial x_1 ... \partial x_D})$ containing the 0^{th} through D^{th}-order partial derivatives with respect to $x_1 ... x_D$. Furthermore, each element will not be symbolic but rather the result of evaluating that derivative at a given input point, hence the tuple is an element of $\mathbb{R}^{2^D}$.

For a fixed D, there will be $\binom{D}{0}$ zeroth derivatives (the single "real" part x_o), $\binom{D}{1}$ first derivatives ($\frac{\partial}{\partial x_1}$ through $\frac{\partial}{\partial x_D}$) all the way up to $\binom{D}{D}$ derivatives of order D – hence we can only compute one D^{th}-order derivative at a time, which is why the polynomial tuple is considered truncated. Thus, for any $d \in \{0, 1, ..., D\}$, there will be *exactly* $\binom{D}{d}$ derivatives of order d stored in the tuple. Furthermore, these derivatives will only be with respect to a predefined set of input variables.

Encoding Truncated Taylor Polynomials Since each entry of a truncated D-degree Taylor polynomial is a derivative evaluated at a concrete real value, we will ultimately reduce reasoning about complicated polynomials to reasoning about their individual coefficients. However, to "depackage" a tuple of length 2^D where each element has a specific semantic meaning (some particular partial derivative), we need an indexing scheme to access the individual elements. This will form a core development of our concrete semantics and will later allow us to capture data dependencies *between* program variables (corresponding to derivatives) when abstracting the semantics.

Variable Indexing As pointed out by [104], maintaining clarity in how one indexes program variables corresponding to the various derivatives in AD is difficult. Therefore, a part of our contribution is to develop a novel set-based indexing scheme, generalizing existing higher-order AD implementations [105]. For a D^{th}-order truncated Taylor polynomial tuple, each of the 2^D elements corresponds to a particular element of the power-set $\mathcal{P}(\{1, ..., D\})$. Furthermore, that element of the power set is *exactly* the set of variables the partial derivative is with respect to. For instance, the empty set $\emptyset$ corresponds to the real part (e.g. x_o) since it is the partial derivative with respect to *no* variables. Likewise, the singleton set $\{1\}$ corresponds to $\frac{\partial}{\partial x_1}$ and the full set $\{1, ..., D\}$ corresponds to the D^{th} derivative $\frac{\partial^D x_o}{\partial x_1 ... \partial x_D}$. Hence, this indexing scheme respects the semantic meaning of the entries. Lastly, while the natural idea is to associate a unique program variable to each number in the indexing set, we can also associate the *same* program variable to multiple numbers of the set. Hence, we could associate a variable x_i to both 1 and 2, meaning the set $\{1, 2\}$ corresponds to $\frac{\partial^2}{\partial x_i \partial x_i}$. In these cases, some of the tuple elements become redundant copies of the same derivative. More details on input state initialization are in Section 3.4.4.

While extremely helpful for making the formalism and proofs simple, this indexing scheme is not the most space efficient. The inefficiency stems from storing duplicate entries, thus future work might improve the storage cost of tracking all the higher derivatives.

Example 3.1. For the following truncated degree 2 Taylor polynomial, where x_1 is associated to 1 and x_2 is associated to 2, given by: $T = (x_o, \frac{\partial x_o}{\partial x_1}, \frac{\partial x_o}{\partial x_2}, \frac{\partial^2 x_o}{\partial x_1 \partial x_2}) \in \mathbb{R}^{2^2}$ we have $T[\emptyset] = x_o$, $T[\{1\}] = \frac{\partial x_o}{\partial x_1}$, $T[\{2\}] = \frac{\partial x_o}{\partial x_2}$ and $T[\{1, 2\}] = \frac{\partial^2 x_o}{\partial x_1 \partial x_2}$.

$$
\begin{array}{rcl}
P & ::= & P_1; P_2 \mid \mathtt{x}_i = Expr \\
Expr & ::= & \mathtt{x}_j + \mathtt{x}_k \mid \mathtt{x}_j - \mathtt{x}_k \mid \mathtt{x}_j * \mathtt{x}_k \\
& \mid & 1/\mathtt{x}_j \mid \log(\mathtt{x}_j) \mid \exp(\mathtt{x}_j) \\
& \mid & \text{SoftPlus}_a(\mathtt{x}_j) \mid \sigma_a(\mathtt{x}_j) \mid c \in \mathbb{R}
\end{array}
$$

Figure 3.4: Differentiable Function Syntax

3.4 LANGUAGE SYNTAX AND SEMANTICS

3.4.1 Syntax

Figure 3.4 presents the syntax of the language. Our language is imperative and syntactically supports arithmetic operations and differentiable function primitives. We will denote the variables in the original program's syntax (e.g. $\mathtt{x}_1$,...,$\mathtt{x}_k$) as *syntactic variables*, or just *SynVars*. However, because AD needs to instrument the program to also track additional variables corresponding to derivatives, we will later need to distinguish these variables from the program variables that actually store the derivatives. Many standard functions in machine learning can be implemented with only these primitives: for instance, we can encode $tanh(x) := 2\sigma_2(x) - 1$. Lastly, we detail in Def. 3.3 the syntactic restrictions on programs needed for the resulting derivatives to be correctly computed.

Definition 3.3. A differentiable program P is *well-formed* if all syntactic variables are either input variables, denoted as x_i^{in}, or accessed only after they have been defined. For simplicity, we also require the program be in SSA form.

Example 3.2. The differentiable program P with two input variables x_1^{in}, x_2^{in} given by $x_3 = x_1^{in} + x_2^{in}; x_4 = \exp(x_3);$ is well-formed while the program $P' \triangleq x_3 = x_1^{in} + x_2^{in}; x_4 = \exp(x_5); x_5 = 3x_3;$ is not, since x_5 is used before being defined.

One can translate a pure mathematical function that uses only the differentiable primitives in our syntax into a well-formed program using an ANF conversion [72].

3.4.2 Concrete AD Meta-Semantics

We now define a meta-semantics, which is a construction that takes as input a maximum derivative order D to be computed and gives a concrete forward-mode AD semantics for computing derivatives up to that order using truncated Taylor polynomials. While we could map each syntactic variable x_i in the original program to a 2^D length real-valued array storing its truncated Taylor polynomial, we instead syntactically desugar this mapping so that

each individual coefficient in the Taylor polynomial is identified by its own unique variable. Our novel set-based indexing scheme (Section 3.3.2) is used to distinguish these individual variables. We call this variable set the augmented variables, or $AugVars$, to distinguish it from $SynVars$, and we will access these augmented variables using the indexing scheme. The added benefit is that by semantically exposing each entry as an intuitively indexed, augmented variable, we can (a) reduce reasoning about complicated mathematical objects (Taylor polynomials) to reasoning about standard real-valued programs and (b) we can capture correlations *between* individual variables when using a relational abstract domain.

Hence, for forward-mode AD, our concrete states σ will map $\sigma\colon AugVars \to \mathbb{R}$. However this means that if we have m syntactic variables in the program source code (e.g. $x_1, ..., x_m$), then $AugVars$ will now have $\mathcal{O}(m \cdot 2^D)$ augmented variables, e.g. $x_1[\emptyset], ..., x_1[\{1, ..., D\}]$ through $x_m[\emptyset]$, ..., $x_m[\{1, ..., D\}]$. Having now defined a concrete AD program state σ (which is what the semantics will ultimately compute), we can formally define the concrete domain upon which the meta-semantics will be built.

Definition 3.4. The concrete domain $\mathcal{D}$ is $\mathcal{P}(AugVars \to \mathbb{R})$ but this set is isomorphic to $\mathcal{P}(\mathbb{R}^{|AugVars|})$, hence we will also say that $\mathcal{D} = \mathcal{P}(\mathbb{R}^{|AugVars|})$.

Intuitively, for a concrete state σ we can just enumerate all $\mathcal{O}(m \cdot 2^D)$ augmented variables into a single state vector with their corresponding Taylor coefficient value, thus this state vector σ is equivalently an element of $\mathbb{R}^{|AugVars|}$. Because our AD states are now standard mappings of (augmented) variables to real numbers, we can construct an interpreter for $[\![\cdot]\!]_D$ using only a standard interpreter for real-valued imperative programs $[\![\cdot]\!]\colon (P, \mathcal{D}) \to \mathcal{D}$. We now formally define $[\![\cdot]\!]_D$ in terms of the base interpreter $[\![\cdot]\!]$.

Definition 3.5. The base interpreter $[\![\cdot]\!]\colon (P, \mathcal{D}) \to \mathcal{D}$ for assignment statements, and $[\![Expr]\!]\colon (Expr, \mathcal{D}) \to \mathbb{R}$ for arithmetic sub-expressions, is given by the rules of Fig. 3.5.

If the value of $\sigma[x_i]$ is not part of a function's input domain, e.g. if $\sigma[x_i] = 0$ when computing $1/(\sigma[x_i])$, the evaluation of $[\![\cdot]\!]$ returns $\bot$. Likewise, any arithmetic operation with $\bot$ returns $\bot$. We can now use $[\![\cdot]\!]$ as a *subroutine* to build an interpreter for performing concrete D^{th} degree forward-mode AD. Lastly, while $[\![\cdot]\!]$ is defined in Fig. 3.5 only for elementary expressions (binary arithmetic and unary functions) we will still notationally write $[\![Expr]\!]$ when $Expr$ is a non-elementary expression, as these are desugared to sequences of elementary operations by merely adding more augmented variables for storing all intermediate sub-expressions.

Definition 3.6. The concrete meta-semantics for performing forward-mode AD are a parametric semantics given by $[\![\cdot]\!]_D\colon (P, \mathcal{D}) \to \mathcal{D}$, where the parameter $D \in \mathbb{N}_{>0}$ is the order of

$$[\![x_i = Expr]\!](\sigma) = \sigma[x_i \leftarrow [\![Expr]\!](\sigma)] \qquad [\![P_1; P_2]\!](\sigma) = [\![P_2]\!]([\![P_1]\!](\sigma))$$
$$[\![x_j + x_k]\!](\sigma) = \sigma[x_j] + \sigma[x_k] \qquad [\![x_j - x_k]\!](\sigma) = \sigma[x_j] - \sigma[x_k]$$
$$[\![x_j * x_k]\!](\sigma) = \sigma[x_j] \cdot \sigma[x_k] \qquad [\![exp(x_k)]\!](\sigma) = exp(\sigma[x_k])$$
$$[\![\sigma_a(x_k)]\!](\sigma) = \sigma_a(\sigma[x_k]) \qquad [\![SoftPlus_a(x_k)]\!](\sigma) = SoftPlus_a(\sigma[x_k])$$
$$[\![log(x_k)]\!](\sigma) = \sigma[x_k] > 0 \ ?\ log(\sigma[x_k]) \ :\ \perp \qquad [\![1/x_k]\!](\sigma) = \sigma[x_k] \neq 0 \ ?\ 1/\sigma[x_k] \ :\ \perp$$
$$[\![c]\!](\sigma) = c$$

Figure 3.5: Semantic rules for the base statement and expression interpreter $[\![\cdot]\!]$ ($\perp$ is the error state).

the highest derivative the AD semantics can compute. Each statement's rule is shown in the following section.

We now give the formal rules for evaluating $[\![\cdot]\!]_D$. We immediately see why the concrete meta-semantics are parametric in D – choosing a D governs how many times $[\![\cdot]\!]$ will be called. This is why we refer to these as a *meta*-semantics – choosing a different D instantiates a different concrete semantics for each statement and expression. We also mention that in all cases, despite the complex form of the right-hand side expressions used in assignments, the entire expression can always be unpacked into the language primitives of Fig. 3.4, even if it requires introducing additional intermediate variables to store intermediate results. Lastly, this construction is defined imperatively, as these are a *state-transformer* semantics which update σ after each application of $[\![\cdot]\!]$.

Addition The rule for addition is simple, as higher-order derivatives also follow linearity. The meta-semantics are given, where for all coefficients $x_i[S]$ in the truncated Taylor polynomial, we compute that coefficient by adding the corresponding coefficients of syntactic variables x_j and x_k.

$$[\![x_i = x_j + x_k]\!]_D(\sigma) \triangleq \textbf{for}\ \ S\ \in\ \mathcal{P}(\{1, ..., D\}):$$
$$\sigma = [\![x_i[S] = x_j[S] + x_k[S]]\!](\sigma)$$
$$\textbf{return}\ \ \sigma$$

Figure 3.6: Concrete Meta-Semantics of Higher-Order Differentiation of Addition

Subtraction Subtraction follows nearly identically to addition, with the only difference being the state gets updated via $\sigma = [\![x_i[S] = x_j[S] - x_k[S]]\!](\sigma)$ instead of $\sigma = [\![x_i[S] = x_j[S] + x_k[S]]\!](\sigma)$.

Multiplication The rule for multiplication follows directly from the Generalized Leibniz formula ($S \setminus P$ is set subtraction of P from S).

$$
\begin{array}{l}
[\![x_i = x_j * x_k]\!]_D(\sigma) \triangleq \ \textbf{for}\ \ d\ \in\ \{0, ..., D\}: \\
\qquad\qquad\qquad\qquad\quad \textbf{for}\ \ S\ \in\ \mathcal{P}_d(\{1, ..., D\}): \\
\qquad\qquad\qquad\qquad\qquad\quad \sigma = [\![x_i[S] = \textstyle\sum_{P \in \mathcal{P}(S)} x_j[P] \cdot x_k[S \setminus P]]\!](\sigma) \\
\qquad\qquad\qquad\quad \textbf{return}\ \ \sigma
\end{array}
$$

Figure 3.7: Concrete Meta-Semantics of Higher-Order Differentiation of Multiplication

While the summation term, $\sum_{P \in \mathcal{P}(S)} x_j[P] \cdot x_k[S \setminus P]$, is given in a general form, the only primitive operations involved are addition and multiplication, hence when instantiating a concrete instance of these semantics for a fixed derivative order $D \in \mathbb{N}_{>0}$ we can unroll the entire summation into primitive additions and multiplications of existing variables, and thus we can still evaluate each assignment in the **for** loop with the base interpreter of Fig. 3.5.

Constants Constants are simple, as all their higher-order derivatives are zero. Hence, for any non-empty set S, the coefficient in x_i's Taylor polynomial indexed by S will necessarily be zero.

$$
\begin{array}{l}
[\![x_i = c]\!]_D(\sigma) \triangleq \ \textbf{for}\ \ S\ \in\ \mathcal{P}(\{1, ..., D\}): \\
\qquad\qquad\quad \textbf{if}\ \ S = \emptyset: \\
\qquad\qquad\qquad\quad \sigma = [\![x_i[\emptyset] = c]\!](\sigma) \\
\qquad\qquad\quad \textbf{else}: \\
\qquad\qquad\qquad\quad \sigma = [\![x_i[S] = 0]\!](\sigma) \\
\qquad\qquad \textbf{return}\ \ \sigma
\end{array}
$$

Figure 3.8: Concrete Meta-Semantics of Higher-Order Differentiation of Constants

Unary Functions For compositions with unary functions $f\colon \mathbb{R} \to \mathbb{R}$, we will need to use Faa di Bruno's formula [98] as a generalization of the Chain rule to compute higher-order derivatives. The multivariate Faa di Bruno formula for a D^{th}-order derivative of a composite function is given below in Def. 3.7:

Definition 3.7. ([98]) Let $f\colon \mathbb{R} \to \mathbb{R}$ and $y = g(x_1, ..., x_D)$ where $g\colon \mathbb{R}^D \to \mathbb{R}$. Then

$$\frac{\partial^D}{\partial x_1 ... \partial x_D} f(y) = \sum_{P \in Part(\{1,...,D\})} f^{|P|}(y) \cdot \prod_{B \in P} \frac{\partial^{|B|} y}{\prod_{j \in B} \partial x_j}$$

where $Part(\{1, ..., D\})$ returns the set of partitions of $\{1, ..., D\}$ and $|P|$ returns the cardinality of set P. Intuitively, we must compute all derivatives up to order D of the function f, as well as all possible partial derivatives of order less than D of the function $y = g(x_1, ..., x_D)$.

We will see that for the subsequent unary function semantic rules, there is a direct correspondence between each term in Def. 3.7 and the respective rule; in particular, each possible partial derivative needed by the Faa di Bruno formula will be stored in a separate variable. Furthermore, as with the multiplication rule, though the notation is condensed, in the subsequent semantic rules, all primitive arithmetic operations are just additions (from $\sum_{P \in Part(S)}$), multiplications (from $\prod_{B \in P}$), variable lookups (e.g. due to $x_j[B]$) or elementary function evaluations of f and its analytically known derivatives. Further, when fixing a particular order of derivative, D, all possible partitions of all possible sets (e.g. $P \in Part(S)$) can be precomputed and enumerated (as they are finite), as can $|P|$. Hence, these seemingly complicated expressions still reduce to elementary operations.

The core requirement is that we are able to *analytically* compute the first through D^{th} derivatives of f evaluated at the real part of x_j (itself stored in $x_j[\emptyset]$). Thus, we restrict f to only elementary functions whose derivatives of all orders can be known *analytically* as combinations of the other elementary functions in the language, hence why the language is restricted as shown in Fig. 3.4. Each of the D derivatives of f will be stored in a unique D element array: $dx_i[0]$ through $dx_i[D]$ within σ, but as with the Taylor coefficients, each of these elements will be exposed as its own variable. Lastly, this is the only part that changes when defining the meta-semantics of different functions.

$$
\begin{aligned}
&[\![x_i = f(x_j)]\!]_D(\sigma) \triangleq\ \sigma = [\![x_i[\emptyset] = f(x_j[\emptyset])]\!](\sigma) \\
&\qquad \textbf{for}\ \ d\ \in\ \{1, ..., D\}: \\
&\qquad\quad \sigma = [\![dx_i[d] = \tfrac{d^d}{dx^d} f(x)|_{x=x_j[\emptyset]}]\!](\sigma) \\
&\qquad\quad \textbf{for}\ \ S\ \in\ \mathcal{P}_d(\{1, ..., D\}): \\
&\qquad\qquad \sigma = [\![x_i[S] = \textstyle\sum_{P \in Part(S)} dx_i[|P|] \cdot \prod_{B \in P} x_j[B]]\!](\sigma) \\
&\qquad \textbf{return}\ \ \sigma
\end{aligned}
$$

Figure 3.9: Concrete Meta-Semantics for Higher-Order Differentiation of general $f(x)$

Division Since Division is composition with the function $f(x) = \frac{1}{x}$, we can encode it using the above technique. Furthermore, the derivatives of $\frac{1}{x}$ have a known elementary form that can be encoded using only primitive arithmetic operations (multiplications and divisions), which specifically is $\frac{d^d}{dx^d}\frac{1}{x} = (-1)^d \frac{d!}{x^{d+1}}$. However $(-1)^d \frac{d!}{x^{d+1}} = \frac{-d}{x} \cdot \frac{(-1)^{d-1}(d-1)!}{x^d} = \frac{-d}{x} \cdot \frac{d^{d-1}}{dx^{d-1}}\frac{1}{x}$. Hence, we have that $\frac{d^d}{dx^d}\frac{1}{x} = \frac{-d}{x} \cdot \frac{d^{d-1}}{dx^{d-1}}\frac{1}{x}$, thus we can use the recurrence $dx_i[d] = -d \cdot x_i[\emptyset] \cdot dx_i[d-1]$ which allows us to capture data dependencies across derivative terms.

$$
\begin{aligned}
&[\![x_i = 1/x_j]\!]_D(\sigma) \triangleq \quad [\![x_i[\emptyset] = 1/(x_j[\emptyset])]\!](\sigma) \\
&\qquad\qquad \sigma = [\![dx_i[1] = -x_i[\emptyset] \cdot x_i[\emptyset]]\!](\sigma) \\
&\qquad\qquad \textbf{for } \; d \; \in \; \{2, ..., D\}: \\
&\qquad\qquad\qquad\qquad \sigma = [\![dx_i[d] = -d \cdot x_i[\emptyset] \cdot dx_i[d-1]]\!](\sigma) \\
&\qquad\qquad\qquad\qquad \textbf{for } \; S \; \in \; \mathcal{P}_d(\{1, ..., D\}): \\
&\qquad\qquad\qquad\qquad\qquad\qquad \sigma = [\![x_i[S] = \textstyle\sum_{P \in Part(S)} dx_i[|P|] \cdot \prod_{B \in P} x_j[B]]\!](\sigma) \\
&\qquad\qquad \textbf{return } \; \sigma
\end{aligned}
$$

Figure 3.10: Concrete Meta-Semantics of Higher-Order Differentiation of $\frac{1}{x}$

Exp and Log As before, for *log* and *exp* the only difference will be the computation of the D-derivatives array, which for the exponential function is given as $[\![dx_i[d] = x_i[\emptyset])]\!](\sigma)$ for each $d \in \{1, ..., D\}$ since $\frac{d^d}{dx^d} exp(x) = exp(x)$ which is already stored in $x_i[\emptyset]$. For the *log* function, the first derivative is just $\frac{1}{x}$, hence we can reuse the functional form from the division rule to compute all further derivatives.

$$
\begin{aligned}
&[\![x_i = exp(x_j)]\!]_D(\sigma) \triangleq \; [\![x_i[\emptyset] = exp(x_j[\emptyset])]\!](\sigma) \\
&\qquad\qquad\qquad \textbf{for } \; d \; \in \; \{1, ..., D\}: \\
&\qquad\qquad\qquad\qquad \sigma = [\![dx_i[d] = x_i[\emptyset]]\!](\sigma) \\
&\qquad\qquad\qquad\qquad \textbf{for } \; S \; \in \; \mathcal{P}_d(\{1, ..., D\}): \\
&\qquad\qquad\qquad\qquad\qquad\qquad \sigma = [\![x_i[S] = \textstyle\sum_{P \in Part(S)} dx_i[|P|] \cdot \prod_{B \in P} x_j[B]]\!](\sigma) \\
&\qquad\qquad\qquad \textbf{return } \; \sigma
\end{aligned}
$$

Figure 3.11: Concrete Meta-Semantics of Higher-Order Differentiation of $\exp(x)$

Sigmoid and SoftPlus The sigmoid function $\sigma_a(x) = \frac{1}{1+e^{-ax}}$ is such that all of its derivatives can be given in terms of an elementary combination of other sigmoid functions. The d^{th} derivative is given as $\frac{d^d}{dx^d}\sigma_a(x) = \sum_{k=0}^{d}(-1)^{d+k}(k!)(S_{d,k})(a^d)\sigma_a(x)(1-\sigma_a(x))^k$, where $S_{d,k}$ are Stirling numbers of the second kind (which are just constants). Hence for any fixed $d \in \mathbb{N}$, we can combinatorially enumerate all the terms in order to have an analytical closed-form expression for the d^{th} derivative that uses only elementary operations (sums, products and constants) and the sigmoid function itself. This means that we need only compute the sigmoid *once*, which we do by having the base interpreter evaluate $[\![x_i[\emptyset] = \sigma_a(x_j[\emptyset])]\!](\sigma)$, after which we can fetch the variable $x_i[\emptyset]$ when computing each $dx_i[d]$ augmented variable. This helps us to capture the dependency across derivatives.

Since the sigmoid function is just the first derivative of the *SoftPlus* function, the d^{th} derivative of a *SoftPlus* is just the $d-1^{th}$ derivative of a sigmoid. Hence we can leverage the exact same functional form when computing $dx_i[d]$ for the *SoftPlus* function.

$$
\begin{aligned}
&[\![x_i = \sigma_a(x_j)]\!]_D(\sigma) \triangleq\ [\![x_i[\emptyset] = \sigma_a(x_j[\emptyset])]\!](\sigma) \\
&\qquad \textbf{for}\ \ d\ \in\ \{1, ..., D\}: \\
&\qquad\quad \sigma = [\![dx_i[d] = \textstyle\sum_{k=0}^{d}(-1)^{d+k}(k!)(S_{d,k})a^d x_i[\emptyset](1 - x_i[\emptyset])^k]\!](\sigma) \\
&\qquad\quad \textbf{for}\ \ S\ \in\ \mathcal{P}_d(\{1, ..., D\}): \\
&\qquad\qquad \sigma = [\![x_i[S] = \textstyle\sum_{P \in Part(S)} dx_i[|P|] \cdot \prod_{B \in P} x_j[B]]\!](\sigma) \\
&\qquad \textbf{return}\ \ \sigma
\end{aligned}
$$

Figure 3.12: Concrete Meta-Semantics of Higher-Order Differentiation of $\sigma_a(x) = \frac{1}{1+e^{-ax}}$

$$
\begin{aligned}
&[\![x_i = \mathit{SoftPlus}_a(x_j)]\!]_D(\sigma) \triangleq\ [\![x_i[\emptyset] = \mathit{SoftPlus}_a(x_j[\emptyset])]\!](\sigma) \\
&\qquad \sigma = [\![dx_i[1] = \sigma_a(x_j[\emptyset])]\!](\sigma) \\
&\qquad \textbf{for}\ \ d\ \in\ \{2, ..., D\}: \\
&\qquad\quad \sigma = [\![dx_i[d] = \textstyle\sum_{k=0}^{d}(-1)^{d+k}(k!)(S_{d,k})a^d dx_i[1]](1 - dx_i[1])^k]\!](\sigma) \\
&\qquad\quad \textbf{for}\ \ S\ \in\ \mathcal{P}_d(\{1, ..., D\}): \\
&\qquad\qquad \sigma = [\![x_i[S] = \textstyle\sum_{P \in Part(S)} dx_i[|P|] \cdot \prod_{B \in P} x_j[B]]\!](\sigma) \\
&\qquad \textbf{return}\ \ \sigma
\end{aligned}
$$

Figure 3.13: Concrete Meta-Semantics of Higher-Order Differentiation of $\mathit{SoftPlus}_a(x) = \frac{1}{a}\log(1 + e^{ax})$

Sequencing Having defined the construction to produce a semantics for computing the D^{th} derivative for individual assignment statements, we now note how sequencing of multiple statements works. We also note that the simplicity in the sequencing rule stems from the fact that programs are assumed to be in SSA form, hence no program variable will be overwritten.

$$
[\![P_1; P_2]\!]_D(\sigma) \triangleq [\![P_2]\!]_D\big([\![P_1]\!]_D(\sigma)\big)
$$

Figure 3.14: Concrete Meta-Semantics of Higher-Order Differentiation of Sequencing

Remark 3.1. When $D = 1$, $[\![\cdot]\!]_D$ computes exactly the dual numbers [5] and when $D = 2$, $[\![\cdot]\!]_D$ computes exactly the hyper-dual numbers [105].

3.4.3 Precision Enhancement

A key contribution of our concrete semantics is to ensure that the transformed program which computes all derivatives is generated in such a way that the later abstract interpretation will be precise. For AD, the computation of derivatives involves heavy reuse of common sub-expressions, hence by sharing these sub-expressions across multiple variables, the data dependence between different derivatives can be made explicit. For instance, in the case of

the *exp* function, all derivatives, $dx_i[d]$, will be computed as $[\![dx_i[d] = x_i[\emptyset]]\!](\sigma)$ where $x_i[\emptyset]$ already stores the result of computing $exp(x_j)$. Likewise, one can see similar data dependencies made explicit in the rules for $\frac{1}{x}$ and the σ_a function. While this will not change the execution result of the concrete semantics, it *does* substantially improve the precision of the abstract interpretation, as will be seen in Section 3.7.

3.4.4 Correctness

We now formally state the correctness as well as how one initializes the state to compute derivatives with respect to specific variables.

Input Variables We must first decide which input variables we wish to differentiate with respect to. However, we first note the following important remark that provides a limitation on how many input variables we can differentiate with respect to in a single forward execution $[\![P]\!]_D(\sigma)$.

Remark 3.2. For any set of syntactic input variables x_i^{in},..., x_j^{in} when evaluating $[\![P]\!]_D(\sigma)$, we can only differentiate with respect to up to D distinct syntactic variables.

This means that if the program P has more syntactic input variables than the maximum derivative order D, some variables will necessarily be treated as constants (meaning not differentiated). Additionally, if we differentiate with respect to the same syntactic variable more than once (e.g. to compute $\frac{\partial^2}{\partial x_i \partial x_i}$) the number of distinct variables we will be differentiating with respect to will be strictly less than D, again meaning some input variables will be treated as constants. However, as we will see, if one wishes to compute more derivatives, one may always rerun $[\![\cdot]\!]_D$ with different input variables initialized to obtain their derivatives.

Input State After deciding which of the (up to D) syntactic input variables we wish to differentiate with respect to, we must properly initialize each of those syntactic input variables' corresponding augmented variables in the input state, as AD only produces correct derivatives if input states are initialized correctly [5]. We generalize this idea to arbitrary order derivatives, as described below.

Definition 3.8. For $[\![\cdot]\!]_D$, an initial state σ is valid if for each singleton set $S \in \mathcal{P}_1(\{1, ..., D\})$, exactly one augmented variable satisfies $x_i^{in}[S] = 1$ while all others satisfy $x_j^{in}[S] = 0$. Furthermore, for each $S \in \mathcal{P}_k(\{1, ..., D\})$ where $k \geq 2$, all augmented variables satisfy $x_i^{in}[S] = 0$.

For different singleton sets $S, S' \in \mathcal{P}_1(\{1, ..., D\})$, where $S \neq S'$, the same input variable x_i^{in} can satisfy both $x_i^{in}[S] = 1$ and $x_i^{in}[S'] = 1$. Setting two different augmented variables that are both associated with the same syntactic variable to 1 allows us to differentiate with respect to the same variable twice (e.g. $\frac{\partial^2}{\partial x_i^{in} \partial x_i^{in}}$) instead of only being able to differentiate with respect to different variables. We now illustrate an example of a valid initial state.

Example 3.3. For $D = 2$ and $P \triangleq x_3 = x_1^{in} + x_2^{in}$, and state σ given as $\sigma[x_1^{in}[\emptyset]] = 2$, $\sigma[x_1^{in}[\{1\}]] = 1$, $\sigma[x_1^{in}[\{2\}]] = 0$, $\sigma[x_1^{in}[\{1,2\}]] = 0$ and $\sigma[x_2^{in}[\emptyset]] = 3.5$, $\sigma[x_2^{in}[\{1\}]] = 0$, $\sigma[x_1^{in}[\{2\}]] = 1$, $\sigma[x_2^{in}[\{1,2\}]] = 0$, then we have that σ is a valid input state to $[\![P]\!]_D$ that can be used to compute $\frac{\partial^2 x_3}{\partial x_1^{in} \partial x_2^{in}}$ at the point $(2, 3.5)$. However, if instead $\sigma[x_2^{in}[\{1\}]] \neq 0$ or if $\sigma[x_2^{in}[\{1,2\}]] \neq 0$ then σ would no longer be a valid initial state.

We next detail a lemma that says a correctly initialized input state computes valid derivatives of the input variables. This lemma also relates the syntactic variables of the program to the augmented variables computed by the interpreter and serves as the base case of our correctness theorem.

Lemma 3.1. Let σ be a valid input state. Then for any syntactic input variable x_i^{in} and any non-empty set $S \in \mathcal{P}(D) \setminus \emptyset$, we have that

$$\sigma[x_i^{in}[S]] = \frac{\partial^{|S|} x_i^{in}}{\prod\limits_{j \in Active(S)} \partial x_j^{in}}\Big|_{(x_1^{in}[\emptyset], ..., x_m^{in}[\emptyset]) \in \mathbb{R}^m}$$

where $Active(S)$ is the list of all input variables x_j^{in} satisfying $x_j^{in}[\{s_j\}] = 1$ for some $s_j \in S$.

Proof. Let $S = \{s_j\}$ where $x_j^{in} \in Active(\{s_j\})$. Then $\frac{\partial^{|S|} x_j^{in}}{\prod\limits_{j \in Active(S)} \partial x_j^{in}} = \frac{\partial x_j^{in}}{\partial x_j^{in}} = 1$ since the derivative of a variable with respect to itself is 1 and since $x_j^{in}[\{s_j\}] = 1$ due to the requirement of a valid input state, then $x_j^{in}[\{s_j\}] = \frac{\partial^{|S|} x_j^{in}}{\prod\limits_{j \in Active(S)} \partial x_j^{in}}$.

For $x_k^{in} \notin Active(\{s_j\})$ then $\frac{\partial^{|S|} x_k^{in}}{\prod\limits_{j \in Active(S)} \partial x_j^{in}} = \frac{\partial x_k^{in}}{\partial x_j^{in}} = 0$ since x_k is not a function of x_j since they are both primitive input variables. But $x_k^{in}[\{s_j\}] = 0$ due to the requirement of a valid input state only having one variable satisfy $x_j^{in}[\{s_j\}] = 1$. Hence $x_k^{in}[\{s_j\}] = \frac{\partial^{|S|} x_k^{in}}{\prod\limits_{j \in Active(S)} \partial x_j^{in}}$.

For any S where $|S| \geq 2$ then due to the requirement of a valid input state $x_i^{in}[S] = 0$ for any i. However any higher derivative of x_i will always be zero, thus $x_i^{in}[S] = 0 = \frac{\partial^{|S|} x_i^{in}}{\prod\limits_{j \in Active(S)} \partial x_j^{in}}$. Hence in all cases the equality holds.

QED.

Computing Derivatives Having established that a valid input state stores derivatives correctly, the idea is to now show that *after* executing each statement, the derivatives are *still* correct.

Theorem 3.1. Let σ be a valid input state and P be a well-formed program, and let $\sigma' = [\![P]\!]_D(\sigma)$. Then for any variable x_i in σ' and any non-empty set $S \in \mathcal{P}(D) \setminus \emptyset$, we have that

$$\sigma'[x_i[S]] = \frac{\partial^{|S|} x_i}{\prod\limits_{j \in Active(S)} \partial x_j^{in}}\Big|_{(x_1^{in}[\emptyset],\ldots,x_m^{in}[\emptyset]) \in \mathbb{R}^m}$$

Proof. We now detail the main cases:

- *Constants.* For any non-empty set S, and constant $c \in \mathbb{R}$, $\frac{\partial^{|S|} c}{\prod\limits_{j \in Active(S)} \partial x_j^{in}}\Big|_{(x_1^{in}[\emptyset],\ldots,x_m^{in}[\emptyset]) \in \mathbb{R}^m} = 0$. However, for any non-empty S, the interpreter always assigns 0 to $\sigma'[x_i[S]]$.
- *Addition.* As the current state σ has been correctly initialized, before executing $[\![x_i = x_j + x_k]\!]_D(\sigma)$ we inductively assume (by Lemma 3.1) that for each and every S, $\sigma[x_j[S]] = \frac{\partial^{|S|} x_j}{\prod\limits_{l \in Active(S)} \partial x_l^{in}}\Big|_{(x_1^{in}[\emptyset],\ldots,x_m^{in}[\emptyset]) \in \mathbb{R}^m}$ and $\sigma[x_k[S]] = \frac{\partial^{|S|} x_k}{\prod\limits_{l \in Active(S)} \partial x_l^{in}}\Big|_{(x_1^{in}[\emptyset],\ldots,x_m^{in}[\emptyset]) \in \mathbb{R}^m}$. Furthermore, by linearity of derivatives, we know that

$$\frac{\partial^{|S|}(x_j + x_k)}{\prod\limits_{l \in Active(S)} \partial x_l^{in}}\Big|_{(x_1^{in}[\emptyset],\ldots,x_m^{in}[\emptyset]) \in \mathbb{R}^m} = \frac{\partial^{|S|} x_j}{\prod\limits_{l \in Active(S)} \partial x_l^{in}}\Big|_{(x_1^{in}[\emptyset],\ldots,x_m^{in}[\emptyset]) \in \mathbb{R}^m} + \frac{\partial^{|S|} x_k}{\prod\limits_{l \in Active(S)} \partial x_l^{in}}\Big|_{(x_1^{in}[\emptyset],\ldots,x_m^{in}[\emptyset]) \in \mathbb{R}^m}.$$

After executing the meta-semantic rule for addition $[\![x_i[S] = x_j[S] + x_k[S]]\!](\sigma)$, we know that $\sigma'[x_i[S]] = \sigma[x_j[S]] + \sigma[x_k[S]]$ by the rules of the base interpreter in Fig. 3.5. Thus, by substitution $\sigma'[x_i[S]] = \frac{\partial^{|S|} x_j}{\prod\limits_{l \in Active(S)} \partial x_l^{in}}\Big|_{(x_1^{in}[\emptyset],\ldots,x_m^{in}[\emptyset]) \in \mathbb{R}^m} + \frac{\partial^{|S|} x_k}{\prod\limits_{l \in Active(S)} \partial x_l^{in}}\Big|_{(x_1^{in}[\emptyset],\ldots,x_m^{in}[\emptyset]) \in \mathbb{R}^m}$, hence $\sigma'[x_i[S]] = \frac{\partial^{|S|}(x_j + x_k)}{\prod\limits_{l \in Active(S)} \partial x_l^{in}}\Big|_{(x_1^{in}[\emptyset],\ldots,x_m^{in}[\emptyset]) \in \mathbb{R}^m} = \frac{\partial^{|S|} x_i}{\prod\limits_{l \in Active(S)} \partial x_l^{in}}\Big|_{(x_1^{in}[\emptyset],\ldots,x_m^{in}[\emptyset]) \in \mathbb{R}^m}$.

- *Unary Functions.* As the current state σ has been correctly initialized, before executing $[\![x_i = f(x_j)]\!]_D(\sigma)$ we inductively assume (by Lemma 3.1) that for each S, $\sigma[x_j[S]] = \frac{\partial^{|S|} x_j}{\prod\limits_{l \in Active(S)} \partial x_l^{in}}\Big|_{(x_1^{in}[\emptyset],\ldots,x_m^{in}[\emptyset]) \in \mathbb{R}^m}$. By Faa Di Bruno's formula : $\frac{\partial^{|S|} f(x_j)}{\prod\limits_{l \in Active(S)} \partial x_l^{in}}\Big|_{(x_1^{in}[\emptyset],\ldots,x_m^{in}[\emptyset]) \in \mathbb{R}^m} = \sum_{P \in Part(S)} f^{(|P|)}(x_j) \prod_{B \in P} \frac{\partial^{|B|} x_j}{\prod_{k \in B} \partial x_k}$. Further, by the inductive assumption $\frac{\partial^{|B|} x_j}{\prod_{k \in B} \partial x_k} = x_j[B]$. Additionally each $dx_i[|P|]$ will store $f^{(|P|)}(x_j)$. Thus $\frac{\partial^{|S|} f(x_j)}{\prod\limits_{l \in Active(S)} \partial x_l^{in}}\Big|_{(x_1^{in}[\emptyset],\ldots,x_m^{in}[\emptyset]) \in \mathbb{R}^m} = \sum_{P \in Part(S)} dx_i[|P|] \prod_{B \in P} x_j[B] = \frac{\partial^{|S|} x_i}{\prod\limits_{l \in Active(S)} \partial x_l^{in}}\Big|_{(x_1^{in}[\emptyset],\ldots,x_m^{in}[\emptyset])}$. Hence the meta-semantic rule for unary functions is sound.

QED.

Complexity While Theorem 3.1 tells us that for the predetermined set of input variables, the derivatives will be correctly computed, as mentioned these derivatives are only with respect to a *subset* of the input variables. To compute *all* possible derivatives with respect to all possible input variables (for the *full* Jacobian or Hessian), we must rerun $[\![P]\!]_D$ multiple times for different (valid) input states σ. In the simplest case, for $D = 1$ (dual numbers), one needs to rerun $[\![P]\!]_1$ exactly m times for functions of the form $f : \mathbb{R}^m \to \mathbb{R}^n$ to obtain the entire Jacobian. We can generalize this result for arbitrary D.

Theorem 3.2. For a program P corresponding to a function $f : \mathbb{R}^m \to \mathbb{R}^n$ where $m > D$, to compute all possible derivatives up to the D^{th} order, requires evaluating $[\![P]\!]_D$ exactly $\binom{m+D-1}{D}$ times.

Proof. (Sketch) We provide the basic intuition here. First, $\binom{m+D-1}{D}$ is the number of ways to select D items from a set of size m, with replacement when order does not matter. The D items we select are precisely the D variable we wish to differentiate with respect to. The reason why replacement is allowed is because we can differentiate with respect to the same variable multiple times. Likewise, the reason why order does not matter is because $\frac{\partial}{\partial x_i \partial x_j} = \frac{\partial}{\partial x_j \partial x_i}$, hence we do not need to recompute them separately. QED.

As we can see, for dual numbers ($D = 1$) this reduces to $\binom{m}{1} = m$ independent forward passes, as is well-known. While the complexity for higher-order AD seems expensive, it is the same complexity given in [5], and more recent work [106] has noted that specifically for higher-order derivatives, forward-mode AD is better suited than reverse mode.

3.5 ABSTRACT SEMANTICS OF HIGHER-ORDER AD

Our construction is the first to provide a generic approach to abstractly interpret the semantics of higher-order AD. The key benefit of our construction is that by exposing each derivative term explicitly in a memory state as an augmented variable, we reduce the problem of reasoning about complex mathematical objects used in AD to reasoning about standard program states. Thus, we can readily apply existing numerical abstract domains. Furthermore, because our formulation is imperative, and our transformed AD program captures data-dependence across derivative terms, we can leverage the state-based semantics during analysis to conveniently use relational abstract domains that also track correlations between variables. This allows us to precisely track correlations *across derivatives* (including different orders). We first define the preliminaries of abstract interpretation suitable for our setting.

Definition 3.9. (Abstract Interpretation) The abstract interpretation primitives that our construction requires are given by the following symbols: $(\mathcal{D}^\#, \gamma, \bot^\#, \top^\#, [\![\cdot]\!]^\#)$ where $\mathcal{D}^\#$ represents the set of abstract program states, $\gamma\colon \mathcal{D}^\# \to \mathcal{D}$ is a concretization function mapping abstract states to sets of concrete states. Further, $\bot^\#$ and $\top^\#$ are the abstract domain's respective least and greatest elements. Lastly, $[\![\cdot]\!]^\#$ are sound abstract transformers for statements in the language of Fig. 3.4.

Given an abstract domain, we can interpret the same program syntax over the abstract domain in a way that over-approximates the concrete program semantics, which are computed by $[\![\cdot]\!]$.

Definition 3.10. (Soundness) An abstract transformer $[\![\cdot]\!]^\#$ for statements is *sound* if the following holds: For any $\sigma^\# \in \mathcal{D}^\#$ and any valid syntactic expression $Expr$ in the language of Fig. 3.4:

$$[\![x_i = Expr]\!](\gamma(\sigma^\#)) \subseteq \gamma([\![x_i = Expr]\!]^\#(\sigma^\#))$$

where $[\![x_i = Expr]\!](\gamma(\sigma^\#)) = \{[\![x_i = Expr]\!](\sigma') \,:\, \sigma' \in \gamma(\sigma^\#)\}$. Hence, we say that the abstract interpreter $[\![\cdot]\!]^\#$ soundly over-approximates the semantics of the base interpreter $[\![\cdot]\!]$ of Fig. 3.5.

3.5.1 Choosing an Abstract Domain

To construct an abstract interpreter $[\![\cdot]\!]^\#_D$ to soundly over-approximate $[\![\cdot]\!]_D$, we use an existing abstract interpreter $[\![\cdot]\!]^\#$ defined over a numerical abstract domain that soundly over-approximates the semantics of the base interpreter $[\![\cdot]\!]$. Having defined the primitives needed for the construction in Defs. 3.9 and 3.10, we now detail what *restrictions* the abstract domain must satisfy.

1. The abstract domain is numeric, where $\mathcal{D}^\#$ abstracts sets of real vectors in $\mathbb{R}^{|AugVars|}$.
2. The concretization function $\gamma\colon \mathcal{D}^\# \to \mathcal{P}(\mathbb{R}^{|AugVars|})$ maps an abstract element $\sigma^\#$ to a set of points in $\mathbb{R}^{|AugVars|}$, where $\gamma(\sigma^\#) = \{\sigma \in \mathbb{R}^{|AugVars|} \;:\; \sigma \in \sigma^\#\}$.
3. $\gamma(\bot) = \emptyset$ and $\gamma(\bot) \subseteq \gamma(\sigma^\#)$ for any $\sigma^\#$.
4. For any arithmetic expression $Expr$ in Fig. 3.4, we require a sound transformer for assignments with that expression, $[\![x_i = Expr]\!]^\#\colon \mathcal{D}^\# \to \mathcal{D}^\#$.
5. (Optionally) we need an abstract test transformer $[\![x_i > c]\!]^\#\colon \mathcal{D}^\# \to \mathcal{D}^\#$ for refining abstract states with linear constraints. We also require that $\gamma([\![x_i > c]\!]^\#(\sigma^\#)) \subseteq \gamma(\sigma^\#)$.

We provide detailed explanation of Items 4 and 5 below, but we first note a (non-exhaustive) list of abstract domains satisfying these requirements.

Remark 3.3. The Interval domain, Octagon domain, Zonotope domain, DeepPoly domain, and Polyhedra domain can all be endowed with the necessary transformers to satisfy the requirements.

Assignment of Arithmetic Expressions The abstract domain must have sound transformers for assignment with arithmetic expressions of Fig. 3.4, which includes arithmetic primitives (e.g., addition and multiplication) and constants but also differentiable unary functions (e.g., exp, log).

Arithmetic Primitives

The abstract domain must provide sound transformers for all primitive arithmetic operations: $+$, $-$, $\cdot$, and $/$, as well as assignments with constants. For the interval domain, there are sound transformers for all of these. For domains like zonotopes, octagons, or polyhedra, one can always employ *linearization* of the non-linear operations of $\cdot$ and $/$ as in [102] and [107] to construct the necessary transformers. [108] has also shown how to construct DeepPoly abstract transformers for $\cdot$ and $/$.

Differentiable Unary Functions

As our language (Fig. 3.4) includes unary functions, the chosen abstract domain must provide sound transformers for the following functions: σ_a, *SoftPlus*, *exp*, and *log*. As mentioned in Section 3.4.2, the derivatives of each of these functions are given in terms of elementary arithmetic combinations of functions already in the language (e.g., $\sigma_a'(x) = \sigma_a(x) \cdot (1 - \sigma_a(x))$, hence this set of functions is "closed" under n^{th} derivatives. As with the arithmetic primitives, for the interval domain, there are sound transformers for all of these, while for zonotopes and polyhedra, one would need to soundly linearize these non-linear functions, such as the Chebyshev method for zonotopes [102], the polyhedral linearization of [107], or using the methods of [109] for the DeepPoly domain.

Automatically Improving Precision Beyond naively composing existing abstract transformers from a numerical domain, we also want to devise a method for automatically improving the precision of the analysis. Having an optimal transformer for one primitive function,

does not necessarily imply one has an optimal transformer for all derivatives of that function, particularly if the derivative requires the composition of multiple primitive function transformers (e.g., $\sigma_a' = \sigma_a(x) \cdot (1 - \sigma_a(x))$ requires composing transformers for σ_a and multiplication), since naive, off-the-shelf composition of numerical abstract transformers can be non-optimal.

To avoid this issue, we leverage analytical properties of the derivatives of the functions to systematically improve the precision of $[\![\cdot]\!]_D^\#$. We now describe an optional requirement on the domain that can improve the abstraction's precision. By using an abstract test operator $[\![x_i > c]\!]^\#(\sigma^\#)$, we can refine the abstract state $\sigma^\#$ using analytical information about the range of the function's derivatives. This allows us to enforce constraints, such as how every odd derivative of $\frac{1}{x}$ is necessarily negative. However, one can always define $[\![x_i > c]\!]^\#(\sigma^\#) = \sigma^\#$ (the identity), using the fallback transformer of [28], hence why this step is optional. By requiring $\gamma([\![x_i > c]\!]^\#(\sigma^\#)) \subseteq \gamma(\sigma^\#)$, this step never loses precision.

Remark 3.4. We do not need a widening operator, since all the programs expressible in the Fig. 3.4 syntax are loop-free. We also do not require a join $\sqcup$ or meet $\sqcap$ operator.

3.5.2 Abstract Meta-Semantics

We now provide an abstract meta-semantics which shows how to leverage an existing numerical abstract domain satisfying our criteria to construct a sound abstract interpreter for D^{th}-order AD. We highlight that the ease in defining our abstract semantics stems from design choices made in constructing our *concrete* semantics (Section 3.4.2). However, merely using the abstract version of each arithmetic operator naively is imprecise, hence we will also see how the abstract meta-semantics leverages mathematical properties of differentiable functions to improve precision.

Definition 3.11. The abstract meta-semantics for performing D-degree Taylor polynomial forward-mode abstract AD are a parametric (abstract) semantics given by $[\![\cdot]\!]_D^\# \colon (P, \mathcal{D}^\#) \to \mathcal{D}^\#$, where P is a program in the syntax of Fig. 3.4. The abstract meta-semantics are parametric in both the order of derivative D, as well as the underlying numeric abstract domain $\mathcal{D}^\#$ and its associated abstract transformers $[\![\cdot]\!]^\#$. The abstract meta-semantics ultimately return an abstract state, $\sigma^\#$ at the program exit.

This is in contrast to the concrete meta-semantics of Definition 3.6 which are parametric only in D, (they have a fixed base interpreter, $[\![\cdot]\!]$). This is because $[\![\cdot]\!]_D^\#$ can use *any* set of abstract transformers $[\![\cdot]\!]^\#$ satisfying the criteria of Section 3.5.1, thus the construction

is *configurable*. This also allows the correctness to be easily established – proving that the abstract meta-semantics soundly over-approximate the concrete meta-semantics reduces to showing that the instantiated numerical abstract domain over-approximates the base interpreter at each step.

Addition The meta-semantics of abstract addition follow very similarly to the concrete case, by adding respective derivative terms (via linearity) albeit using the abstract interpreter $[\![\cdot]\!]^\#$ of the chosen numerical abstract domain instead of the base interpreter $[\![\cdot]\!]$, noting that by the assumptions of Section 3.5.1, $[\![\cdot]\!]^\#$ has sound transformers for (abstract) addition of two variables.

$$
\begin{aligned}
[\![x_i = x_j + x_k]\!]^\#_D(\sigma^\#) \triangleq\ & \textbf{for}\ \ d \in \{0, ..., D\}: \\
& \quad \textbf{for}\ \ S \in \mathcal{P}_d(\{1, ..., D\}): \\
& \quad\quad \sigma^\# = [\![x_i[S] = x_j[S] + x_k[S]]\!]^\#(\sigma^\#) \\
& \textbf{return}\ \ \sigma^\#
\end{aligned}
$$

Figure 3.15: Abstract Meta-Semantics of Higher-Order Differentiation of Addition

Multiplication The rule for multiplication follows an abstracted form of the generalized Leibniz formula, again noting by assumption that $[\![\cdot]\!]^\#$ has sound transformers for assignments involving the sum and product of variables and that $S \setminus P$ represents set subtraction.

$$
\begin{aligned}
[\![x_i = x_j * x_k]\!]^\#_D(\sigma^\#) \triangleq\ & \textbf{for}\ \ d \in \{0, ..., D\}: \\
& \quad \textbf{for}\ \ S \in \mathcal{P}_d(\{1, ..., D\}): \\
& \quad\quad \sigma^\# = [\![x_i[S] = \textstyle\sum_{P \in \mathcal{P}(S)} x_j[P] \cdot x_k[S \setminus P]]\!]^\#(\sigma^\#) \\
& \textbf{return}\ \ \sigma^\#
\end{aligned}
$$

Figure 3.16: Abstract Meta-Semantics of Higher-Order Differentiation of Multiplication

Constants For this case, we need to assign each augmented variable to a constant (either c or 0), hence it only requires that the abstract domain have sound transformers for constant assignment.

Unary Functions

As in the concrete meta-semantics, with unary functions, we must use Faa di Bruno's formula (Def. 3.7). In the computation of the first through D^{th} derivatives of f (stored in $dx_i[1]$ through $dx_i[D]$), since we know the derivatives' analytical forms, we also know

$$
\begin{array}{l}
[\![x_i = c]\!]^{\#}_D(\sigma^{\#}) \triangleq \ \textbf{for} \ \ S \ \in \ \mathcal{P}(\{1, ..., D\}) : \\
\qquad\qquad \textbf{if} \ \ S = \emptyset : \\
\qquad\qquad\qquad \sigma^{\#} = [\![x_i[\emptyset] = c]\!]^{\#}(\sigma^{\#}) \\
\qquad\qquad \textbf{else} : \\
\qquad\qquad\qquad \sigma^{\#} = [\![x_i[S] = 0]\!]^{\#}(\sigma^{\#}) \\
\qquad\quad \textbf{return} \ \ \sigma^{\#}
\end{array}
$$

Figure 3.17: Abstract Meta-Semantics for Higher-Order Differentiation of Constants

their valid ranges, hence we can refine the abstract state $\sigma^{\#}$ to use this knowledge via the $[\![x_i > c]\!]^{\#}$ abstract test transformer. This is helpful for enforcing domain-specific knowledge. Since the refinement depends on the particular function, we only show cases for specific unary functions.

Division As noted, division is composition with the function $f(x) = \frac{1}{x}$, hence we again use Faa di Bruno's formula. Additionally, every odd derivative of $\frac{1}{x}$ is necessarily negative, hence we can apply the abstract test operator to refine $\sigma^{\#}$. Lastly because the derivatives (dx_i) that are used to compute $x_i[S]$ for $S \neq \emptyset$ are given in terms of the real part $x_i[\emptyset]$, when instantiated with a relational domain like zonotopes, we are tracking correlations *across derivative orders*.

$$
\begin{array}{l}
[\![x_i = 1/x_j]\!]^{\#}_D(\sigma^{\#}) \triangleq \quad [\![x_i[\emptyset] = 1/(x_j[\emptyset])]\!]^{\#}(\sigma^{\#}) \\
\qquad\qquad \sigma^{\#} = [\![dx_i[1] = -x_i[\emptyset] \cdot x_i[\emptyset]]\!]^{\#}(\sigma^{\#}) \\
\qquad\qquad \textbf{for} \ \ d \ \in \ \{2, ..., D\} : \\
\qquad\qquad\qquad\qquad \sigma^{\#} = [\![dx_i[d] = -d \cdot x_i[\emptyset] \cdot dx_i[d-1]]\!]^{\#}(\sigma^{\#}) \\
\qquad\qquad\qquad\qquad \textbf{if} \ \ d \ \text{ odd} : \\
\qquad\qquad\qquad\qquad\qquad \sigma^{\#} = [\![dx_i[d] < 0]\!]^{\#}(\sigma^{\#}) \\
\qquad\qquad\qquad\qquad \textbf{for} \ \ S \ \in \ \mathcal{P}_d(\{1, ..., D\}) : \\
\qquad\qquad\qquad\qquad\qquad \sigma^{\#} = [\![x_i[S] = \sum_{P \in Part(S)} dx_i[|P|] \cdot \prod_{B \in P} x_j[B]]\!]^{\#}(\sigma^{\#}) \\
\qquad\qquad \textbf{return} \ \ \sigma^{\#}
\end{array}
$$

Figure 3.18: Abstract Meta-Semantics of Higher-Order Differentiation of $\frac{1}{x}$

Exp and Log Since all n^{th} derivatives of *exp* are also *exp*, and the *exp* function is strictly positive, we can always refine the abstract state with this constraint. Additionally, we can exploit the fact that all derivatives of *exp* are also *exp*. Therefore, we only need to compute this once via the $[\![x_i[\emptyset] = exp(x_j[\emptyset])]\!]^{\#}(\sigma^{\#})$, and then we can merely copy the result into the variables corresponding to each derivative dx_i. However, the benefit of this is not only computational savings, but also the potential for improved precision if instantiating the construction with a relational abstract domain. Since all the dx_i variables are directly used

to compute the $x_i[S]$ for $S \neq \emptyset$, if $[\![\cdot]\!]^\#$ is instantiated with a relational domain, one is able to track correlations across those derivatives.

$$
\begin{aligned}
[\![x_i = exp(x_j)]\!]^\#_D(\sigma^\#) \triangleq\ & \sigma^\# = [\![x_i[\emptyset] = exp(x_j[\emptyset])]\!]^\#(\sigma^\#) \\
& \sigma^\# = [\![x_i[\emptyset] > 0]\!]^\#(\sigma^\#) \\
& \textbf{for}\ \ d \in \{1, ..., D\}: \\
& \quad \sigma^\# = [\![dx_i[d] = x_i[\emptyset]]\!]^\#(\sigma^\#) \\
& \quad \textbf{for}\ \ S \in \mathcal{P}_d(\{1, ..., D\}): \\
& \quad\quad \sigma^\# = [\![x_i[S] = \textstyle\sum_{P \in Part(S)} dx_i[|P|] \cdot \prod_{B \in P} x_j[B]]\!]^\#(\sigma^\#) \\
& \textbf{return}\ \ \sigma^\#
\end{aligned}
$$

Figure 3.19: Abstract Meta-Semantics of Higher-Order Differentiation of $\exp(x)$

Sigmoid and SoftPlus The abstract meta-semantics for these two functions follow almost identically to their concrete meta-semantics counterparts. The $(-1)^{d+k}(k!)(S_{d,k})a^d$ terms inside the summations are all constants which will be known at compile time, hence the summation simplifies to only multiplication and addition operations on the variables. Since our construction assumes abstract transformers for multiplication and addition of variables, this entire expression can be automatically and soundly analyzed. If one is using the zonotope abstract domain, then for the *SoftPlus* function, the abstract transformer $[\![x_i[\emptyset] = \mathit{SoftPlus}_a(x_j[\emptyset])]\!]^\#(\sigma^\#)$ could be the Chebyshev construction later described in Def. 3.12.

$$
\begin{aligned}
[\![x_i = \sigma_a(x_j)]\!]^\#_D(\sigma^\#) \triangleq\ & [\![x_i[\emptyset] = \sigma_a(x_j[\emptyset])]\!]^\#(\sigma^\#) \\
& \textbf{for}\ \ d \in \{1, ..., D\}: \\
& \quad \sigma^\# = [\![dx_i[d] = \textstyle\sum_{k=0}^{d}(-1)^{d+k}(k!)(S_{d,k})a^d x_i[\emptyset](1 - x_i[\emptyset])^k]\!]^\#(\sigma^\#) \\
& \quad \textbf{for}\ \ S \in \mathcal{P}_d(\{1, ..., D\}): \\
& \quad\quad \sigma^\# = [\![x_i[S] = \textstyle\sum_{P \in Part(S)} dx_i[|P|] \cdot \prod_{B \in P} x_j[B]]\!]^\#(\sigma^\#) \\
& \textbf{return}\ \ \sigma^\#
\end{aligned}
$$

Figure 3.20: Abstract Meta-Semantics of Higher-Order Differentiation of $\sigma_a(x) = \frac{1}{1+e^{-ax}}$

Abstract Sequencing Abstractly interpreting the sequencing of multiple statements follows similarly to the concrete meta-semantics.

Remark 3.5. Since all variables and their derivatives are included in the abstract state $\sigma^\#$, when instantiated with a relational domain (e.g., zonotopes), this construction produces an abstract interpreter that captures dependencies across derivatives and derivative orders.

$$
\begin{array}{l}
[\![x_i = SoftPlus_a(x_j)]\!]^{\#}_D(\sigma^{\#}) \triangleq\ [\![x_i[\emptyset] = SoftPlus_a(x_j[\emptyset])]\!]^{\#}(\sigma^{\#}) \\
\qquad\qquad \sigma^{\#} = [\![dx_i[1] = \sigma_a(x_j[\emptyset])]\!]^{\#}(\sigma^{\#}) \\
\qquad\qquad \textbf{for}\ \ d\ \in\ \{2, ..., D\}: \\
\qquad\qquad\quad \sigma^{\#} = [\![dx_i[d] = \textstyle\sum_{k=0}^{d}(-1)^{d+k}(k!)(S_{d,k})a^d dx_i[1]](1 - dx_i[1])^k]\!]^{\#}(\sigma^{\#}) \\
\qquad\qquad\quad \textbf{for}\ \ S\ \in\ \mathcal{P}_d(\{1, ..., D\}): \\
\qquad\qquad\qquad \sigma^{\#} = [\![x_i[S] = \textstyle\sum_{P \in Part(S)} dx_i[|P|] \cdot \prod_{B \in P} x_j[B]]\!]^{\#}(\sigma^{\#}) \\
\qquad\qquad \textbf{return}\ \ \sigma^{\#}
\end{array}
$$

Figure 3.21: Abstract Meta-Semantics of Higher-Order Differentiation of $SoftPlus_a(x) = \frac{1}{a}\log(1 + e^{ax})$

$$
[\![P_1; P_2]\!]^{\#}_D(\sigma^{\#}) \triangleq [\![P_2]\!]^{\#}_D\big([\![P_1]\!]^{\#}_D(\sigma^{\#})\big)
$$

Figure 3.22: Abstract Meta-Semantics of Higher-Order Differentiation of Sequencing

3.5.3 Soundness

Intuitively the soundness follows from construction. By composing abstract transformers at each step that are sound for each primitive, the end construction soundly over-approximates the original concrete semantics. However, we now state this formally.

Theorem 3.3. (Soundness of Abstraction) For any program P expressible in the syntax of Fig. 3.4, $D \in \mathbb{N}$, $\sigma^{\#} \in \mathcal{D}^{\#}$ and any $\sigma \in \gamma(\sigma^{\#})$, then we have that $[\![P]\!]_D(\sigma) \in \gamma([\![P]\!]^{\#}_D(\sigma^{\#}))$.

Thus, we can compute a sound over-approximation of the set of values that the (possibly higher-order) derivatives take, provided all the requirements of Section 3.5.1 are satisfied.

Proof. (Sketch) We provide sketches of main cases:

• *Addition:* We need to show that for any D, $\sigma^{\#} \in \mathcal{D}^{\#}$ with $\sigma \in \gamma(\sigma^{\#})$ that the following holds: $[\![x_i = x_j + x_k]\!]_D(\sigma) \in \gamma([\![x_i = x_j + x_k]\!]^{\#}_D(\sigma^{\#}))$. The idea is straightforward, for each application of $\sigma = [\![x_i[S] = x_j[S] + x_k[S]]\!](\sigma)$ in the for loop of the meta-semantic rule for $[\![x_i = x_j + x_k]\!]_D(\sigma)$, there is the corresponding abstract application $\sigma^{\#} = [\![x_i[S] = x_j[S] + x_k[S]]\!]^{\#}(\sigma^{\#})$, thus by the initial assumption that $\sigma \in \gamma(\sigma^{\#})$, and that fact that $[\![\cdot]\!]^{\#}$ soundly over-approximates $[\![\cdot]\!]$ for addition (by assumption of the sound abstract transformers in Section 3.5.1) for every application (regardless of D) we know $[\![x_i[S] = x_j[S] + x_k[S]]\!](\sigma) \in \gamma([\![x_i[S] = x_j[S] + x_k[S]]\!]^{\#}(\sigma^{\#}))$.

• *Multiplication:* We need to show that for any D, $\sigma^{\#} \in \mathcal{D}^{\#}$ with $\sigma \in \gamma(\sigma^{\#})$ that the following holds: $[\![x_i = x_j * x_k]\!]_D(\sigma) \in \gamma([\![x_i = x_j * x_k]\!]^{\#}_D(\sigma^{\#}))$. The idea is straightforward, for each application of $\sigma = [\![x_i[S] = \sum_{P \in \mathcal{P}(S)} x_j[P] \cdot x_k[S \setminus P]]\!](\sigma)$ in the for loop of the meta-semantic rule for $[\![x_i = x_j * x_k]\!]_D(\sigma)$, there is the corresponding abstract application

$\sigma^\# = [\![x_i[S] = \sum_{P \in \mathcal{P}(S)} x_j[P] \cdot x_k[S \setminus P]]\!]^\#(\sigma^\#)$ in the abstract meta-semantics. Thus by the initial assumption that $\sigma \in \gamma(\sigma^\#)$, and that fact that $[\![\cdot]\!]^\#$ soundly over-approximates $[\![\cdot]\!]$ for addition and multiplication (by assumption of the sound abstract transformers in Section 3.5.1) then for every application (regardless of D) we know $[\![x_i[S] = \sum_{P \in \mathcal{P}(S)} x_j[P] \cdot x_k[S \setminus P]]\!](\sigma) \in \gamma([\![x_i[S] = \sum_{P \in \mathcal{P}(S)} x_j[P] \cdot x_k[S \setminus P]]\!]^\#(\sigma^\#))$.

• *Division:* We need to show that for any D, $\sigma^\# \in \mathcal{D}^\#$ with $\sigma \in \gamma(\sigma^\#)$ that the following holds: $[\![x_i = 1/x_j]\!]_D(\sigma) \in \gamma([\![x_i = 1/x_j]\!]^\#_D(\sigma^\#))$. The idea is the same, we note that for each application $\sigma = [\![dx_i[d] = -d \cdot x_i[\emptyset] \cdot dx_i[d-1]]\!](\sigma)$ in the concrete meta-semantics, that the corresponding application $\sigma^\# = [\![dx_i[d] = -d \cdot x_i[\emptyset] \cdot dx_i[d-1]]\!]^\#(\sigma^\#)$ in the abstract meta-semantics soundly over-approximates it by assumption of the sound multiplication abstract transformers in Section 3.5.1. To address the application of $[\![dx_i[d] < 0]\!]^\#(\sigma^\#)$, we note that in the concrete semantics, $dx_i[d]$ will never be greater than 0 if d is odd since all odd derivatives of $\frac{1}{x}$ are negative. Hence this refinement of the abstract state is always sound. Thus, even after applying this refinement, we still have that $\sigma \in \gamma(\sigma^\#)$. Likewise, for each application $\sigma = [\![x_i[S] = \sum_{P \in Part(S)} dx_i[|P|] \cdot \prod_{B \in P} x_j[B]]\!](\sigma)$ in the concrete meta-semantics, the corresponding application $\sigma^\# = [\![x_i[S] = \sum_{P \in Part(S)} dx_i[|P|] \cdot \prod_{B \in P} x_j[B]]\!]^\#(\sigma^\#)$ soundly over-approximates it.

• *Sequencing:* This case represents the inductive step. By assumption for any $\sigma \in \gamma(\sigma^\#)$, $[\![P_1]\!]_D(\sigma) \in \gamma([\![P_1]\!]^\#_D(\sigma^\#))$. Likewise by assumption for any $\sigma_2 \in \gamma(\sigma_2^\#)$ $[\![P_2]\!]_D(\sigma_2) \in \gamma([\![P_2]\!]^\#_D(\sigma_2^\#))$. Hence, by substituting $\sigma_2 = [\![P_1]\!]_D(\sigma)$ and $\sigma_2^\# = [\![P_1]\!]^\#_D(\sigma^\#)$, we obtain $[\![P_1; P_2]\!]_D(\sigma) \in \gamma([\![P_1; P_2]\!]^\#_D(\sigma^\#))$. QED.

3.6 INSTANTIATIONS

We formally describe instantiations of our framework, which we will later evaluate in order to study the effects of first vs. higher derivatives as well as relational vs. non-relational abstract domains.

3.6.1 Interval AD

While interval arithmetic has been applied to AD, our construction is more general, thus using the interval domain is but one instantiation of our framework.

Interval Domain The interval domain, denoted $\mathcal{D}^\# = \mathcal{P}(\mathbb{IR}^{|AugVars|})$, is among the simplest numeric abstract domains [28] where in an abstract state $\sigma^\#$ a variable is mapped to an interval $[a, b] \in \mathbb{IR}$, hence $\sigma^\# : AugVar \to \mathbb{IR}$. The interval domain is *non-relational*.

First Derivatives Our construction can be used to produce the Dual Interval domain of [34]. This is an instantiation of our framework where $D = 1$ with the interval domain.

Higher Derivatives We may also encode an interval abstraction of the hyperdual numbers of [105] using our framework, which allows us to obtain interval bounds on second derivatives. This is done by instantiating our construction with $D = 2$ and the interval domain.

3.6.2 Zonotope AD

To the best of our knowledge, prior to our work, there has been no general zonotope abstract interpretation for forward-mode AD, especially for higher-order derivatives.

Zonotope Abstract Domain We first describe the zonotope abstract domain [101], which we denote with $\mathcal{D}^\# = \mathbf{Zono}$. The abstract state $\sigma^\# \in \mathbf{Zono}$ maps each variable to an affine form, where an affine form is a tuple (c, g) with center $c \in \mathbb{R}$ and $g \in \mathbb{R}^{|generators|}$ where $|generators|$ is the number of noise symbols (also called generators). For a variable x, to denote its affine form, we will write $x = c + \sum_{i=1}^{|generators|} g_i\epsilon_i$, where each noise symbol $\epsilon_i \in [-1, 1]$. To index a variable's affine form in the abstract state $\sigma^\#$, we may write $\sigma^\#[x] = c + \sum_{i=1}^{|generators|} g_i\epsilon_i$ as well as $\sigma^\#[x][c]$ and $\sigma^\#[x][g_i]$ to access the coefficients of the center and noise symbol terms. Because the ϵ_i are shared across variables, this is a relational domain; furthermore, the set of states encoded by $\sigma^\#$ is exactly a zonotope. This idea can be seen from the concretization function $\gamma\colon \mathbf{Zono} \to \mathcal{P}(\mathbb{R}^{|AugVars|})$ where the concretization of an abstract element, $\gamma(\sigma^\#)$, is defined as the following set:

$$\gamma(\sigma^\#) = \{(x_0, ..., x_{|AugVars|}) \in \mathbb{R}^{|AugVars|} \mid \forall j, i :\ x_j = \sigma^\#[x_j][c] + \sum_{i=1}^{|generators|} \sigma^\#[x_j][g_i]\cdot\epsilon_i \wedge \epsilon_i \in [-1, 1]\} \tag{3.1}$$

As mentioned in [110], $|generators|$ grows as the program executes since new noise symbols (the $\epsilon_i \in [-1, 1]$) are dynamically added with each non-linear operation (e.g., multiplication, $\sigma_a(x)$, etc.). However, the zonotope domain loses no precision when encoding affine transformations, since these can be done precisely in the domain.

One can also convert an affine form $x = c + \sum_{i=1}^{|generators|} g_i\epsilon_i$ to an interval $[lb(x), ub(x)] \in \mathbb{IR}$ where the lower bound $lb(x)$ is given as $lb(x) = c - \sum_{i=1}^{|generators|} |g_i|$ and likewise an upper bound can be computed as $ub(x) = c + \sum_{i=1}^{|generators|} |g_i|$. This will prove useful when constructing sound transformers for arithmetic primitives.

Differentiable Zonotope Transformers While the construction of Section 3.5 shows us how to produce a D^{th}-order forward-mode AD abstract interpreter $[\![\cdot]\!]_D^\#$ using the chosen abstract domain $\mathcal{D}^\#$, it assumes the existence of sound transformers, $[\![\cdot]\!]^\#$, for the arithmetic functions and gives no way to produce them. However, for zonotopes, prior works [52, 93] give sound transformers for some (e.g., *tanh*), but not all of the functions we support. Therefore, when instantiating with zonotopes, we need an automatic construction for sound differentiable function transformers. To do so, we use the Chebyshev construction [102, 111]:

Definition 3.12. Given a function $f\colon \mathbb{R} \to \mathbb{R}$ that is bounded and twice differentiable on some interval $[lb(x), ub(x)] \in \mathbb{IR}$, such that f' is invertible and f'' does not change signs on $[lb(x), ub(x)]$, then for an affine form $x = c + \sum_{i=1}^{|generators|} g_i\epsilon_i$, the affine form for $y = f(x)$ can be given as:

$$y = (\alpha \cdot c + \zeta) + \Big(\sum_{i=1}^{|generators|} (\alpha \cdot g_i)\epsilon_i \Big) + \delta\epsilon_{new},$$

where $\alpha = \frac{f(ub(x))-f(lb(x))}{ub(x)-lb(x)}$, $r(v) = \frac{f(ub(x))-f(lb(x))}{ub(x)-lb(x)}(v - lb(x)) + f(lb(x))$, $u = f'^{-1}(\alpha)$, $\zeta = -\alpha u + \frac{f(u)+r(u)}{2}$ and $\delta = \frac{|f(u)-r(u)|}{2}$. Following the definition in [111], we refer to this formula as the Chebyshev construction for f.

The Chebyshev construction gives us sound zonotope transformers for the following functions: $SoftPlus_a(x)$, $log(x)$, $exp(x)$, and $\frac{1}{x}$. For the other functions, we can use existing zonotope transformers (e.g., those in [52]) or just default to the interval domain abstract transformers. We can then collectively use all of these abstract transformers in the general construction of $[\![\cdot]\!]_D^\#$.

Theorem 3.4. When $D = 2$ and when the abstract domain is the zonotope domain, $[\![\cdot]\!]_D$ computes a zonotope abstraction of the hyper-dual numbers of [105].

Precision For many functions, the zonotope transformers are strictly more precise than the interval ones. As mentioned in [102, 111], the Chebyshev construction is *optimal* in the input-output plane, hence it gives the same guarantees as [52]. The zonotope transformers in this case are incomparable to the interval ones. We also found that existing abstract test transformers for zonotopes [30, 112] do not boost precision, but increase cost. Hence, when instantiating our construction with zonotopes, we use the identity function $[\![x_i > c]\!](\sigma^\#) = \sigma^\#$ for the tests.

3.7 CASE STUDIES

We now present a set of case studies that highlight the benefits of an abstraction supporting both higher-order derivatives and more precise abstract domains.

3.7.1 Methodology

We briefly describe the experimental setup and the network architectures used in our experiments. We ran our experiments on a 3.70 GHz Intel Xeon W-2135 CPU with 32 GB of main memory. For the first case study on the robust interpretation of first- and second-order effects, we trained a 3-layer neural network with 5 inputs, 64 neurons in the first hidden layer, 10 neurons in the second hidden layer, and 1 output neuron to approximate a 5-input, 1-output synthetic function polynomial of degree two with five interactions; the network uses the *SoftPlus* activation function after every layer. For the Lipschitz certification case study, we trained four fully connected networks on the MNIST [50] image classification dataset (which contains 70,000 28×28 images of handwritten digits) – three are smaller networks with 3, 4, and 5 layers having 100 neurons in each hidden layer, and one is the FFNNBig architecture from [53] consisting of 4,106 neurons (4 hidden layers of 1,024 neurons each). Each hidden layer is followed by a *SoftPlus* activation function. All networks attain over 98% accuracy on the test set. For the perturbation function, we consider the Haze perturbation studied in [61]. For all experiments, we also present runtimes for the analyses.

Implementation

The instantations of our framework for both first and second derivatives on the interval and zonotope domains were all implemented in PyTorch [37].

3.7.2 Robust Derivative-Based Interpretations

As neural networks are notoriously hard to understand, significant work has focused on constructing more interpretable explanations, in order to understand which features are most relevant to the network's outputs. A standard way of interpreting and explaining neural networks involves computing derivatives of the network's outputs with respect to their inputs [18, 19]. However, prior work has shown that it is not enough to generate explanations for scalar points; rather, explanations should be robust, hence why [100] used robust interval explanations. Our case study serves to show how our construction allows a user to com-

Table 3.1: Improved bounds on Jacobian by using zonotopes over intervals.

Jacobian Entry	Width Reduction Factor
J_1	4.57x
J_2	4.51x
J_3	4.76x
J_4	4.20x
J_5	4.39x

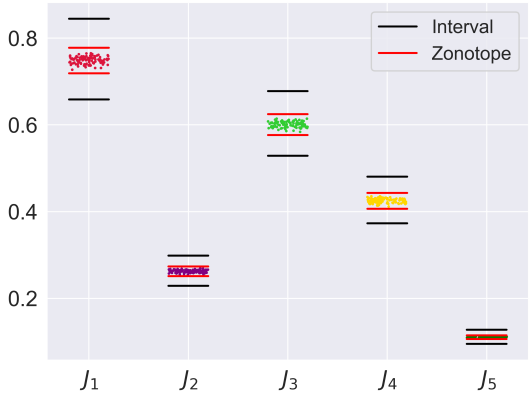


Figure 3.23: Bounds on Jacobian entries via the interval and zonotope abstract domains.

Table 3.2: Improved bounds on Hessian by using zonotopes over intervals.

Width Red.	$H_{i,1}$	$H_{i,2}$	$H_{i,3}$	$H_{i,4}$	$H_{i,5}$
$H_{1,j}$	6.98x	5.69x	6.40x	5.31x	5.43x
$H_{2,j}$		4.97x	4.65x	4.07x	4.41x
$H_{3,j}$			4.64x	4.33x	4.62x
$H_{4,j}$				3.92x	4.08x
$H_{5,j}$					4.76x

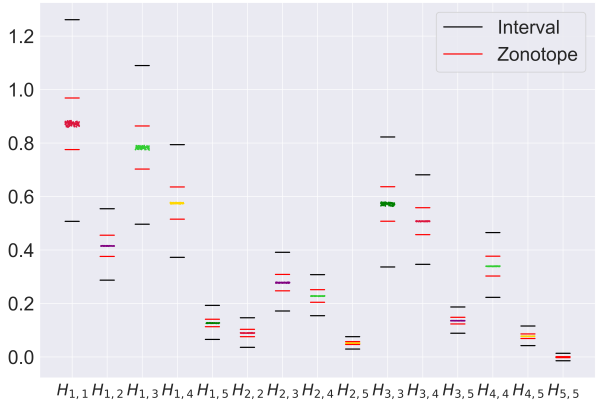


Figure 3.24: Bounds on Hessian entries via the interval and zonotope abstract domains.

pute provably robust interpretations via derivative-based first-order feature attributions and second-order feature interactions. For our experiments, we uniformly generate 5 random inputs in the range $[0, 1)$ and enclose each input in an interval of ± 0.01. We then soundly bound the first and second derivatives of the neural network (trained to approximate a polynomial as described in Section 3.7.1) with respect to these input ranges. We repeat this experiment for 5 different seeds.

Table 3.1 and Table 3.2 demonstrate the precision improvement of the zonotope domain on bounding the Jacobian and the Hessian, respectively. In both tables, each cell presents the geometric mean over 5 trials of the ratio of the interval-bounded derivative's width over its zonotope-bounded counterpart. For zonotopes and intervals, the average runtimes across all trials are 0.014 and 0.009 seconds for the first-order information and 0.23 and 0.11 seconds for the second-order information, respectively. This demonstrates that the substantial increase in precision when using the zonotope domain does not incur a large runtime overhead. Furthermore, performing the separate passes to compute each Jacobian or Hessian term takes virtually the same time, meaning that the input one differentiates with respect to does not affect the runtime.

We next show detailed results for the trial that yields the median improvement in precision

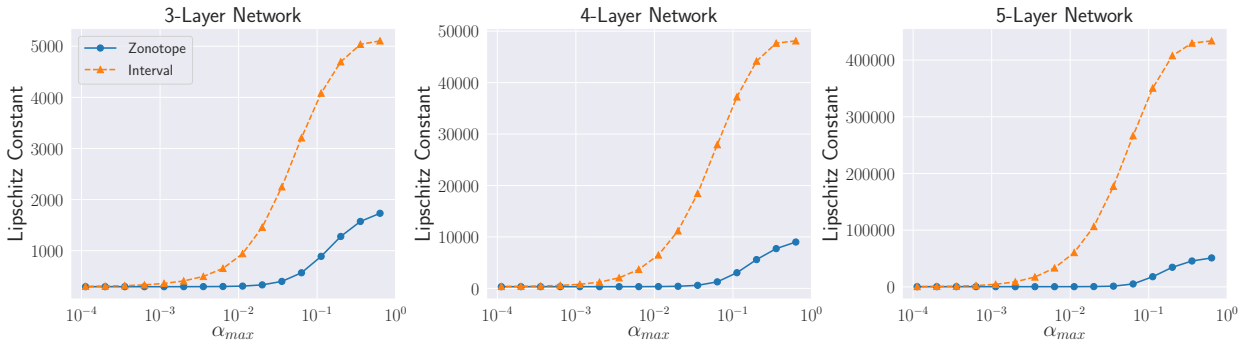
when using zonotopes over intervals (the other trials show the same trend). Fig. 3.23 presents the bounds on the first derivatives (the Jacobian). The entries on the x-axis correspond to the first derivative of the network's output node with respect to each of the 5 input variables. The value of the y-axis denotes the value of the derivative. We compare derivatives computed at sampled scalar points with both an interval and zonotope instantiation of our framework for first derivatives. The zonotope bounds on the Jacobian entries are much tighter than the bounds computed via the interval domain, both of which enclose the sampled points (shown in multiple colors).

We also compute the second derivatives to study the magnitude of interactions between input variables over an entire region of the input space. Fig. 3.24 presents the results (for the same trial as in Fig. 3.23). The entries along the x-axis represent Hessian terms instead of just individual input derivatives (as in Fig. 3.23). As before, the y-axis represents the magnitude of the (second) derivative. Zonotope bounds on the second derivatives are also much more precise at enclosing the sampled points than standard interval domain bounds, which stems from the fact that relational domains allow us to preserve correlation across derivative orders, thus improving precision.

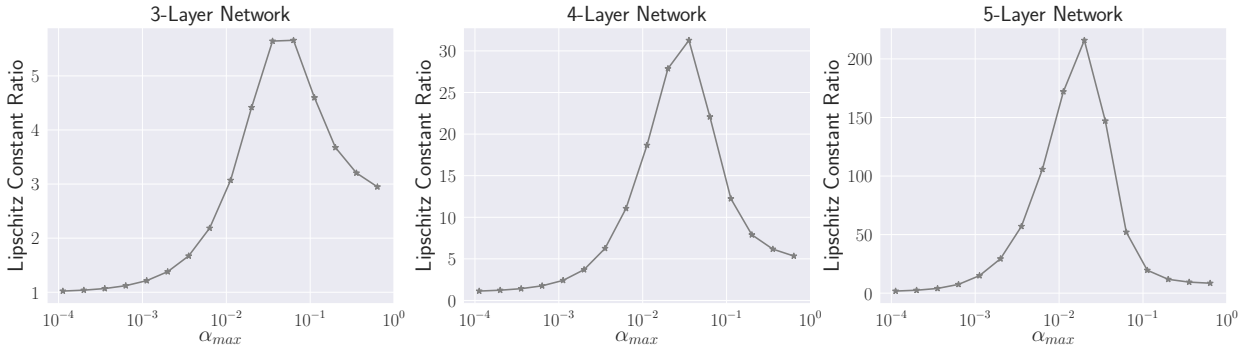
3.7.3 Lipschitz Certification

Our second case study involves bounding the local Lipschitz constant of neural networks with respect to semantic perturbations, as in [34, 113]. We study networks trained on MNIST data and consider the Haze perturbation, defined as the pixelwise transformation $p_\alpha(x_i; \alpha) = (1 - \alpha)x_i + \alpha$, where x_i is the value of each pixel and α represents the haze amount. Let f denote a neural network. Then, given an image x, we compute a sound upper bound on the (local) Lipschitz constant of $f(p_\alpha(x; \alpha))$ with respect to α over *a range of* haze values, specifically where α lies in $[0, \alpha_{max}]$. The parameter α_{max} represents the maximum haze amount, and we consider values of $\alpha_{max} \in \{10^{-k/4} \cdot 2 : k \in [2, 18]\}$. We evaluate on the zonotope domain (enabled by this work) and compare with [34] (as no other work can handle this setting).

Fig. 3.25 shows the Lipschitz certification results for the 3-layer (left), 4-layer (center), and 5-layer (right) MNIST networks. Fig. 3.25a presents the computed Lipschitz constant bounds. The x-axis shows the value of α_{max}, and the y-axis shows the upper bound on the Lipschitz constant computed with the zonotope and interval domains (smaller is better). Fig. 3.25b presents the increase in precision of the zonotope domain. The x-axis again shows the value of α_{max}, and the y-axis shows the ratio of the interval-bounded Lipschitz constants over the zonotope-bounded Lipschitz constants (larger is better). For each plot, the x-axis



(a) Average upper bound on the local Lipschitz constant for the zonotope and interval domains.



(b) Increase in precision of the zonotope domain over the interval domain.

Figure 3.25: Lipschitz certification of 3-layer (left), 4-layer (center), and 5-layer (right) MNIST networks against the haze perturbation on 1,000 correctly classified test-set images. The top row (Fig. 3.25a) presents the average upper bound on the local Lipschitz constant for both the zonotope and interval domains. The bottom row (Fig. 3.25b) presents the increase in precision of the zonotope domain, computed as the ratio of the interval-bounded Lipschitz constant over the zonotope-bounded Lipschitz constant.

uses a logarithmic scale, while the y-axis uses a linear scale. Each data point is the average over the first 1,000 correctly classified test set images. For the 3-, 4-, and 5-layer networks, the average zonotope runtimes are 2.8, 4.1, and 5.9 milliseconds per image and the average interval runtimes are 2.9, 3.2, and 4.0 milliseconds per image, respectively. These runtimes showcase how our analysis, even when using a precise relational domain, is fast and scalable.

AD with zonotopes is always more precise than AD with intervals – the zonotope-bounded Lipschitz constants are always smaller. Compared to intervals, zonotopes are up to 6×, 31×, and 216× more precise and the computed Lipschitz constants are up to 3374, 39105, and 382755 smaller for the 3-, 4-, and 5-layer networks, respectively. This showcases how AD with intervals is much more over-approximate for deeper networks, while zonotope AD retains much more precision.

For each network, we see a similar trend across values of α_{max}. Fig. 3.25a shows that as α_{max} (i.e., the range of perturbation) increases, the *absolute* difference between the interval-

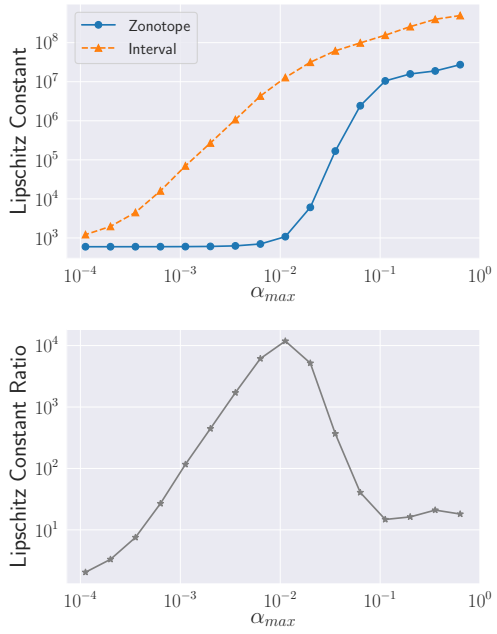


Figure 3.26: Top: Average local Lipschitz constant upper bounds (lower is better) for the zonotope and interval domains on FFNNBig. Bottom: the increase in precision of the zonotope over the interval domain.

and zonotope-bounded Lipschitz constant increases significantly. For small perturbation ranges (when $\alpha_{max} < 10^{-3}$), the Lipschitz constant between the two domains are similar in magnitude. Fig. 3.25b shows that for small values of α_{max}, both domains are quite precise, since the input range is very small. For large values of α_{max}, both domains incur larger degrees of over-approximation (though intervals are much more over-approximate). This is why the relative increase in precision of zonotopes over intervals is not as great for both ends of the input range. Zonotopes are *relatively* most precise for an input range that is around $10^{-1.5}$. The maxima in Fig. 3.25b exactly correspond to the points in Fig. 3.25a where the zonotope curve already asymptotes (i.e., induces little over-approximation), while the interval curve is still very over-approximate. Zonotopes allow us to obtain much tighter bounds on neural networks' Lipschitz constants with respect to semantic perturbations than the previous interval-domain work of [34].

Our approach can also scale to larger networks: Fig. 3.26 shows the Lipschitz certification results on the FFNNBig architecture. All axes are in log scale for clarity; as before, each data point is the average over the first 1,000 correctly classified test images. The average zonotope and interval runtimes for this experiment are 0.58 and 0.12 seconds per image, respectively. Compared to the interval domain, our zonotope domain construction obtains Lipschitz constants that are up to 11,850× more precise, and the computed Lipschitz con-

stants are up to 4.67×10^8 smaller. Again, we can observe that using the more precise zonotope domain (versus just intervals) increases analysis precision significantly for larger networks, highlighting the importance of our analysis.

3.8 RELATED WORK

To the best of our knowledge, there is not any work that provides a general construction for abstract interpretation of higher-order AD.

Higher-Order AD Higher-order forward-mode AD has been explored going back to [103] and [95]. [96] also proposed a forward-mode scheme; then, later implementations in JAX [114], λ_S [38], and the work of [115] also followed in this spirit. In particular, λ_S also supports Clarke Jacobians. However, all of these works only give *concrete* (and not abstract) semantics for AD. Additionally, unlike our concrete semantics, [38, 92, 95, 96, 115] all define their concrete semantics in functional styles, whereas we use an imperative semantics (that generalizes [105]) for ease in building the formalism. Lastly, while we refer to our semantics as propagating tuples of Taylor coefficients, these are technically closer to Tensor coefficients of [5], as our tuple coefficients are unscaled by factorial terms.

Abstract Interpretation of AD As pointed out in [34] and [116], there is very little work on Abstract Interpretation for AD. Most recently, [34] developed an interval abstract interpretation for bounding Clarke Jacobians; however, they cannot support relational domains or higher derivatives. [93] provide a way to propagate zonotopes through vector-Jacobian products, but their formalism, soundness guarantees, and implementation are only valid for first derivatives. The benefit of their work is that beyond improving precision, by capturing linear dependencies between derivative terms, they can solve linear programming optimization problems defined over the derivative terms in closed form.

Other works like [116] and [41] use the interval domain to bound AD, but their works cannot use any relational domain (e.g., zonotopes or polyhedra). [16] can abstract higher-order derivatives and in fact use derivatives up to third order for abstract sensitivity analysis (applied to approximate computing); however, like the other works, they are also restricted to only the interval domain. [94] develops a solver to bound the solutions of differential equations using interval and zonotope bounds on first and second derivatives. However, all their derivatives are evaluated purely symbolically, and so they make no use of automatic differentiation; thus, their work suffers scalability problems. Despite their derivative computations being purely symbolic, evaluating those symbolic expressions using zonotopes to

capture linear dependencies between first and second-order derivative terms does greatly improve the precision of their verified second-order Runge Kutta solver. Also motivated by verified ODE solutions, [7] and [117] develop a validated AD framework for computing interval arithmetic liftings of the higher-order derivatives needed to compute higher-order Taylor series expansions of the solution to an ODE, but their work is restricted to only the interval domain.

Meta-Abstract Interpretation There are a few works which provide meta-abstract interpretations [82, 118, 119, 120], meaning they do not propose a new abstract domain, but rather a new construction that can be instantiated with different abstract domains. However, none of these works target AD. Additionally, there are techniques to automate the construction of transformers for a variety of numerical abstract domains [107], but the precision and scalability of these transformers are usually suboptimal.

3.9 SUMMARY

We developed the first general construction for abstract interpretation of automatic differentiation that supports both higher-order derivatives and virtually any numerical abstract domain. We instantiate our method with both intervals and zonotopes, and in the latter case, we show how to relationally leverage dependencies across derivatives to further improve precision. Our evaluation demonstrates the applicability and scalability of our technique.

CHAPTER 4: SYNTHESIZING PRECISE STATIC ANALYZERS WITH PASADO

We present Pasado, a technique for synthesizing precise static analyzers for Automatic Differentiation. Our technique allows one to automatically construct a static analyzer specialized for the Chain Rule, Product Rule, and Quotient Rule computations for Automatic Differentiation in a way that abstracts all of the nonlinear operations of each respective rule simultaneously. By directly synthesizing an abstract transformer for the composite expressions of these three most common rules of AD, we are able to obtain significant precision improvement compared to prior works which compose standard abstract transformers together suboptimally. We prove our synthesized static analyzers sound and additionally demonstrate the generality of our approach by instantiating these AD static analyzers with different nonlinear functions, different abstract domains (both intervals and zonotopes) and both forward-mode and reverse-mode AD.

We evaluate Pasado on multiple case studies, namely computing certified bounds on a neural network's local Lipschitz constant, soundly bounding the sensitivities of financial models, certifying monotonicity, and lastly, bounding sensitivities of the solutions of differential equations from climate science and chemistry for verified ranges of initial conditions and parameters. The local Lipschitz constants computed by Pasado on our largest CNN are up to 2750× more precise compared to the existing state-of-the-art zonotope analysis. Additionally, the bounds obtained on the sensitivities of the climate, chemical, and financial differential equation solutions are between $1.31 - 2.81\times$ more precise (on average) compared to a state-of-the-art zonotope analysis.

4.1 INTRODUCTION

While Chapters 1, 2 and 3 of this dissertation describe various abstract interpretation approaches used to certify guarantees on derivatives computed by AD, limitations still exist. Earlier Abstract Interpretation work focused primarily on developing precise abstract transformers for *linear* operations and assignments [29, 30]. However, AD computations are *highly non-linear*. While one could reduce these groups of nonlinear operations to a series of basic (nonlinear) primitives, and then compose the corresponding primitives' abstract transformers, this construction loses precision. While one could try to design custom abstract transformers for groups of nonlinear operations, as in [121], this idea requires significant human expertise to handcraft the transformers and must be redone for each different function (e.g. $\tanh(x)$, $\sigma(x)$) [111] and for each different abstract domain. Hence to date, there exists

no general recipe for constructing AD static analyzers which can precisely abstract multiple non-linear operations for different functions, support multiple abstract domains and support both modes of AD.

Challenges. We focus on developing a general technique to synthesize precise static analyzers *tailored to the needs and structure of AD.* By abstracting multiple nonlinear computations all at once instead of suboptimally composing abstractions of each primitive operation, we seek to significantly improve the precision of the abstract interpretation of AD. Lastly, we seek to remove the burden on the abstraction designer from having to design transformers for each function and each abstract domain and each mode of AD. However, this approach must overcome two major challenges:

1. *Determining the right granularity for the abstraction.* While grouping more operations together for the abstraction improves precision, the difficulty in synthesizing abstract transformers scales with the number of operations due to the need to solve multivariate optimization problems. Hence we must identify patterns to abstract that strike a balance between scalability and precision. The composite abstraction of a group of AD operations should be general enough to yield efficient and precise static analyzers for (a) multiple different functions (e.g. $\sigma(x), \exp(x), \sqrt{x}$) and (b) be instantiated with different abstract domains (e.g. interval, zonotope[101], DeepPoly[53]). We also want to ensure the AD patterns we abstract are general enough to support forward-mode *and* reverse-mode AD.

2. *Proving the synthesis procedure obtains sound abstract transformers for different functions and abstract domains.* To synthesize an abstract transformer for a group of nonlinear operations over multiple variables, one must first solve a multi-dimensional, non-convex optimization problem to obtain soundness guarantees. However, we also want the proof techniques used to prove soundness to be general so that we can apply them to different nonlinear functions and have these proof technique support different abstract domains.

Our Work. To address these challenges, this Chapter proposes Pasado. The main idea of Pasado is to soundly synthesize precise linear abstract transformers that are tailored to the Chain rule, Product Rule and Quotient Rule expressions of AD. Given their ubiquity in AD programs, we have identified that these patterns strike a desirable balance between efficiency, precision and tractability of the synthesized transformers. Pasado synthesizes sound abstract transformers for these *composite, multivariate, nonlinear* AD expressions directly instead of naively composing standard abstractions for each primitive nonlinear operation together. Additionally, Pasado supports both forward- and reverse-mode AD. Pasado's procedure is automated, hence one need not design hand-crafted abstract transformers as in [52]

and [122]. Furthermore, Pasado is general enough to support different function primitives and different abstract domains so that this idea can be fully leveraged to synthesize general static analyzers for differentiable programming languages. Lastly, Pasado is tractable, scalable and efficient enough to analyze large computations such as automatically differentiating through Convolutional Neural Networks (CNNs).

Pasado synthesizes these abstract transformers using a multi-step procedure. We first use linear regression to fit a hyperplane to the pattern we are trying to abstractly interpret. This step involves sub-sampling the AD expression at concrete points in a given input interval and using those evaluations for the linear regression. These input bounds may come from variables in the primal, hence Pasado simultaneously uses existing abstract interpretation methods to obtain bounds on the primal. Additionally, because the hyperplane is only a linear approximation it is unsound, hence we must also account for the deviation between this hyperplane and the original function to maintain soundness. Thus we solve for the maximum deviation between this hyperplane and the original AD expression we are abstracting and use this deviation to obtain sound linear bounds enclosing the hyperplane. A core part of our contribution is the mathematical proof of an efficient solution to this maximization problem. Specifically, we show how to provably rule out virtually all critical points, effectively reducing this optimization problem to examining boundary points.

We evaluate Pasado on multiple Case Studies including (1) robust sensitivity analysis of ODE solutions for both Neural ODEs and climate models, (2) provable bounds on sensitivities of financial models, (3) local Lipschitz robustness certification and (4) monotonicity certification. Due to Pasado's precision and scalability, the bounds obtained on the sensitivities of the climate, chemical, and financial differential equation solutions are between $1.31 - 2.81\times$ more precise (on average) compared to a state-of-the-art zonotope AD analysis and on our largest CNN, the local Lipschitz constants computed by Pasado are up to $2750\times$ more precise compared to the existing state-of-the-art zonotope AD analysis from [35].

Contributions. In summary, this chapter makes the following contributions:

1. **Approach**. We present Pasado, a general formalism for synthesizing precise static analyzers for composite AD expressions. Due to Pasado's generality, this approach can be instantiated for a broad class of functions with both forward and reverse mode AD and with different abstract domains. (Sections 4.4.1-4.4.3)

2. **Guarantees**. We formally prove the soundness of the abstract transformers synthesized by Pasado (Section 4.4.4) and theorems about their precision and generality (Sections 4.4.5-4.4.6).

```
function ODESolve_f(Q, α, e, R, t_0, T_0, step_size, steps) :
    for (i=1; i < steps; i++) {
        T_i = T_{i-1} + f(T_{i-1}, t_{i-1}, Q, α, e, R) · step_size   // Euler integrator
        t_i = t_{i-1} + step_size }
    return T, t
```

Figure 4.1: Numerical ODE Solver Source Code

3. **Implementation**. We implement Pasado as a practical tool and instantiate it with both forward- and reverse-mode AD, as well as both the interval and zonotope domains and their reduced product. (Section 4.5.1). Pasado is available at `https://github.com/uiuc-arc/Pasado`.

4. **Evaluation**. We experimentally show the benefit of Pasado on Lipschitz certification, monotonicity analysis and differential equation models from chemistry, finance, and climate science, and derive polyhedral bounds on sensitivities of these ODE solutions for the first time. (Sections 4.5.2-4.5.5)

4.2 EXAMPLE

We next proceed with a simple example illustrating our technique.

Running Example A key application of *static analysis of Automatic Differentiation* that we target in this work is for robust sensitivity analysis of ordinary differential equation (ODE) solvers. We note that we are the first to consider such a static analysis. Our running example consists of automatically differentiating through the ODE solver (as in [8]) for a simple ODE. Our ODE is a popular energy balance model from climate science [123, 124], where the global temperature T is a function of time t and Q, R, α, e are the global insolation, heat capacity, albedo and (scaled) emissivity parameters respectively. This ODE is given as:

$$\frac{dT}{dt} = \frac{Q \cdot (1 - \alpha) - e \cdot T^4}{R} \tag{4.1}$$

The program that we will statically analyze solves this ODE numerically, thus obtaining a sequence $[T_0, ..., T_m]$ of temperatures for m time steps. To solve the ODE numerically, we will need the system dynamics function (RHS of Eq. 4.1) which as can be seen, is described by the function $f(T, t, Q, \alpha, e, R) = \frac{Q \cdot (1-\alpha) - e \cdot T^4}{R}$. The program for a simple Euler ODE solver applied to Eq. 4.1, which we will denote as $\texttt{ODESolve}_f$ is given in Fig. 4.1:

Thus by solving the ODE numerically (instead of symbolically) we can automatically differentiate through the numerical ODE solver itself, to get the derivative of the numerical

solver's output with respect to the initial condition inputs, in this case T_0.

Applying AD to `ODESolve` is possible because `ODESolve` is *itself* a differentiable program since it performs only differentiable operations, which are just addition, multiplication (by the step size), as well as the multiplications, division, subtraction, and the (4^{th}) power function found inside of the dynamics f. Automatically differentiating through `ODESolve` will allow us to better understand how sensitive the numerical solution of ODEs are to different initial conditions or parameters (e.g. different values of e or R). Hence we will ultimately compute $\frac{\partial \texttt{ODESolve}_f(Q,\alpha,e,R,t_0,T_0,step\_size,steps)}{\partial T_0}$.

Abstract Sensitivity Analysis. However, our goal is to not just automatically differentiate through `ODESolve` to perform the sensitivity analysis at individual points (as done in [8]), rather our goal is to *abstractly* compute these sensitivities for an *entire range* of initial conditions. Furthermore, since the physical parameters of this climate ODE can vary (e.g. e has dependence on the weather), we want this sensitivity analysis to capture an entire range of feasible parameter values. This abstract sensitivity analysis is performed by abstractly interpreting AD applied to the ODE solver. Hence we abstractly compute $\frac{\partial \texttt{ODESolve}_f(Q,\alpha,e,R,t_0,T_0,step\_size,steps)}{\partial T_0}$ for various ranges of T_0, e and R. This abstract sensitivity analysis can be viewed as a static analysis (using abstract interpretation) of a differentiable program, in this case the `ODESolve` program.

Specification. By performing the AD-based sensitivity analysis abstractly for *ranges* of initial conditions and parameter values, we can ultimately prove properties like the *monotonicity* of the numerical ODE solution with respect to the initial values for *all* values in the given input ranges. Proving monotonicity of the final ODE solution with respect to the initial conditions has been shown to be important for understanding physical processes [125]. Proving monotonicity of the output of this climate model with respect to the initial temperature conditions for a range of values and atmospheric conditions, allows us to prove properties such as the future temperature (after N time steps) being a strictly increasing function of the current temperature (the initial condition), even under a range of different atmospheric conditions. Thus, our goal in this example is to prove monotonicity of the future temperature computed by $\texttt{ODESolve}_f$ with respect to the initial temperature, T_0 after $N = 12$ steps when both T_0 and the atmospheric conditions are known only up to some interval. We will prove this property by certifying that the derivative is always non-zero. While abstract interpretation of AD has been done before [34, 35, 116], we will soon see that those techniques are not precise enough to prove the monotonicity property that we are interested in.

Example Source Code. Since ODE solvers run for a fixed, finite number of iterations, conceptually they can be unrolled. Hence the first step is to unroll the ODE solver program

```
1  T[0].real, T[0].dual = ... // These values are passed in as inputs
2  e.real, e.dual =  ...
3  R.real, R.dual = ...
4  step_size = ... // always treated as fixed scalar constant
5  Q, alpha = ...  // treated as fixed scalar constants for this example
6  C = Q*(1-alpha) // constant propagation for Q and alpha (both are constants)
7
8  i1.real = FourthPower(T[0].real) // Computes Line 3 of ODESolve_f
9  i1.dual = 4*Cube(T[0].real)*T[0].dual    // Chain Rule
10
11 i2.real = e.real*i1.real
12 i2.dual = (e.real*i1.dual)+(e.dual*i1.real)  // Product Rule
13
14 i3.real, i3.dual = C - i2.real, -i2.dual  // Linearity
15
16 i4.real = i3.real/R.real
17 i4.dual = ((i3.dual*R.real) - (i3.real*R.dual)) / Square(R.real)    // Quotient Rule
18
19 T[1].real, T[1].dual = (step_size*i4.real)+T[0].real, (step_size*i4.dual)+T[0].dual
```

Figure 4.2: Unrolled AD source code for differentiating a single iteration of $\texttt{ODESolve}_f$

into straight-line code that we can abstractly interpret. For illustration simplicity, Pasado's abstract AD applied to a single iteration of $\texttt{ODESolve}_f$ is shown in Fig. 4.2, however, Pasado can analyze any fixed, finite number of iterations, though more iterations will diminish the precision. Indeed, the full results of Pasado's analysis after 12 iterations are later shown in Fig. 4.4. Since we are performing AD, we must also keep track of each variable's derivative. For this example, we elect for forward mode AD, hence we store each variable's value in the `real` component and its derivative in the associated `dual` component, as we encode derivatives using the canonical *dual numbers* [5]. Hence, our concrete semantics are first-order version of [35]. As will be seen later, our approach is general enough to support both forward and reverse mode AD. Lines 1-5 correspond to program statements that read in and store the input values for the initial condition T_0, as well as the parameter values for Q, α, e and R.

Chain Rule. On line 9, we compute the derivative of composing the 4^{th} power function with the input variable T_0, stored in variable `i1`. Due to the chain rule, this step also requires multiplying by `T[0].dual`. We compute the derivative of this function application using the chain rule, where in this case $\frac{d}{dT_0} T_0^4 = 4T_0^3$. Between the cube function (`Cube(T[0])`), and the multiplication by `T[0].dual`, there are 3 nonlinear operations (all multiplications) involved in the computation of `i1.dual`. These multiple nonlinear operations pose a direct challenge for the subsequent abstract interpretation, however, as we will see, Pasado is specifically designed to overcome this challenge.

Product Rule. To propagate derivatives through the computation, we now need to compute the derivative of the product of T^4 with e. This step involves computing the product rule for the program variables `i1` and `e`. However, as can be seen on line 12,

the product rule involves 2 nonlinear multiplication operations. Upon computing both the product of T^4 with e (line 11) and its derivative (line 12), we next execute line 14 to compute the difference of this product with $Q \cdot (1 - a)$, which has already been stored in variable `C` in line 6. Because of linearity, the derivative of this difference (line 14) is straightforward to compute. Thus upon completion of line 14, we finally have computed both the numerator of the dynamics function f, and its associated derivative.

Quotient Rule. Having computed the intermediate expression needed for the numerator, we next come to the division operation shown in line 16. To compute the derivative of this division, AD must compute the quotient rule, as shown in line 17. It is important to see that the quotient rule has 4 nonlinear operations, which as mentioned, pose challenges for precise abstract interpretation.

Upon computing the quotient and its derivative (with respect to T_0), we finally compute the value of the next time step, T_1 and its associated derivative in line 19. Thus we now have both the value of the temperature at the next time step as well as the derivative of the temperature at the next time step with respect to the initial condition, which is just $\frac{\partial \texttt{ODESolve}_f(Q,\alpha,e,R,t_0,T_0,step\_size,steps)}{\partial T_0}$. We will next see how to abstractly interpret this AD computation precisely using Pasado.

Precise Abstract Interpretation with Pasado. As observed in the preceding description, there are multiple nonlinear operations involved in computing the derivatives that are stored in the intermediate variables' `dual` components. These nonlinearities present a challenge for precise abstract interpretation, as precise numerical abstract domains like zonotopes and polyhedra were designed for analyzing linear functions and are thus not well suited for handling nonlinear operations [126]. For instance, with the zonotope abstract domain, each nonlinear operation introduces a new noise symbol, thus the large number of nonlinear operations (like multiplications) in AD runs the risk of substantial over-approximation. Existing abstract interpretations for AD [34, 35] suffer from these weaknesses and, as we will later see, produce bounds that are too loose to be useful for our analysis.

For this example, we will use the reduced product of the zonotope domain with intervals to represent our abstraction. Pasado's synthesis method produces precise abstractions for both domains improving the precision of the reduced product. As we will later see, our construction is general and thus could be instantiated with other abstract domains such as the DeepPoly domain or quadratic zonotopes. Since we are using the reduced product of zonotopes with intervals, the abstract program state is shown in purple where $\wr\ \mathrm{Var} = a_0 + \sum_j a_j\epsilon_j,\ l_{Var} = c_l,\ u_{Var} = c_u\ \int$ signifies that variable Var (which could be either a real or dual component) is abstractly represented with affine form $a_0 + \sum_j a_j\epsilon_j$, but also maintains, tighter, refined bounds $[c_l, c_u]$. Here $a_j, c_l, c_u \in \mathbb{R}$ and each $\epsilon_j \in [-1, 1]$ is a noise

1 T[0].real, T[0].dual = ... // **These values are passed in as inputs**
2 ⟅ T[0].real = 337.5 + -62.5ϵ_1, $l_{T[0].real} = 275$, $u_{T[0].real} = 400$ ⟆
3 ⟅ T[0].dual = 1, $l_{T[0].dual} = 1$, $u_{T[0].dual} = 1$ ⟆
4 e.real, e.dual = ...
5 ⟅ e.real = $4.25 \cdot 10^{-8}$ + $-8.50 \cdot 10^{-9}\epsilon_3$, $l_{e.real} = 3.40 \cdot 10^{-8}$, $u_{e.real} = 5.1 \cdot 10^{-8}$ ⟆
6 ⟅ e.dual = 0, $l_{e.dual} = 0$, $u_{e.dual} = 0$ ⟆
7 R.real, R.dual = ...
8 ⟅ R.real = 2.8 + -0.15ϵ_5, $l_{R.real} = 2.65$, $u_{R.real} = 2.95$ ⟆
9 ⟅ R.dual = 0, $l_{R.dual} = 0$, $u_{R.dual} = 0$ ⟆
10 step_size = ... // **always treated as fixed scalar constant**
11 ⟅ step_size = 0.025, $l_{step\_size} = 0.025$, $u_{step\_size} = 0.025$ ⟆
12 Q, alpha = ... // **treated as fixed scalar constants for this example**
13 ⟅ Q = 342, $l_Q = 342$, $u_Q = 342$ ⟆
14 ⟅ alpha = 0.3, $l_{alpha} = 0.3$, $u_{alpha} = 0.3$ ⟆
15 C = Q*(1−alpha) // **constant propagation for Q and alpha (both are constants)**
16 ⟅ C = 239.4, $l_C = 239.4$, $u_C = 239.4$ ⟆
17
18 i1.real = **FourthPower**(T[0].real) // **Computes Line 3 of** $\texttt{ODESolve}_f$
19 ⟅ i1.real = 12974633789.062 + -9610839843.75ϵ_1 + 889892578.12ϵ_7 + 2124633789.062ϵ_8,
20 $l_{i1.real} = 5719140625$, $u_{i1.real} = 25600000000$ ⟆
21 i1.dual = 4***Cube**(T[0].real)*T[0].dual // **Chain Rule**
22 ⟅ i1.dual = 161798882.57 + -86167091.83ϵ_1 + 8034025.58ϵ_9, $l_{i1.dual} = 83187500$, $u_{i1.dual} = 256000000$ ⟆
23
24 i2.real = e.real*i1.real
25 ⟅ i2.real = 551.78 + -408.72ϵ_1 + -110.35ϵ_3 + 37.84ϵ_7 + 90.35ϵ_8 + 107.38ϵ_{10}, $l_{i2.real} = 194.57$, $u_{i2.real} = 1306.45$ ⟆
26 i2.dual = (e.real*i1.dual)+(e.dual*i1.real) // **Product Rule**
27 ⟅ i2.dual = 6.88 + -3.66ϵ_1 + 2.94e-15ϵ_7 + 7.03e-15ϵ_8 + 0.34ϵ_9 + 2.17ϵ_{11}, $l_{i2.dual} = 2.83$, $u_{i2.dual} = 13.06$ ⟆
28
29 i3.real, i3.dual = C − i2.real, − i2.dual // **Linearity**
30 ⟅ i3.real = -312.38 + 408.72ϵ_1 + 110.35ϵ_3 + -37.84ϵ_7 + -90.35ϵ_8 + -107.38ϵ_{10}, $l_{i3.real} = -1067.05$, $u_{i3.real} = 44.82$ ⟆
31 ⟅ i3.dual = -6.88 + 3.66ϵ_1 + -2.94e-15ϵ_7 + -7.03e-15ϵ_8 + -0.34ϵ_9 + -2.17ϵ_{11}, $l_{i3.dual} = -13.06$, $u_{i3.dual} = -2.83$ ⟆
32
33 i4.real = i3.real/R.real
34 ⟅ i4.real = -111.9 + 146.39ϵ_1 + 39.52ϵ_3 + -5.38ϵ_5 + -13.55ϵ_7 + -32.36ϵ_8 + -38.46ϵ_{10} + -0.61ϵ_{12} + 14.48ϵ_{13},
35 $l_{i4.real} = -402.66$, $u_{i4.real} = 16.91$ ⟆
36 i4.dual = ((i3.dual*R.real)−(i3.real*R.dual))/**Square**(R.real) // **Quotient Rule**
37 ⟅ i4.dual = -2.47 + 1.31ϵ_1 + -0.0009ϵ_3 + -0.15ϵ_5 + 0.0003ϵ_7 + 0.0007ϵ_8 + 0.0009ϵ_{10} + -0.12ϵ_9 + -0.78ϵ_{11} + 0.11ϵ_{14},
38 $l_{i4.dual} = -4.93$, $u_{i4.dual} = -0.96$⟆
39
40 T[1].real, T[1].dual = (step_size*i4.real)+T[0].real, (step_size*i4.dual)+T[0].dual
41 ⟅ T[1].real = 334.7 + -58.84ϵ_1 + 0.99ϵ_3 + -0.13ϵ_5 + -0.33ϵ_7 + -0.81ϵ_8 + -0.96ϵ_{10} + -0.015ϵ_{12} + 0.36ϵ_{13},
42 $l_{T[1].real} = 272.25$, $u_{T[1].real} = 397.15$⟆
43 ⟅ T[1].dual = 0.94 + 0.03ϵ_1 + -2.01e-5ϵ_3 + -0.004ϵ_5 + 6.88e-6ϵ_7 + 1.64e-5ϵ_8 + -0.003ϵ_9 + 1.95e-5ϵ_{10} + -0.02ϵ_{11} + 0.002ϵ_{14},
44 $l_{T[1].dual} = 0.8767$, $u_{T[1].real} = 0.9760$⟆

Figure 4.3: Abstractly Interpreted AD code to differentiate one iteration of $\texttt{ODESolve}_f$

symbol, which intuitively is a term in a first-order symbolic polynomial.

Fig. 4.3 shows the abstract interpretation of the original source code. In this example we want to abstractly compute guaranteed bounds on $\frac{\partial \texttt{ODESolve}_f(Q,\alpha,e,R,t_0,T_0,step\_size,steps)}{\partial T_0}$ when $T_0 \in [275, 400]$, $R \in [2.65, 2.95]$, $e \in [0.6\sigma, 0.9\sigma]$, $\alpha = 0.3$, $Q = 342$, $step\_size = 0.025$ and $steps = 1$. For the scaled emmisivity e, the value of σ is the Stefan-Boltzmann constant $5.67 \cdot 10^{-8}$. Thus to specify these bounds over the input, we initialize the abstract program state as shown in lines 2-16.

Synthesized Abstraction for Chain Rule. The first computation we perform is the computation of the quartic power function on line 18 and its derivative on line 21. As our contribution is focused solely on constructing synthesized abstraction for just the derivative

terms, we use the standard interval and zonotope multiplication abstract transformers for the quartic function on line 18. It is important to note that after abstractly executing line 18 but before executing line 21, the zonotope abstract state will have 8 noise symbols ϵ_{1-8}. To abstractly interpret line 21, we use Pasado's synthesized abstract transformer instead of the standard zonotope multiplication abstract transformer. The first step of Pasado's synthesis procedure is to solve a linear regression problem for $A, B, C \in \mathbb{R}$ such that within the box $[l_{T[0].real}, u_{T[0].real}] \times [l_{T[0].dual}, u_{T[0].dual}] \subset \mathbb{R}^2$:

$$A \cdot (T[0].real) + B \cdot (T[0].dual) + C \approx 4 \cdot (T[0].real)^3 \cdot (T[0].dual) \tag{4.2}$$

Pasado performs this step by sampling uniformly-spaced points in the box, which here is $[275, 400] \times [1, 1]$. For each sampled point x_i, y_i we evaluate $4x_i^3 \cdot y_i$ which is used as the "ground-truth" for the linear regression. For this example linear regression produces the result: $A = 1378673.47$, $B = 0$, $C = -304748724.48$. Hence the affine form will be $\widehat{i1.dual} = A\widehat{T[0].real} + B\widehat{T[0].dual} + C + D\epsilon_{new}$, where ϵ_{new} is a single new noise symbol and the $\widehat{hat}$ notation denotes the variable's affine form stored in the abstract state. The next step is computing D by soundly bounding the error between this linear approximation and the true function $4 \cdot (T[0].real)^3 \cdot (T[0].dual)$, as shown in Eq. 4.3

$$D = \max_{x \in [275,400], y \in [1,1]} |4x^3 \cdot y - (Ax + By + C)| \tag{4.3}$$

Noting that $[l_{T[0].real}, u_{T[0].real}] = [275, 400]$ and $[l_{T[0].dual}, u_{T[0].dual}] = [1, 1]$. While at first glance this may seem like a difficult nonconvex, multivariable optimization problem a core contribution of Pasado is proving that we can reduce this multivariable optimization problem to simpler 1D optimization subproblems as well as just checking the corner points. Hence we can actually find the *exact* value of D by solving for roots of $12(T[0].real)^2 \cdot l_{T[0].dual} - A = 0$ and $12(T[0].real)^2 \cdot u_{T[0].dual} - A = 0$, checking those roots and additionally checking the corner points $\{l_{T[0].real}, u_{T[0].real}\} \times \{l_{T[0].dual}, u_{T[0].dual}\}$. The coefficient D will be used as the new noise symbol's magnitude to ensure the linear approximation is still a sound zonotopic enclosure.

An important detail is that because of our technique, the abstract transformer for the composite expression `4*Cube(T[0].real)*T[0].dual` only introduces a *single* new noise symbol ($D\epsilon_{new}$), hence why there are only nine noise symbols (ϵ_{1-9}) as shown in line 22. This insight is critical, as evaluating the same expression with the standard zonotope multiplications would introduce three noise symbols instead of one. It is known that having fewer noise symbols can lead to computational savings [127], and this reduction helps us offset some of the cost of synthesizing transformers. Furthermore, we can use the *same* technique

to solve the simpler optimization problems $\min_{x\in[275,400],y\in[1,1]} 4x^3 \cdot y$ and $\max_{x\in[275,400],y\in[1,1]} 4x^3 \cdot y$, which can be used to give us refined interval lower and upper bounds, which in this case are $[8.318 \cdot 10^7, 2.56 \cdot 10^8]$. While these bounds appear large, they will eventually be scaled to a tight range upon multiplication with e since $e < 10^{-7}$. Further, while in this case these interval bounds are identical to those obtained via standard interval arithmetic, for other functions, our optimally solved bounds are often much more precise. We take the intersection of the bounding box of the affine form which is $[6.759 \cdot 10^7, 2.56 \cdot 10^8]$ with $[8.318 \cdot 10^7, 2.56 \cdot 10^8]$ to get the final refined lower and upper bounds. In this case the refined interval bounds obtained by solving the two simpler optimization problems are *strictly* tighter than the affine form's bounding box (hence the result is still $[8.318 \cdot 10^7, 2.56 \cdot 10^8]$), however, this need not always be true, hence why we take the intersection. A key benefit of our approach is that because our abstract transformers use interval bounds to solve their optimization problem they can *directly* benefit from having these refined bounds which directly leads to more precise affine forms. In contrast, as noted in [127], the standard zonotope transformers for multiplication *cannot* take advantage of the refined bounds to improve precision of the resulting affine form.

Synthesized Abstraction for Product Rule. Pasado's next step is to compute the product in line 24, as well as the derivative using the product rule in line 26. As mentioned, Pasado is focused solely on developing precise abstractions for the derivatives, hence the multiplication of the real parts in line 24 uses the standard interval and zonotope domain multiplications. Similarly to the chain rule, our synthesized abstract transformer will solve a linear regression problem for new $A, B, C, D, E \in \mathbb{R}$ such that in the region $[l_{e.real}, u_{real}] \times [l_{e.dual}, u_{e.dual}] \times [l_{i1.real}, u_{i1.real}] \times [l_{i1.dual}, u_{i1.dual}]$ the following holds:

$$A \cdot (e.real) + B \cdot (e.dual) + C \cdot (i1.real) + D \cdot (i1.dual) + E \approx e.real \cdot i1.dual + e.dual \cdot i1.real \quad (4.4)$$

Hence the resulting affine form is $\widehat{i2.dual} = A(\widehat{e.real}) + B(\widehat{e.dual}) + C(\widehat{i1.real}) + D(\widehat{i1.dual}) + E + F\epsilon_{new}$ In this example, the linear regression finds the new coefficients should be $A = 0$, $B = -6 \cdot 10^{-24}$, $C = 3 \cdot 10^{-24}$, $D = 4.25 \cdot 10^{-8}$ and $E = -5 \cdot 10^{-14}$. Our abstraction must also solve for the maximum error F. Among our key contributions is proving that for the product rule, this error can be obtained by just enumerating the 2^4 corner points. Hence in Eq. 4.5, we compute F as:

$$F = \max_{(x_1,y_1,x_2,y_2)\in\ Corners} |(x_1 \cdot y_2) + (x_2 \cdot y_1) - (Ax_1 + By_1 + Cx_2 + Dy_2 + E)| \quad (4.5)$$

where $Corners$ is the set $\{l_{e.real}, u_{e.real}\} \times \{l_{e.dual}, u_{e.dual}\} \times \{l_{i1.real}, u_{i1.real}\} \times \{l_{i1.dual}, u_{i1dual}\}$. In this example, $F = 2.17$. While our example has two multiplications on line 26, Pasado introduces only a single noise symbol ($F\epsilon_{new}$) instead of two as the standard zonotope multiplication would. Furthermore, we can solve a simpler version of the previous optimization problem for direct interval lower and upper bounds using the same approach (enumerating over the corners) and then take the intersection of those with the bounding box of the affine form. Performing this procedure ultimately gives us $i2.dual$'s refined bounds of $[2.83, 13.06]$ as shown on line 27.

Synthesized Abstraction for Quotient Rule. Upon computing the products, computing the affine transformations on line 29 is straightforward. Hence the abstract interpreter next proceeds to line 33 to compute the quotient. The derivative is computed via quotient rule on line 36. As before, Pasado uses linear regression to solve for coefficients $A, B, C, D, E \in \mathbb{R}$ such that:

$$A \cdot (i3.real) + B \cdot (i3.dual) + C \cdot (R.real) + D \cdot (R.dual) + E \approx \frac{(R.real \cdot i3.dual) - (i3.real \cdot R.dual)}{R.real^2} \tag{4.6}$$

Here the values are $A = -4.2 \cdot 10^{-6}$, $B = 0.35$, $C = 1.01$, $D = 6.5 \cdot 10^{-6}$, and $E = -2.84$, where the resulting affine form is $\widehat{i4.dual} = A(\widehat{i3.real}) + B(\widehat{i3.dual}) + C(\widehat{R.real}) + D(\widehat{R.dual}) + E + F\epsilon_{new}$. Solving for the maximum error, F, is more involved, as we must solve:

$$F = \max_{\substack{x_1 \in [l_{i3.real}, u_{i3.real}], y_1 \in [l_{i3.dual}, u_{i3.dual}] \\ x_2 \in [l_{R.real}, u_{R.real}], y_2 \in [l_{R.dual}, u_{R.dual}]}} \left| \frac{(x_2 \cdot y_1) - (x_1 \cdot y_2)}{x_2^2} - (Ax_1 + By_1 + Cx_2 + Dy_2 + E) \right| \tag{4.7}$$

The solution to this optimization requires Pasado to solve for roots of a cubic equation, as we will see. However, Pasado can likewise use that same technique to directly solve for optimal interval bounds: $\min \frac{(x_2 \cdot y_1) - (x_1 \cdot y_2)}{x_2^2}$ and $\max \frac{(x_2 \cdot y_1) - (x_1 \cdot y_2)}{x_2^2}$ over the same 4D region as above. We ultimately obtain the affine form shown in line 43 and the refined bounds $[0.8767, 0.9760]$ in line 44.

Results. Abstractly interpreting the computation using Pasado's synthesized transformers offers notable precision gains over standard interval and zonotope abstract transformers. As mentioned, Pasado's bounds on the derivative after one iteration are: $[0.8767, 0.9760]$, whereas, the respective bounds computed with intervals are $[0.862, 0.978]$ and with zonotopes are $[0.868, 1.013]$. While Figs. 4.2 and 4.3 show only a single iteration, the benefits of Pasado compound over multiple iterations.

Fig. 4.4 shows the bounds computed for up to $N = 12$ iterations by both Pasado and the

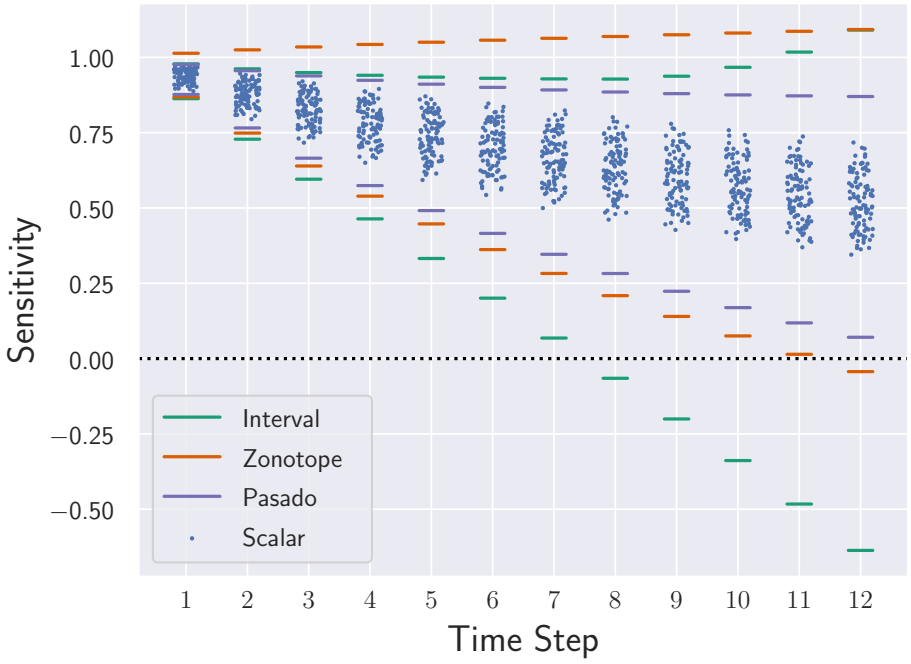


Figure 4.4: Comparison of Pasado with interval and zonotope abstract AD for bounding derivative-based sensitivities with respect to T_0 for 12 full iterations of $\texttt{ODESolve}_f$. Each region between line segments of the same color represents the interval bound computed by that respective method's corresponding abstract AD. The dots represent the sensitivities evaluated concretely, using scalar points sampled from the input intervals.

interval and zonotope AD baselines. This plot shows how Pasado's improvement compounds, and how after 12 iterations (full code not shown), Pasado's derivative bounds stay provably positive, and thus can prove monotonicity with respect to T_0, whereas interval and zonotope abstract AD cannot.

4.3 PRELIMINARIES

We now detail the necessary preliminaries for describing both automatic differentiation as well as abstract interpretation. We also detail some of the key mathematical requirements for Pasado.

4.3.1 Automatic Differentiation Implementation

Forward-Mode AD In forward-mode AD, one computes both the primal (the original program) as well as the tangent derivatives simultaneously in a single forward pass. For functions $f : \mathbb{R}^m \to \mathbb{R}^n$, computing the full Jacobian via forward mode requires m passes,

hence forward mode is more efficient when $m < n$. Since the computation of both the original primal program and the derivatives are interwoven into a single forward pass and thus happens simultaneously, one must designate separate variables for the primal (the real part) and the associated derivatives (the dual part). This separation of variables into disjoint sets is canonically implemented with *dual numbers* [5], where the real component of the dual number encodes the variables of the primal, and the dual component of the dual number encodes the associated derivatives. The standard rules of calculus – chain rule, product rule and quotient rule can be encoded by overloading the respective arithmetic operation for dual numbers. For a given variable x, we will denote the real part as $x.real$ and the dual part storing the associated derivative as $x.dual$.

Reverse-Mode AD In reverse mode AD, one first computes the primal and then back-propagates the derivatives from the output variable back to the input variables [5]. For functions $f : \mathbb{R}^m \to \mathbb{R}^n$, computing the full Jacobian via reverse mode requires n passes, hence reverse mode is more efficient when $m > n$. Unlike with forward-mode AD, in reverse-mode AD the original program (the primal) is computed in its entirety before even a single derivative is computed. However, one still needs a separate set of variables to store the derivatives. Following convention, we will simply let x denote a given variable in the primal and $\overline{x}$ denote the adjoint derivative for x that is computed in the backward pass.

4.3.2 Abstract Interpretation

We now describe the necessary preliminaries of Abstract Interpretation. Abstract Interpretation [1] is a framework for soundly over-approximating the set of possible executions of a program. For Pasado's analysis, we require the following:

1. A numerical abstract domain, $\mathbb{A}$ which may be either Intervals or any abstract domain that can represent linear transformations exactly (e.g. Zonotopes, Quadratic Zonotopes, Polyhedra). $\mathbb{A}$ can also be a reduced product of those aforementioned domains.

2. A concretization function $\gamma : \mathbb{A} \to \mathcal{P}(\mathbb{R}^n)$ that maps an abstract element in the Abstract Domain to sets of AD program states (which are just tuples of n real numbers.).

3. A bounding box function $Range : \mathbb{A} \to \mathbb{R}^n \times \mathbb{R}^n$ that takes an abstract element describing sets of AD program states of n variables and returns the bounds on each variable.

4. Sound abstract transformers, $T_f^\sharp : \mathbb{A} \to \mathbb{A}$, for each univariate nonlinear function $f : \mathbb{R} \to \mathbb{R}$. The functions we consider are $\{\log(x), \exp(x), \sqrt{x}, x^2, x^3, x^4, \tanh(x), \sigma(x), \mathit{NormalCDF}(x)\}$.

5. Sound abstract transformers, $T_{op}^\sharp : \mathbb{A} \to \mathbb{A}$, for each binary operation $op \in \{+, -, *, /\}$.

Additionally, unique to our approach is that we will also require a guaranteed root solver for the second derivative of each nonlinear function f listed in (4), so that we can solve for all $x^* \in [l, u]$ (or certify that none exist inside $[l, u]$) such that $f''(x^*) = C$ for any given C. For this root-solving, one may use a verified root finding technique like Bisection or a verified Newton's method, however for many of the functions such that f''^{-1} has an analytical formula, we can use the analytical formula to solve for x^* directly. For instance when $f(x) = \exp(x)$, we know $f''^{-1}(x) = \log(x)$.

For our purposes we will use the reduced product of the Zonotope domain with the Interval domain as this combination is essentially a (restricted) polyhedral domain that is more expressive than standard zonotopes, however as described in Theorem 4.5, our construction would equally work for other domains which can symbolically express linear relationships exactly such as quadratic zonotopes, or the DeepPoly domain.

For our purposes an abstract state $a \in \mathbb{A}$ will map each variable x_i to both an affine form, $a[x_i].\widehat{x_i} = x_{i0} + \sum_j c_j \epsilon_j$ and an interval $a[x_i].[l_{x_i}, u_{x_i}]$. The bounding box function returns the intersection of the associated interval with the bound on the affine form, hence $\mathit{Range}(a[x_i]) = [x_{i0} - \sum_j |c_j|, x_{i0} + \sum_j |c_j|] \sqcap a[x_i].[l_{x_i}, u_{x_i}]$. Since we take the reduced product, our concretization function uses both the standard interval and zonotope concretization functions $\gamma_{Int}, \gamma_{Zono}$ [101] and is given in Eq. 4.8 as:

$$\gamma(a) = \left\{ (x_1, ..., x_n) \in \mathbb{R}^n : (x_1, ..., x_n) \in \gamma_{Zono}\left(\bigwedge_{j=1}^{n} a[x_j].\widehat{x}_j\right) \cap \gamma_{Int}\left(\bigwedge_{j=1}^{m} a[x_j].[l_{x_j}, u_{x_j}]\right) \right\} \tag{4.8}$$

As mentioned in Section 4.2, for analyzing the real (non-derivative) part of the program we will use the given domain's standard abstract transformers required in (4) and (5) hence Pasado only focuses on synthesizing abstractions for the derivative terms to improve precision. Further, since addition can already be done exactly with zonotopes, Pasado will use the standard transformers for addition, even in the derivative computations.

4.4 SYNTHESIZING PRECISE AD STATIC ANALYZERS

We now present Pasado, our technique for synthesizing precise abstract transformers, specialized for AD. Pasado's technique allows us to synthesize precise abstract transformers for the Chain Rule, Product Rule and Quotient Rule of Calculus, which we will denote $T^\sharp_{C_f}$, $T^\sharp_P$, $T^\sharp_Q$: $\mathbb{A} \to \mathbb{A}$. Pasado uses standard abstract transformers (e.g., $T^\sharp_f$, $T^\sharp_{op}$ required in Section 4.3.2) to abstract the primal and $T^\sharp_{C_f}$, $T^\sharp_P$, $T^\sharp_Q$ to abstract the derivative computation.

Since both forward-mode AD and reverse-mode AD ultimately use these same rules, we can synthesize precise abstractions for each of the core operations for either mode of AD. The only difference between AD modes is the order of application, for instance in forward mode for $a \in \mathbb{A}$, the application order would be $T^\sharp_P(T^\sharp_*(T^\sharp_{C_f}(T^\sharp_f(a))))$ whereas for reverse mode the order would be $T^\sharp_P(T^\sharp_{C_f}(T^\sharp_*(T^\sharp_f(a))))$ as the entire primal must be abstracted before any derivatives can be. While there are other rules of calculus, (e.g., generalized power rule), for which our techniques are inapplicable (due to differences in the Hessians), these three rules comprise the majority of nonlinear AD operations and thus are the most important.

Pasado's abstract transformer synthesis involves a combination of linear regression at uniformly spaced points and solving a nonlinear optimization problem to ensure soundness. For tractability, we limit the number of sampled grid points used in the linear regression (which trades off precision).

4.4.1 Chain Rule Synthesized Transformer

The first rule of Calculus for which we want to synthesize a precise abstraction is the Chain Rule. For functions $f, g : \mathbb{R} \to \mathbb{R}$, the chain rule is mathematically given as:

$$f(g(u))' = f'(g(u)) \cdot g'(u) \tag{4.9}$$

Forward-Mode Chain Rule. In forward-mode AD this rule is implemented via:

```
1 z.real = f(x.real);
2 z.dual = f'(x.real)*x.dual---//chain-rule
```

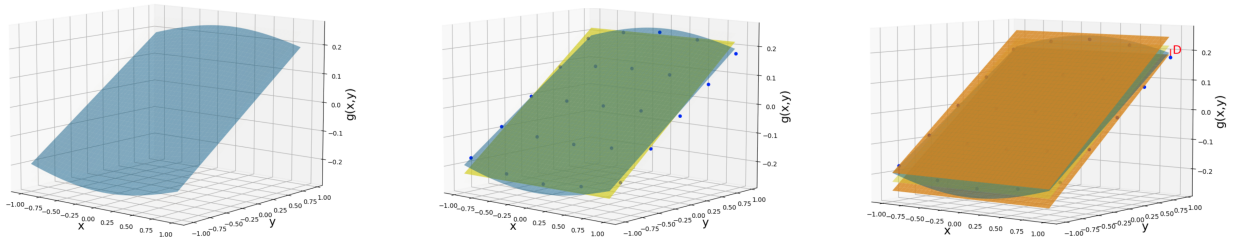
Figure 4.5: Forward Mode Chain Rule Pattern

where intuitively, $x.real = g(u)$, $x.dual = g'(u)$, $z.real = f(g(u))$, and $z.dual = f(g(u))'$.

Reverse-Mode Chain Rule. Likewise in reverse-mode AD, this rule is implemented as:

```
1 z = f(x); ...
2 z̄ = ...;
3 x̄ += f'(x)*z̄;   //chain rule
```

Figure 4.6: Reverse Mode Chain Rule Pattern



(a) (b) (c)

Figure 4.7: Visualization of Pasado's abstract transformer synthesis for the Chain rule pattern $g(x, y) = \sigma(x) \cdot (1 - \sigma(x)) \cdot y$ on $[-1, 1] \times [-1, 1]$. In (a), the blue surface represents $g(x, y)$. In (b), the blue dots on the blue surface represent evaluations of $g(x, y)$ at grid sampled points. The yellow hyperplane in (b) is computed by performing linear regression with these blue points and has equation $Ax+By+C$. In (c), the red lines show the difference between $g(x, y)$ and the plane and D represents the maximum such difference. The lower and upper orange planes in (c) are the enclosing linear bounds given by $Ax + By + C \pm D$. The enclosing bounds are parallel for the Zonotope domain and here the maximum difference D occurs at a corner point.

where the "..." at the end of the first line represents the break between the end of the *primal* part of the differentiable program and the start of the *adjoint* part of the same differentiable program, which computes all the derivatives (e.g. $\overline{z}$, $\overline{x}$).

Chain Rule Abstraction Pattern. Based on these implementations, the main expression, present in both forward and reverse AD, for which we want to synthesize an abstract transformer, $T^{\sharp}_{C_f}$, is:

$$g(x, y) = f'(x) \cdot y \tag{4.10}$$

The benefit of synthesizing an abstraction for this chain rule pattern is that this pattern could have *multiple* nonlinear operations. For instance, if $f(x) = \sigma(x)$, then $f'(x) = \sigma(x) \cdot (1 - \sigma(x))$, which has a nonlinear multiplication, in addition to the nonlinear multiplication with y. Thus naively composing the abstract transformers for each nonlinear operation e.g., $T^{\sharp}_{*}(T^{\sharp}_{*}(T^{\sharp}_{-}(T^{\sharp}_{\sigma}(a))))$ as in [35] can lead to imprecision. Particularly when using zonotopes, each of those nonlinear operations introduces a new noise symbol which adds additional over-approximation. In contrast, $T^{\sharp}_{C_f}$ introduces only a single noise symbol for the entire chain rule derivative expression.

Abstraction. We now present how to abstract the chain rule pattern. Algorithm 4.1 presents the abstract transformer $T^{\sharp}_{C_f}$ and Fig. 4.7 presents a geometric intuition. The core idea is to sample uniformly spaced points that lie within the range of the input intervals and then solve a linear regression problem to find the best linear approximation of $f'(x) \cdot y$

Algorithm 4.1: Chain Rule Abstract Transformer $T^{\sharp}_{C_f}$

Input: Abstract state a where $\widehat{x} = a[x].\widehat{x}$ and $\widehat{y} = a[y].\widehat{y}$
$l_x, u_x \leftarrow Range(a[x])$
$l_y, u_y \leftarrow Range(a[y])$
$grid \leftarrow GridSample([l_x, u_x] \times [l_y, u_y])$
$pts \leftarrow \{f'(x) \cdot y : (x, y) \in grid\}$
$A, B, C \leftarrow LinearRegression(grid, pts)$
if $A = 0$ then $A \leftarrow A + \delta$
$D \leftarrow \max_{x \in [l_x, u_x], y \in [l_y, u_y]} |f'(x) \cdot y - (Ax + By + C)|$
$l, u \leftarrow \min_{x \in [l_x, u_x], y \in [l_y, u_y]} f'(x) \cdot y, \max_{x \in [l_x, u_x], y \in [l_y, u_y]} f'(x) \cdot y$
return $A\widehat{x} + B\widehat{y} + C + D\epsilon_{new}\ ,\ [l, u]$

at those points. However, the most critical step for proving soundness is solving a challenging multidimensional, nonconvex optimization problem, to soundly enclose the linear approximation, which we next describe. The soundness of formally proven in Theorem 4.1.

Optimization Problem. The core technical difficulty of the Chain Rule abstract transformer lies in solving the following equation for the maximum deviation between the linear approximation $(Ax + By + C)$ and the function $f'(x) \cdot y$ itself (example shown in Fig. 4.7). This maximum deviation is needed to obtain the tightest enclosure around the linear approximation $(Ax + By + C)$ such that this enclosure still provably contains the range of $f'(x) \cdot y$. This deviation D is computed as:

$$D = \max_{x \in [l_x, u_x], y \in [l_y, u_y]} |f'(x) \cdot y - (Ax + By + C)| \tag{4.11}$$

Pasado reduces this multivariate, non-convex optimization problem into two simpler univariate problems as well as simply checking the four corner points: $\{l_x, u_x\} \times \{l_y, u_y\}$ (we provide a full explanation in Section 4.4.4). For the correctness of our proof which is subsequently shown in Theorem 4.1, it is a technical requirement that $A \neq 0$. If linear regression obtains $A = 0$, we perturb A by a small quantity, $\delta < 10^{-9}$. To solve the two univariate optimization problems, we compute all $x^* \in [l_x, u_x]$ such that $f''(x^*) = \frac{A}{l_y}$ and all $x^{**} \in [l_x, u_x]$ such that $f''(x^{**}) = \frac{A}{u_y}$. Thus we must also examine the points (x^*, l_y) and (x^{**}, u_y). We can solve for all x^* and x^{**} using the guaranteed root solver that we required in Section 4.3.2. Hence the optimization problem ultimately reduces to:

$$D = \max_{(x,y) \in \left(\{l_{x_1}, u_{x_1}\} \times \{l_{y_1}, u_{y_1}\}\right) \cup \{(x^*, l_y), (x^{**}, u_y)\}} |f'(x) \cdot y - (Ax + By + C)| \tag{4.12}$$

While inspired by [109], our proof technique is more general as we can handle *any* f *satisfying the properties of Section 4.3.2.* The generality of our approach also stems from expanding this proof technique to other patterns arising from AD. We also show how to adapt this proof to obtain precise interval domain transformers. Indeed, the key benefit is that we can use virtually the same proof to get the *exact* lower and upper bounds of $f'(x) \cdot y$ for the given input intervals. Hence we can compute optimal lower and upper bounds, l and u, as follows:

$$l = \min_{(x,y) \in \left(\{l_{x_1}, u_{x_1}\} \times \{l_{y_1}, u_{y_1}\}\right) \cup \{(x^*, l_y), (x^{**}, u_y)\}} f'(x) \cdot y \tag{4.13}$$

$$u = \max_{(x,y) \in \left(\{l_{x_1}, u_{x_1}\} \times \{l_{y_1}, u_{y_1}\}\right) \cup \{(x^*, l_y), (x^{**}, u_y)\}} f'(x) \cdot y \tag{4.14}$$

The core benefit of having both zonotope affine forms and separately computed interval lower and upper bounds is that not only does the same proof strategy give us sound abstractions for both domains, but by taking their reduced product, we can always use the interval results to refine the zonotope, as in [128] to enhance precision.

4.4.2 Product Rule Synthesized Transformer

The next rule of calculus for which we wish to synthesize a precise abstract transformer, $T_P^\sharp$ is the Product Rule. For functions $f, g : \mathbb{R} \to \mathbb{R}$, the product rule is mathematically given as:

$$(f(u) \cdot g(u))' = f'(u) \cdot g(u) + f(u) \cdot g'(u) \tag{4.15}$$

Forward-Mode Product Rule. The forward-mode product rule is implemented as:

```
1 z.real = x.real*y.real;
2 z.dual = (x.dual*y.real)+(x.real*y.dual)  //product rule
```

Figure 4.8: Forward Mode Product Rule Pattern

where intuitively, $x.real = f(u)$, $y.real = g(u)$, $x.dual = f'(u)$, and $y.dual = g'(u)$. It is important to note that the computation of $z.dual$ involves 2 nonlinear multiplications.

Reverse-Mode Product Rule. Similarly, in reverse-mode AD, the product rule is encoded as:

```
1 z = x*y; ...
2 z̄ = ...;
3 x̄ += y*z̄;  //derivative of product wrt x
4 ȳ += x*z̄;  //derivative of product wrt y
```

Figure 4.9: Reverse Mode Product Rule Pattern

Product Rule Abstraction Pattern. Based on the product rule implementations shown above, the computational pattern for which we want to synthesize an abstraction is:

$$g(x_1, y_1, x_2, y_2) = (x_1 \cdot y_2) + (x_2 \cdot y_1) \tag{4.16}$$

In the event that some of the arguments are zero, this pattern could involve a single multiplication. Conversely, when all arguments are nonzero, this pattern entails two nonlinear multiplications. In particular, the instantiation of this abstraction for reverse-mode AD can be seen as a special case whereby we set the first argument x_1 to 0.

Abstraction. We now present Pasado's synthesis technique for the product rule abstract transformer, $T_P^\sharp$ in Algorithm 4.2. As with the chain rule pattern, we synthesize the coefficients by solving a linear regression, and a tractable, but nonconvex optimization problem.

Algorithm 4.2: Product Rule Abstract Transformer $T_P^\sharp$

Input: Abstract state a where $\widehat{x_1} = a[x_1].\widehat{x_1}$, $\widehat{y_1} = a[y_1].\widehat{y_1}$, $\widehat{x_2} = a[x_2].\widehat{x_2}$, and $\widehat{y_2} = a[y_2].\widehat{y_2}$

$l_{x_1}, u_{x_1} \leftarrow \mathit{Range}(a[x_1])$ and $l_{y_1}, u_{y_1} \leftarrow \mathit{Range}(a[y_1])$

$l_{x_2}, u_{x_2} \leftarrow \mathit{Range}(a[x_2])$ and $l_{y_2}, u_{y_2} \leftarrow \mathit{Range}(a[y_2])$

$\mathit{grid} \leftarrow \mathit{GridSample}([l_{x_1}, u_{x_1}] \times [l_{y_1}, u_{y_1}] \times [l_{x_2}, u_{x_2}] \times [l_{y_2}, u_{y_2}])$

$\mathit{pts} \leftarrow \{(x_1 \cdot y_2) + (x_2 \cdot y_1) : (x_1, y_1, x_2, y_2) \in \mathit{grid}\}$

$A, B, C, D, E \leftarrow \mathit{LinearRegression}(\mathit{grid}, \mathit{pts})$

$$F \leftarrow \max_{\substack{x_1 \in [l_{x_1}, u_{x_1}], y_1 \in [l_{y_1}, u_{y_1}] \\ x_2 \in [l_{x_2}, u_{x_2}], y_2 \in [l_{y_2}, u_{y_2}]}} |(x_1 \cdot y_2) + (x_2 \cdot y_1) - (Ax_1 + By_1 + Cx_2 + Dy_2 + E)|$$

$$l, u \leftarrow \min_{\substack{x_1 \in [l_{x_1}, u_{x_1}], y_1 \in [l_{y_1}, u_{y_1}] \\ x_2 \in [l_{x_2}, u_{x_2}], y_2 \in [l_{y_2}, u_{y_2}]}} (x_1 \cdot y_2) + (x_2 \cdot y_1), \max_{\substack{x_1 \in [l_{x_1}, u_{x_1}], y_1 \in [l_{y_1}, u_{y_1}] \\ x_2 \in [l_{x_2}, u_{x_2}], y_2 \in [l_{y_2}, u_{y_2}]}} (x_1 \cdot y_2) + (x_2 \cdot y_1)$$

return $A\widehat{x_1} + B\widehat{y_1} + C\widehat{x_2} + D\widehat{y_2} + E + F\epsilon_{new},\ [l, u]$

There is another key benefit to Pasado's product rule abstract transformer. The standard zonotope multiplication operates directly on the coefficients of the input affine forms. As a result, the standard abstract transformer cannot leverage tighter ranges on the inputs, as these ranges are never taken into account. Thus, when performing the two multiplications of the product rule with regular zonotopes, the ability to leverage improved precision from a reduced product is limited. In contrast, since Pasado's synthesized abstraction *directly* incorporates the bounds on the input, Pasado can immediately benefit from the improved precision that a reduced product with intervals offers.

Optimization Problem. For the product rule abstraction, we must solve the following

4D nonlinear, nonconvex optimization problem:

$$F = \max_{\substack{x_1 \in [l_{x_1}, u_{x_1}], y_1 \in [l_{y_1}, u_{y_1}] \\ x_2 \in [l_{x_2}, u_{x_2}], y_2 \in [l_{y_2}, u_{y_2}]}} |(x_1 \cdot y_2) + (x_2 \cdot y_1) - (Ax_1 + By_1 + Cx_2 + Dy_2 + E)| \tag{4.17}$$

The benefit of this optimization problem is that we only need to check the 2^4 corner points, i.e., $\{l_{x_1}, u_{x_1}\} \times \{l_{y_1}, u_{y_1}\} \times \{l_{x_2}, u_{x_2}\} \times \{l_{y_2}, u_{y_2}\}$, hence the optimization problem of Eq. 4.17 reduces to:

$$F = \max_{\substack{(x_1, y_1, x_2, y_2) \in \\ \{l_{x_1}, u_{x_1}\} \times \{l_{y_1}, u_{y_1}\} \times \{l_{x_2}, u_{x_2}\} \times \{l_{y_2}, u_{y_2}\}}} |(x_1 \cdot y_2) + (x_2 \cdot y_1) - (Ax_1 + By_1 + Cx_2 + Dy_2 + E)| \tag{4.18}$$

However, as with the chain rule, we will also solve for the lower and upper bounds, l and u, for $(x_1 \cdot y_2) + (x_2 \cdot y_1)$ exactly. Furthermore, we can use the same proof technique to obtain these bounds and thus merely enumerate over the 2^4 corners to evaluate the expression, hence:

$$l = \min_{(x_1, y_1, x_2, y_2) \in \{l_{x_1}, u_{x_1}\} \times \{l_{y_1}, u_{y_1}\} \times \{l_{x_2}, u_{x_2}\} \times \{l_{y_2}, u_{y_2}\}} (x_1 \cdot y_2) + (x_2 \cdot y_1) \tag{4.19}$$

$$u = \max_{(x_1, y_1, x_2, y_2) \in \{l_{x_1}, u_{x_1}\} \times \{l_{y_1}, u_{y_1}\} \times \{l_{x_2}, u_{x_2}\} \times \{l_{y_2}, u_{y_2}\}} (x_1 \cdot y_2) + (x_2 \cdot y_1) \tag{4.20}$$

Alternatively, one can use the ideas of multilinear polynomial optimization over compact intervals [129] to prove the soundness of these bounds.

4.4.3 Quotient Rule Synthesized Transformer

Synthesizing an abstract transformer for Quotient Rule is challenging due to the divisions and multiplications involved. In fact, many works [130, 131] decompose the abstraction of the division x/y into first abstracting the univariate function $1/y$, then abstracting the multiplication of that intermediate result with x. However, our goal is to *avoid* decomposing these expressions into basic primitives, as naively composing primitive abstract transformers leads to imprecision. For functions $f, g : \mathbb{R} \to \mathbb{R}$, the quotient rule is defined as:

$$\left(\frac{f(u)}{g(u)}\right)' = \frac{f'(u) \cdot g(u) - f(u) \cdot g'(u)}{g(u)^2} \tag{4.21}$$

Forward-Mode Quotient Rule. In forward-mode AD the quotient rule is implemented as shown in Fig. 4.10:

```
1 z.real = x.real/y.real;
2 z.dual = ((x.dual*y.real)-(x.real*y.dual))/(y.real*y.real) //quotient rule
```

Figure 4.10: Forward Mode Quotient Rule Pattern

where intuitively $z.dual = (\frac{f(u)}{g(u)})'$, $x.dual = f'(u)$, and $y.dual = g'(u)$.

Reverse-Mode Quotient Rule. Likewise in reverse-mode AD the quotient rule is encoded as:

```
1 z = x/y; ...
2 z̄ = ...
3 x̄ += (1/y)*z̄; //derivative of quotient wrt x
4 ȳ += (-x/(y*y))*z̄; //derivative of quotient wrt y
```

Figure 4.11: Reverse Mode Quotient Rule Pattern

If one were to take the computational pattern of $z.dual$, but set $x.dual = 0$, the structure of the computation would be similar to the computation of $\overline{y}$. The abstraction for $\overline{x}$ can be handled separately using the chain rule abstraction $T^{\sharp}_{C_f}$, as the chain rule abstraction for $\log'(y) \cdot z$ is $\frac{z}{y}$.

Quotient Rule Abstraction Pattern. Based on the quotient rule implementations shown above, the computational pattern for which we want to synthesize an abstract transformer is given in Eq. 4.22 as:

$$g(x_1, y_1, x_2, y_2) = \frac{(x_2 \cdot y_1) - (x_1 \cdot y_2)}{x_2^2} \tag{4.22}$$

This pattern is a desirable choice for having a single abstraction, $T^{\sharp}_{Q}$ because there are 3 nonlinear multiplications and a nonlinear division, hence the pattern would otherwise be abstracted with the composition $T^{\sharp}_{*}(T^{\sharp}_{/}(T^{\sharp}_{-}(T^{\sharp}_{*}(T^{\sharp}_{*}(a))))))$. Furthermore there is a high degree of correlation between the numerator and denominator since they both contain x_2, however, due to the imprecision most abstract interpreters face in the presence of multiple nonlinearities, it is hard to precisely capture this dependency. Further, as mentioned, the computation of $\overline{y}$ in the reverse mode is a special instance of this abstraction pattern when the second argument $y_1 = 0$.

Abstraction. We now present the synthesis of the abstract transformer for the Quotient Rule, $T^{\sharp}_{Q}$, which Algorithm 4.3 presents. As before, Pasado must solve an optimization problem to obtain sound bounds around the synthesized linear approximation (obtained via regression). To ensure that there are not additional interior critical points to consider, we require some constraints on the synthesized coefficients A, B, C, D, E given in terms of the input bounds $\kappa_{x_i} \in \{l_{x_i}, u_{x_i}\}$, $\kappa_{y_i} \in \{l_{y_i}, u_{y_i}\}$. Further it is simple to enforce these constraints - if the synthesized coefficients do not satisfy the following constraints, we can perturb the

coefficients by some small δ, which will not affect the abstraction's precision, but will ensure soundness. The conditions we require are:

$$\kappa_{x_1} \neq \frac{-D}{A^2} \wedge \sqrt[2]{\sqrt[3]{\frac{-\kappa_{y_1}}{AD}}} \neq \frac{-3\kappa_{y_1}}{2C} \wedge \frac{1}{B} \neq \kappa_{x_2} \wedge \frac{A}{B^2} \neq \kappa_{y_2} \tag{4.23}$$

The *Adjust* function checks that the coefficients obtained by the linear regression satisfy these constraints, and if not, it will add small perturbations ($\delta < 10^{-9}$) until the conditions of Eq. 4.23 are satisfied.

Algorithm 4.3: Quotient Rule Abstract Transformer $T_Q^\sharp$

Input: Abstract state a where $\widehat{x_1} = a[x_1].\widehat{x_1}$, $\widehat{y_1} = a[y_1].\widehat{y_1}$, $\widehat{x_2} = a[x_2].\widehat{x_2}$, and $\widehat{y_2} = a[y_2].\widehat{y_2}$

$l_{x_1}, u_{x_1} \leftarrow Range(a[x_1])$ and $l_{y_1}, u_{y_1} \leftarrow Range(a[y_1])$

$l_{x_2}, u_{x_2} \leftarrow Range(a[x_2])$ and $l_{y_2}, u_{y_2} \leftarrow Range(a[y_2])$

$grid \leftarrow GridSample([l_{x_1}, u_{x_1}] \times [l_{y_1}, u_{y_1}] \times [l_{x_2}, u_{x_2}] \times [l_{y_2}, u_{y_2}])$

$pts \leftarrow \left\{ \frac{(x_2 \cdot y_1) - (x_1 \cdot y_2)}{x_2^2} : (x_1, y_1, x_2, y_2) \in grid \right\}$

$A, B, C, D, E \leftarrow LinearRegression(grid, pts)$

$Adjust(A, B, C, D, E)$

$$F \leftarrow \max_{\substack{x_1 \in [l_{x_1}, u_{x_1}], y_1 \in [l_{y_1}, u_{y_1}] \\ x_2 \in [l_{x_2}, u_{x_2}], y_2 \in [l_{y_2}, u_{y_2}]}} \left| \frac{(x_2 \cdot y_1) - (x_1 \cdot y_2)}{x_2^2} - (Ax_1 + By_1 + Cx_2 + Dy_2 + E) \right|$$

$$l, u \leftarrow \min_{\substack{x_1 \in [l_{x_1}, u_{x_1}], y_1 \in [l_{y_1}, u_{y_1}] \\ x_2 \in [l_{x_2}, u_{x_2}], y_2 \in [l_{y_2}, u_{y_2}]}} \frac{(x_2 \cdot y_1) - (x_1 \cdot y_2)}{x_2^2}, \max_{\substack{x_1 \in [l_{x_1}, u_{x_1}], y_1 \in [l_{y_1}, u_{y_1}] \\ x_2 \in [l_{x_2}, u_{x_2}], y_2 \in [l_{y_2}, u_{y_2}]}} \frac{(x_2 \cdot y_1) - (x_1 \cdot y_2)}{x_2^2}$$

return $A\widehat{x_1} + B\widehat{y_1} + C\widehat{x_2} + D\widehat{y_2} + E + F\epsilon_{new},\ [l, u]$

Optimization Problem. To obtain a sound enclosure around the linear approximation for the Quotient rule, we must solve the following nonlinear, nonconvex 4D optimization problem:

$$\max_{\substack{x_1 \in [l_{x_1}, u_{x_1}], y_1 \in [l_{y_1}, u_{y_1}] \\ x_2 \in [l_{x_2}, u_{x_2}], y_2 \in [l_{y_2}, u_{y_2}]}} \left| \frac{(x_2 \cdot y_1) - (x_1 \cdot y_2)}{x_2^2} - (Ax_1 + By_1 + Cx_2 + Dy_2 + E) \right| \tag{4.24}$$

This optimization problem is more difficult than the one needed for the product rule, however, much of the core idea is the same and *it is still tractable to solve automatically.* We will still need to check all 2^4 corner points $\{l_{x_1}, u_{x_1}\} \times \{l_{y_1}, u_{y_1}\} \times \{l_{x_2}, u_{x_2}\} \times \{l_{y_2}, u_{y_2}\}$, however, we will also need to check for potential critical points where the gradient could be zero and that cannot be immediately ruled out by the Hessian test. Thankfully, solving for these critical points involves only *univariate* cubic polynomial equations, as we will be able

to provably rule out any interior critical point where x_1, y_1, and y_2 are *not* fixed to their respective lower or upper bounds. Thus the only potential critical points are along the edges of the 4D hypercube where x_1, y_1, and y_2 are fixed. To find these potential critical points, we only have to solve a cubic polynomial in x_2. Thus this set will still be finite as each cubic polynomial has at most 3 real roots. We now define this set of potential critical points, S as solutions to 8 possible cubic equations.

$$S = \{(x_1, y_1, x_2, y_2) : x_1 \in \{l_{x_1}, u_{x_1}\}, y_1 \in \{l_{y_1}, u_{y_1}\}, y_2 \in \{l_{y_2}, u_{y_2}\}, Cx_2^3 + y_1 x_2 - 2x_1 y_2 = 0, x_2 \in [l_{x_2}, u_{x_2}]\} \tag{4.25}$$

Thus we can solve the maximization problem of Eq. 4.24 by enumerating over the corner points and all points in S, reducing the problem to:

$$\max_{\substack{(x_1,y_1,x_2,y_2) \in \{l_{x_1},u_{x_1}\} \times \{l_{y_1},u_{y_1}\} \times \{l_{x_2},u_{x_2}\} \times \{l_{y_2},u_{y_2}\} \\ \cup S}} \left| \frac{(x_2 \cdot y_1) - (x_1 \cdot y_2)}{x_2^2} - (Ax_1 + By_1 + Cx_2 + Dy_2 + E) \right| \tag{4.26}$$

This strategy of solving for roots of a cubic equation to check as possible extrema can also be used to solve for the optimal interval lower and upper bounds. Hence we compute the following:

$$l = \min_{\substack{x_1 \in [l_{x_1},u_{x_1}], y_1 \in [l_{y_1},u_{y_1}] \\ x_2 \in [l_{x_2},u_{x_2}], y_2 \in [l_{y_2},u_{y_2}]}} \frac{(x_2 \cdot y_1) - (x_1 \cdot y_2)}{x_2^2} = \min_{\substack{\{l_{x_1},u_{x_1}\} \times \{l_{y_1},u_{y_1}\} \times \{l_{x_2},u_{x_2}\} \times \{l_{y_2},u_{y_2}\} \\ \cup S}} \frac{(x_2 \cdot y_1) - (x_1 \cdot y_2)}{x_2^2} \tag{4.27}$$

$$u = \max_{\substack{x_1 \in [l_{x_1},u_{x_1}], y_1 \in [l_{y_1},u_{y_1}] \\ x_2 \in [l_{x_2},u_{x_2}], y_2 \in [l_{y_2},u_{y_2}]}} \frac{(x_2 \cdot y_1) - (x_1 \cdot y_2)}{x_2^2} = \max_{\substack{\{l_{x_1},u_{x_1}\} \times \{l_{y_1},u_{y_1}\} \times \{l_{x_2},u_{x_2}\} \times \{l_{y_2},u_{y_2}\} \\ \cup S}} \frac{(x_2 \cdot y_1) - (x_1 \cdot y_2)}{x_2^2} \tag{4.28}$$

where the only difference is that the set S in Equations 4.27 and 4.28 is now given by $S = \{(x_1, y_1, x_2, y_2) \ : \ x_1 \in \{l_{x_1}, u_{x_1}\}, y_1 \in \{l_{y_1}, u_{y_1}\}, y_2 \in \{l_{y_2}, u_{y_2}\}, x_2 = \frac{2x_1 y_2}{y_1}, x_2 \in [l_{x_2}, u_{x_2}]\}$.

4.4.4 Soundness

In this section we prove the soundness of Pasado's synthesized abstract transformers. At a high level, our proof relies on the fact that at interior critical points the Hessian will be indeterminant (ensuring they are saddle points). Intuitively, this insight allows for ruling out the (multi-dimensional) interior, and thus checking only (lower dimensional) boundaries to solve the optimization problem. We first prove the following useful lemma:

Lemma 4.1. Let $f(x) : \mathbb{R}^m \to \mathbb{R}$ be twice differentiable $x \in \mathbb{R}^m$. If $Hessian(f, x)$ is indeterminant then so is $Hessian(-f, x)$

Proof. If $Hessian(f, x)$ is indeterminant then it has both positive and negative eigenvalues. The eigenvalues of $Hessian(-f, x)$ will just be -1 times each eigenvalue of $Hessian(f, x)$, hence there will still be both positive and negative eigenvalues meaning $Hessian(-f, x)$ is indeterminant. QED.

Theorem 4.1. The Chain Rule Transformers $T^\sharp_{C_f}$, synthesized by Pasado for any $f : \mathbb{R} \to \mathbb{R}$ obeying the properties of Sec. 4.3.2 are sound. Equivalently for $a \in \mathbb{A}$, $\{f'(x) \cdot y : (x, y) \in \gamma(a)\} \subseteq \gamma(T^\sharp_{C_f}(a))$

Proof. (**Soundness of Linear Bounds**) Let $f(x) : \mathbb{R} \to \mathbb{R}$ and $A, B, C \in \mathbb{R}$ be the coefficients inferred by the linear regression. To ensure soundness of $T^\sharp_{C_f}$, our goal reduces to solving the following optimization problem:

$$\max_{x \in [l_x, u_x], y \in [l_y, u_y]} |f'(x) \cdot y - (Ax + By + C)|$$

Instead of looking for interior critical points with the first derivative test, we first examine the Hessian, H, of $f'(x) \cdot y - (Ax + By + C)$ which is given as:

$$H = \begin{bmatrix} f'''(x) \cdot y & f''(x) \\ f''(x) & 0 \end{bmatrix}$$

The Hessian determinant is $-(f''(x)^2)$ which is always ≤ 0. For any x such that $f''(x) \neq 0$, then $-(f''(x)^2) < 0$, ruling out any interior critical point since the Hessian determinant is negative. Further, any x such that $f''(x) = 0$ cannot be a critical point, since from the first derivative test $f''(x) - A = 0$ is necessary for a critical point, however, if $f''(x) = 0$ then $f''(x) - A = -A \neq 0$, since we required $A \neq 0$. Thus the optimal value will occur along the 4 boundary lines of the square. The first two boundary lines ($x = l_x$ or $x = u_x$) are simple, since the optimal value of a linear function will occur at the end points, thus it suffices to check the 4 corners $\{l_x, u_x\} \times \{l_y, u_y\}$. For the boundary lines where we fix $y = l_y$ or $y = u_y$, we now only have to solve univariate optimization problems. Applying the first derivative test to $f'(x)l_y$ and $f'(x)u_y$ we must solve for all $x^* \in [l_x, u_x]$ such that $f''(x^*)l_y - A = 0$ as well as all $x^{**} \in [l_x, u_x]$ such that $f''(x^{**})u_y - A = 0$. However, these equations can be solved as we already required that one has a verified root solver for f''. Hence we just check (x^*, l_y) and (x^{**}, u_y) for all x^*, $x^{**} \in [l_x, u_x]$ returned by the root solver. Furthermore, the Hessian determinant of $-(f'(x) \cdot y - (Ax + By + C))$ will be identical, and any critical point

of $f'(x) \cdot y - (Ax + By + C)$ will be a critical point of $-(f'(x) \cdot y - (Ax + By + C))$, hence we need not worry about the absolute value in the maximization problem.

(**Soundness of Interval Bounds**). The correctness of the interval bounds also follows nearly identically as the proof for zonotopes, albeit there is a single edge case whenever $\exists x^* \in [l_x, u_x] : f''(x^*) = 0$ and $f'(x^*) = 0$ as the Hessian test would be inconclusive (since its determinant would be 0), but unlike with zonotopes, we cannot ensure that the gradient at x^* is non-zero (as we did by enforcing $A \neq 0$) since there could be x^* such that both $f'(x^*) = f''(x^*) = 0$. If the function f is such that there is never any shared root of both f' and f'', the proof is complete as this will never happen (this is the case for $\exp, \log, \sigma, tanh$) but for functions like x^4 or x^3 it is possibility. However any such x^* will be a root of $f''(x) \cdot y$ for any value of y, hence we can call the same verified root solver with $A = 0$, $y = l_y$ to obtain the x^* . Further, $f'(x^*) = f''(x^*) = 0$ implies $f'(x^*) \cdot y = 0$ for all y, hence checking at (x^*, l_y) is sufficient, and this point is already included in the points we evaluate.

QED.

Theorem 4.2. The Product Rule Transformer, $T_P^\sharp$, synthesized by Pasado is sound. Equivalently for $a \in \mathbb{A}$, $\{x_1 \cdot y_2 + x_2 \cdot y_1 : (x_1, y_1, x_2, y_2) \in \gamma(a)\} \subseteq \gamma(T_P^\sharp(a))$

Proof. (**Soundness of Linear Bounds**) The Hessian for $(x_1 \cdot y_2) + (x_2 \cdot y_1) - (Ax_1 + By_1 + Cx_2 + Dy_2 + E)$ at any point is the matrix $\begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$ which has both positive and negative eigenvalues, meaning the Hessian is everywhere indeterminant. Hence any potential critical point is necessarily a saddle point, thus the extrema must occur along the boundaries. We repeat this procedure for each of the $\binom{4}{3}$ 3D boundary optimization problems where in each case 1 of the 4 dimensions is fixed to a lower or upper boundary. The (unique) 3D subproblem Hessians are $\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$, $\begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}$, and $\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ all of which have both positive and negative eigenvalues meaning any interior point along the 3D boundary of the 4D cube is necessarily a saddle point. Repeating this same procedure for the 2D subproblems we again find that all 2D Hessians are either indeterminant which implies any interior critical point is a saddle (thus the extrema occurs on the corners) or the 2D subproblem is such that the function is linear in which case the extrema will also occur only on the corners. Furthermore, by lemma 4.1 these same properties hold for $-((x_1 \cdot y_2) + (x_2 \cdot y_1) - (Ax_1 + By_1 + Cx_2 + Dy_2 + E))$ thus handling the absolute value in the maximization problem.

(**Soundness of Interval Bounds**) Unlike in the case of the Chain rule where some of the cases depended on ensuring the gradients were nonzero, which had to be handled differently for zonotopes vs. intervals, the entire technique for product rule relies only on the Hessian information which will be the same for intervals in order to compute $\min(x_1 \cdot y_2) + (x_2 \cdot y_1)$ and $\max(x_1 \cdot y_2) + (x_2 \cdot y_1)$ since $Hessian((x_1 \cdot y_2) + (x_2 \cdot y_1) - (Ax_1 + By_1 + Cx_2 + Dy_2 + E))$ is the same as $Hessian((x_1 \cdot y_2) + (x_2 \cdot y_1))$. Thus for computing the bounds on the zonotope error symbol and computing the precise interval lower and upper bounds, it suffices to simply check the 2^4 corners.

QED.

Theorem 4.3. The Quotient Rule Transformer, $T_Q^\sharp$ synthesized by Pasado is sound. Equivalently for $a \in \mathbb{A}$, $\{\frac{x_2 \cdot y_1 - x_1 \cdot y_2}{x_2^2} : (x_1, y_1, x_2, y_2) \in \gamma(a)\} \subseteq \gamma(T_Q^\sharp(a))$

Proof. (**Linear Bounds Soundness**) The 4D Hessian is $\begin{bmatrix} 0 & 0 & \frac{2d}{x_2^3} & \frac{-1}{x_2^2} \\ 0 & 0 & \frac{-1}{x_2^2} & 0 \\ \frac{2y_2}{c^3} & \frac{-1}{x_2^2} & \frac{6(y_1x_2 - x_1y_2)}{x_2^4} - \frac{4y_1}{x_2^3} & \frac{2x_1}{x_2^3} \\ \frac{-1}{x_2^2} & 0 & \frac{2x_1}{x_2^3} & 0 \end{bmatrix}$ and its determinant is $\frac{1}{x_2^8}$, meaning that the the Hessian has no zero eigenvalues since $x_2 \neq 0$. Furthermore, by Sylvester's criteria the Hessian is neither positive definite nor negative definite, hence it is indeterminant which implies any interior critical point is necessarily a saddle point, thus the extrema will occur along a boundary of the 4D cube.

We will repeat this idea for the lower dimensional subproblems and show that when restricted to the 3 dimensional boundaries, they also do not have any interior extrema, thus the optimal value must occur on *their* boundaries (the boundary of the boundaries of the 4D-hypercube). The cases for optimizing along these 3D boundaries, and their respective 2D boundaries, and their respective 1D boundaries are all shown below.

3D Subproblems

Proof. Case 1) Fixed y_2 to either l_{y_2} or u_{y_2} - we denote the fixed constant value of y_2 as κ_{y_2}, hence $\kappa_{y_2} \in \{l_{y_2}, u_{y_2}\}$. In this case the first derivatives are:

$$\frac{\partial}{\partial x_1} = \frac{\kappa_{y_2}}{x_2^2} - A$$

$$\frac{\partial}{\partial y_1} = \frac{1}{x_2} - B$$

$$\frac{\partial}{\partial x_2} = \frac{2x_1\kappa_{y_2} - y_1x_2}{x_2^3} - C$$

If $\kappa_{y_2} = 0$, then $\frac{\partial}{\partial x_1} \neq 0$ since by requirement $A \neq 0$, and $x_2 \neq 0$, thus there is no critical point since the gradient $(\frac{\partial}{\partial x_1}, \frac{\partial}{\partial y_1}, \frac{\partial}{\partial x_2})$ cannot be the all-zeros vector.

If $\kappa_{y_2} \neq 0$ then because $x_2 \neq 0$, $A, B \neq 0$, we can ensure that $\frac{\partial}{\partial x_1} \neq 0$ by requiring the condition on A, B that $\frac{A}{B^2} \neq \kappa_{y_2}$. Since for any $x_2 \in [l_{x_2}, u_{x_2}]$ this guarantees that $\frac{\kappa_{y_2}}{x_2^2} - A$ and $\frac{1}{x_2} - B$ cannot both be zero. Thus why we require both $\frac{A}{B^2} \neq l_{y_2}$ and $\frac{A}{B^2} \neq u_{y_2}$.

Case 2) Fixed x_2 to either l_{x_2} or u_{x_2} - we denote the fixed constant value of x_2 as κ_{x_2}. In this case the first derivatives are:

$$\frac{\partial}{\partial x_1} = \frac{y_2}{\kappa_{x_2}^2} - A$$

$$\frac{\partial}{\partial y_1} = \frac{1}{\kappa_{x_2}} - B$$

$$\frac{\partial}{\partial y_2} = \frac{x_1}{\kappa_{x_2}^2} - D$$

It sufficis to require that $\frac{1}{B} \neq \kappa_{x_2}$, as this guarantees that $\frac{\partial}{\partial y_1} \neq 0$, thus the gradient cannot be the all-zeros vector.

Case 3) Fixed y_1 to either l_{y_1} or u_{y_1} - we denote the fixed constant value of y_1 as κ_{y_1}, hence $\kappa_{y_1} \in \{l_{y_1}, u_{y_1}\}$. In this case the first derivatives are:

$$\frac{\partial}{\partial x_1} = \frac{-y_2}{x_2^2} - A$$

$$\frac{\partial}{\partial x_2} = \frac{2x_1y_2 - \kappa_{y_1}x_2}{x_2^3} - C$$

$$\frac{\partial}{\partial y_2} = \frac{-x_1}{{x_2}^2} - D$$

.

For this case we will show that any hypothetical root of the above system of equations (which is what is needed for a critical point) is necessarily a saddle point.

Case 3.1) $\kappa_{y_1} \neq 0$. If $\frac{\partial}{\partial x_1} = \frac{\partial}{\partial x_2} = \frac{\partial}{\partial y_2} = 0$ then any critical point will be a root of the above system of equations, furthermore such a root (x_1^*, x_2^*, y_2^*) would be of the form: $(-Dx_2^{*2}, x_2^*, -Ax_2^{*2})$, as that would be needed to ensure that $\frac{\partial}{\partial x_1} = \frac{\partial}{y_2} = 0$. Plugging such a hypothetical root into $\frac{\partial}{\partial x_2} = 0$ implies that x_2^* must *also* be a root of

$$\frac{2ADx_2^{*4} - \kappa_{y_1}x_2^*}{x_2^{*3}} - C = 0$$

But since x_2^* must be nonzero, then equivalently it must be a root of:

$$x_2^{*3} - \frac{C}{2AD}x_2^{*2} - \frac{\kappa_{y_1}}{2AD} = 0$$

Furthermore the Hessian determinant is $-\frac{2(x_1y_2+\kappa_{y_1}x_2)}{x_2^8}$, however the Hessian has 0 in its upper left corner, meaning the Hessian is neither positive definite nor negative definite (Sylvester's criteria). Thus if the Hessian determinant is non-zero, then the Hessian is necessarily indeterminant meaning the hypothetical root would be a saddle point. Thus we simply need to show that at such a hypothetical root $(-Dx_2^{*2}, x_2^*, -Ax_2^{*2})$, the Hessian determinant is non-zero. The Hessian determinant is non-zero provided that the numerator $(x_1y_2 + \kappa_{y_1}x_2) \neq 0$, However computing this numerator at our hypothetical root gives:

$$(AD(x_2^*)^4 + \kappa_{y_1}x_2^*)$$

Which is zero if $x_2^* = 0$ (which is not possible) or:

$$x_2^* = \sqrt[3]{\frac{-\kappa_{y_1}}{AD}}$$

Thus we just need to ensure that $\sqrt[3]{\frac{-\kappa_{y_1}}{AD}}$ is not also a root of $x_2^{*3} - \frac{C}{2AD}x_2^{*2} - \frac{\kappa_{y_1}}{2AD} = 0$. We can ensure this by requiring that

$$\sqrt[\frac{2}{3}]{\frac{-\kappa_{y_1}}{AD}} \neq \frac{-3\kappa_{y_1}}{2C}$$

Case 3.2) $\kappa_{y_1} = 0$. In this case it is still true that any root would be of the form $(-Dx_2^{*2}, x_2^*, -Ax_2^{*2})$, further because $\kappa_{y_1} = 0$, x_2^* must necessarily be

$$x_2^* = \frac{C}{2AD}$$

However, as before the Hessian has a 0 in the upper left corner meaning it is not positive definite or negative definite, furthermore the determinant is $-\frac{2x_1y_2}{x_2^8}$, which at the point $(-D(\frac{C}{2AD})^2, \frac{C}{2AD}, -A(\frac{C}{2AD})^2)$ is non-zero meaning the Hessian is indeterminant

It is worth noting that the Reverse mode version of the quotient rule is an instance of this case.

Case 4) Fixed x_1 to either l_{x_1} or u_{x_1} - we denote the fixed constant value of x_1 as κ_{x_1}. In this case the first derivatives are:

$$\frac{\partial}{\partial y_1} = \frac{1}{x_2} - B$$
$$\frac{\partial}{\partial x_2} = \frac{2\kappa_{x_1} y_2 - y_1 x_2}{x_2^3} - C$$
$$\frac{\partial}{\partial y_2} = \frac{\kappa_{x_1}}{{x_2}^2} - D$$

.

If $\kappa_{x_1} = 0$, then $\frac{\partial}{\partial y_2} = -D$ and $D \neq 0$, hence it is impossible for there to be an extrema. If $\kappa_{x_1} \neq 0$, then it suffices to require that $\kappa_{x_1} \neq \frac{-D}{A^2}$. This is because $\frac{1}{x_2} - B = 0 = \frac{\kappa_{x_1}}{x_2^2} - D$ implies that $B^2 = \frac{-D}{\kappa_{x_1}}$. Hence by the contrapositive $\frac{1}{x_2} - B = 0 = \frac{\kappa_{x_1}}{x_2^2} - D$ cannot have been true. QED.

2D Subproblems

Here we show that when solving for the optimal values along the boundary of the boundary, we can still ensure that there are no interior critical points, and thus all local extrema must occur on the boundary of the boundary of the boundary.

Proof. Case 1) Fixed x_2, y_2 to κ_{x_2} and κ_{y_2} respectively. In this case the function $\frac{(x_2 \cdot y_1) - (x_1 \cdot y_2)}{x_2^2} - (Ax_1 + By_1 + Cx_2 + Dy_2 + E)$ becomes linear in both dimensions, and the optimal values will occur at the corner points and hence there are no interior critical points.

Case 2) Fixed y_1, y_2 to κ_{y_1} and κ_{y_2} respectively. The 2D Hessian determinant in this 2D subproblem is $\frac{-4\kappa_{y_2}^2}{x_2^6}$ which is negative provided $\kappa_{y_2} \neq 0$. Hence if $\kappa_{y_2} \neq 0$ any interior point is a saddle. If $\kappa_{y_2} = 0$ then $\frac{\partial}{\partial x_1} = -A \neq 0$, hence there are no interior critical points to begin with.

Case 3) Fixed y_1, x_2 to κ_{y_1} and κ_{x_2} respectively. In this case the 2D Hessian determinant is $\frac{-1}{x_2^4}$ which is always strictly negative, hence any potential critical point would necessarily be a saddle point and thus not a local extrema.

Case 4) Fixed x_1, y_2 to κ_{x_1} and κ_{y_2} respectively. In this case the 2D Hessian determinant is also $\frac{-1}{x_2^4}$ which is always strictly negative, hence any potential critical point would necessarily be a saddle point and thus not a local extrema.

Case 5) Fixed x_1, x_2 to κ_{x_1} and κ_{x_2} respectively. In this case the function $\frac{(x_2 \cdot y_1) - (x_1 \cdot y_2)}{x_2^2} - (Ax_1 + By_1 + Cx_2 + Dy_2 + E)$ becomes linear in both dimensions, and the optimal values will occur at the corner points and hence there are no interior critical points. Case 6) Fixed x_1, y_1 to κ_{x_1} and κ_{y_1} respectively. In this case the Hessian determinant is $\frac{-4\kappa_{x_1}^2}{x_2^6}$ which is

negative provided $\kappa_{x_1} \neq 0$ hence *any* interior critical point is necessarily a saddle point. If $\kappa_{x_1} = 0$, then $\frac{\partial}{\partial y_2} = -D \neq 0$, thus there is no critical point to begin with. QED.

1D Subproblems

Proof. Case 1) Fix every variable to its lower or upper bounds *except* x_1. In this case the function becomes linear and thus the extrema will occur at either $x_1 = l_{x_1}$ or $x_1 = u_{x_1}$

Case 2) Fix every variable to its lower or upper bounds *except* y_1. In this case the function still is linear. and thus the extrema will occur at either $y_1 = l_{y_1}$ or $y_1 = u_{y_1}$

Case 3) Fixed every variable to its lower or upper bounds *except* x_2. In this case the function is *not* linear hence we have to solve for critical points, however thankfully this is now only a univariate problem. We have to solve for $x_2 \in [l_{x_2}, u_{x_2}]$ such that $\frac{\partial}{\partial x_2} = \frac{2\kappa_{x_1}\kappa_{y_2} - \kappa_{y_1} x_2}{x_2^3} - C = 0$. Hence we must solve the 3rd degree polynomial $Cx_2^3 + \kappa_{y_1} x_2 - 2\kappa_{x_1}\kappa_{y_2} = 0$. However because $\kappa_{x_1}, \kappa_{y_1}, \kappa_{y_2}$ each could be the respective lower *or* upper bounds, this means we must actually solve 8 versions of this 3rd degree (univariate) polynomial. However this can still easily be done analytically, and thus we would check if each of the 8 equations has a root in $[l_{x_2}, u_{x_2}]$

Case 4) Fix every variable to its lower or upper bounds *except* y_2. In this case the function still is linear and thus the extrema will occur at either $y_2 = l_{y_2}$ or $y_2 = u_{y_2}$ QED.

0D Subproblems

Proof. To find the optimal value along the 0D boundaries, we just enumerate over all 2^4 corners: $(x_1, y_1, x_2, y_2) \in \{l_{x_1}, u_{x_1}\} \times \{l_{y_1}, u_{y_1}\} \times \{l_{x_2}, u_{x_2}\} \times \{l_{y_2}, u_{y_2}\}$ QED.

(**Soundness of Interval Bounds**). We now detail the rest of the cases for the quotient rule soundness for the Interval domain. These cases follow almost identically to the zonotope version of the proof described above, but with a few notable differences that we will subsequently see.

3D Subproblems

Proof. Case 1) Fixed y_2 to either l_{y_2} or u_{y_2} - we denote the fixed constant value of y_2 as κ_{y_2}, hence $\kappa_{y_2} \in \{l_{y_2}, u_{y_2}\}$. In this case the first derivatives are:

$$\frac{\partial}{\partial x_1} = \frac{\kappa_{y_2}}{x_2^2}$$

$$\frac{\partial}{\partial y_1} = \frac{1}{x_2}$$

$$\frac{\partial}{\partial x_2} = \frac{2x_1\kappa_{y_2} - y_1x_2}{x_2^3}$$

Since the range $[l_{x_2}, u_{x_2}]$ excludes zero, this ensures that $\frac{\partial}{\partial y_1} = \frac{1}{x_2}$ is never zero

Case 2) Fixed x_2 to either l_{x_2} or u_{x_2} - we denote the fixed constant value of x_2 as κ_{x_2}. In this case the first derivatives are:

$$\frac{\partial}{\partial x_1} = \frac{y_2}{\kappa_{x_2}^2}$$

$$\frac{\partial}{\partial y_1} = \frac{1}{\kappa_{x_2}}$$

$$\frac{\partial}{\partial y_2} = \frac{x_1}{\kappa_{x_2}^2}$$

Since the range $[l_{x_2}, u_{x_2}]$ excludes zero, this ensures $\kappa_{x_2} \neq 0$, hence $\frac{\partial}{\partial y_1} = \frac{1}{\kappa_{x_2}}$ is never zero

Case 3) Fixed y_1 to either l_{y_1} or u_{y_1} - we denote the fixed constant value of y_1 as κ_{y_1}, hence $\kappa_{y_1} \in \{l_{y_1}, u_{y_1}\}$. In this case the first derivatives are:

$$\frac{\partial}{\partial x_1} = \frac{-y_2}{x_2^2}$$

$$\frac{\partial}{\partial x_2} = \frac{2x_1y_2 - \kappa_{y_1}x_2}{x_2^3}$$

$$\frac{\partial}{\partial y_2} = \frac{-x_1}{{x_2}^2}$$

.

Case 3.1) $\kappa_{y_1} \neq 0$. Since $x_2 \neq 0$, the only way for $\frac{\partial}{\partial x_1} = \frac{\partial}{\partial y_2} = \frac{\partial}{\partial x_2} = 0$ is if $x_1 = y_2 = 0$ and $\kappa_{y_1} = 0$, hence if $\kappa \neq 0$, then it is not possible for $\frac{\partial}{\partial x_2}$ to be zero.

Case 3.2 $\kappa_{y_1} = 0$. In this case if $[l_{x_1}, u_{x_1}]$ and $[l_{y_2}, u_{y_2}]$ both include 0, then we could have a critical point that the Hessian test cannot immediately rule out. However the value of the function at this critical point is always 0, hence it suffices to add a single additional point (0) to the finite list of points to check

Case 4) Fixed x_1 to either l_{x_1} or u_{x_1} - we denote the fixed constant value of x_1 as κ_{x_1}. In this case the first derivatives are:

$$\frac{\partial}{\partial y_1} = \frac{1}{x_2}$$

$$\frac{\partial}{\partial x_2} = \frac{2\kappa_{x_1} y_2 - y_1 x_2}{x_2^3}$$

$$\frac{\partial}{\partial y_2} = \frac{\kappa_{x_1}}{{x_2}^2}$$

Since the range $[l_{x_2}, u_{x_2}]$ excludes zero, this ensures that $\frac{\partial}{\partial y_1} = \frac{1}{x_2}$ is never zero. QED.

2D Subproblems

Proof. Case 1) Fixed x_2, y_2 to κ_{x_2} and κ_{y_2} respectively. In this case the function $\frac{(x_2 \cdot y_1) - (x_1 \cdot y_2)}{x_2^2}$ becomes linear in both dimensions, and the optimal values (both min *and* max) will occur at the corner points and hence there are no interior critical points.

Case 2) Fixed y_1, y_2 to κ_{y_1} and κ_{y_2} respectively. The 2D Hessian determinant in this 2D subproblem is $\frac{-4\kappa_{y_2}^2}{x_2^6}$ which is negative provided $\kappa_{y_2} \neq 0$. Hence if $\kappa_{y_2} \neq 0$ any interior point is a saddle. If $\kappa_{y_2} = 0$ then the function $\frac{(x_2 \cdot y_1) - (x_1 \cdot y_2)}{x_2^2}$ becomes $\frac{\kappa_{y_1}}{x_2}$, hence $\frac{\partial}{\partial x_2} = \frac{-\kappa_{y_1}}{x_2^2}$ which is non-zero provided $\kappa_{y_1} \neq 0$. If κ_{y_1} and $\kappa_{y_2} = 0$, then the function is everywhere 0, which will caught when we check corner points.

Case 3) Fixed y_1, x_2 to κ_{y_1} and κ_{x_2} respectively. In this case the 2D Hessian determinant is $\frac{-1}{x_2^4}$ which is always strictly negative, hence any potential critical point would necessarily be a saddle point and thus not a local extrema.

Case 4) Fixed x_1, y_2 to κ_{x_1} and κ_{y_2} respectively. In this case the 2D Hessian determinant is also $\frac{-1}{x_2^4}$ which is always strictly negative, hence any potential critical point would necessarily be a saddle point and thus not a local extrema.

Case 5) Fixed x_1, x_2 to κ_{x_1} and κ_{x_2} respectively. In this case the function $\frac{(x_2 \cdot y_1) - (x_1 \cdot y_2)}{x_2^2}$ becomes linear in both dimensions, and the optimal values will occur at the corner points and hence there are no interior critical points.

Case 6) Fixed x_1, y_1 to κ_{x_1} and κ_{y_1} respectively. In this case the Hessian determinant is $\frac{-4\kappa_{x_1}^2}{x_2^6}$ which is negative provided $\kappa_{x_1} \neq 0$ hence *any* interior critical point is necessarily a saddle point. If $\kappa_{x_1} = 0$, then the function $\frac{(x_2 \cdot y_1) - (x_1 \cdot y_2)}{x_2^2}$ becomes $\frac{\kappa_{y_1}}{x_2}$, hence $\frac{\partial}{\partial x_2} = \frac{-\kappa_{y_1}}{x_2^2}$ which is non-zero provided $\kappa_{y_1} \neq 0$. If κ_{y_1} and $\kappa_{x_1} = 0$, then the function is everywhere 0, which will caught when we check corner points. QED.

1D Subproblems

Proof. Case 1) Fix every variable to its lower or upper bounds *except* x_1. In this case the function becomes linear and thus the extrema will occur at either $x_1 = l_{x_1}$ or $x_1 = u_{x_1}$

Case 2) Fix every variable to its lower or upper bounds *except* y_1. In this case the function still is linear. and thus the extrema will occur at either $y_1 = l_{y_1}$ or $y_1 = u_{y_1}$

Case 3) Fixed every variable to its lower or upper bounds *except* x_2. In this case the function is *not* linear hence we have to solve for critical points, however thankfully this is now only a univariate problem. We have to solve for $x_2 \in [l_{x_2}, u_{x_2}]$ such that $\frac{\partial}{\partial x_2} = \frac{2\kappa_{x_1}\kappa_{y_2} - \kappa_{y_1} x_2}{x_2^3} = 0$. Hence we must solve $2\kappa_{x_1}\kappa_{y_2} - \kappa_{y_1} x_2 = 0$. Hence for each possible root $x_2 = \frac{2\kappa_{x_1}\kappa_{y_2}}{\kappa_{y_1}}$, we must check if $\frac{2\kappa_{x_1}\kappa_{y_2}}{\kappa_{y_1}} \in [l_{x_2}, u_{x_2}]$, and if so we will need to evaluate the function $\frac{(x_2 \cdot y_1) - (x_1 \cdot y_2)}{x_2^2}$ at said critical point. We keep in mind that there are really 8 versions of this equation that we must resolve.

Case 4) Fix every variable to its lower or upper bounds *except* y_2. In this case the function still is linear and thus the extrema will occur at either $y_2 = l_{y_2}$ or $y_2 = u_{y_2}$ QED.

0D Subproblems

We just enumerate over all 2^4 corners: $(x_1, y_1, x_2, y_2) \in \{l_{x_1}, u_{x_1}\} \times \{l_{y_1}, u_{y_1}\} \times \{l_{x_2}, u_{x_2}\} \times \{l_{y_2}, u_{y_2}\}$

QED.

Given the similarity in all three proofs, the same ideas could theoretically be used to support other nonlinear expressions, provided their respective Hessians also rule out interior critical points.

4.4.5 Precision

Having now defined how to solve the optimization problems needed for Pasado's abstract transformers and their soundness, we can now state the following theorem about their precision:

Theorem 4.4. The lower and upper interval bounds computed by Pasado's synthesized abstract transformers in Equations 4.13, 4.14, 4.19, 4.20, 4.27, and 4.28 are optimal for the interval domain.

Proof. (sketch) Since Pasado solves these optimization problems exactly, instead of upper bounding the maximum or lower bounding the minimum, the bounds cannot be any tighter. QED.

Hence for these AD patterns, a standard interval arithmetic where one composes abstractions of each primitive function or operation ($T_f^\sharp$, $T_{op}^\sharp$) can never be more precise than Pasado.

Further, because Pasado uses the standard abstract transformers in the real (primal) part of the program, those bounds are at least as precise as standard interval arithmetic, thus Pasado's derivative bounds are never less precise than abstractly interpreting AD with the interval domain.

4.4.6 Generality

While we focus on the zonotope and interval abstract domains, Pasado is applicable to other abstract domains that can represent linear relationships symbolically. We now state how to use Pasado to obtain sound transformers for quadratic and polynomial zonotopes and the DeepPoly domains.

Theorem 4.5. The abstract transformers synthesized by Pasado are also sound transformers for quadratic [126] and polynomial zonotopes and for the DeepPoly [53] abstract domain.

Proof. As we only synthesize linear transformations of the input affine forms, if these affine forms were replaced with quadratic or polynomial forms, a linear transformation of them would still be a valid quadratic or polynomial term expressible in those domains. For adapting the chain rule to the DeepPoly domain we set the lower linear bound to be $a^{\geq} = Ax_1 + By_1 + (C - D)$ and the upper linear bound to be $a^{\leq} = Ax_1 + By_1 + (C + D)$ instead of returning an affine form and still use l, u for the interval bounds. Likewise for adapting the product and quotient rule transformers to the DeepPoly domain we set the lower linear bound as $a^{\geq} = Ax_1 + By_1 + Cx_2 + Dy_2 + (E - F)$ and the upper linear bound as $a^{\leq} = Ax_1 + By_1 + Cx_2 + Dy_2 + (E + F)$ and use l, u for the interval bounds. QED.

Thus, Pasado's combined support for multiple AD computational patterns, function primitives, and abstract domains, provides the necessary generality for synthesizing static analyzers for AD.

4.5 CASE STUDIES

We now present multiple Case Studies demonstrating the benefits of synthesizing precise abstract transformers tailored to the structure of both forward-mode AD (Sections 4.5.2 and 4.5.4) and reverse-mode AD (Sections 4.5.3 and 4.5.5).

4.5.1 Methodology

We describe our experimental setup. We ran the experiments in Sections 4.5.2 and 4.5.3 on a 10-core Apple M1 Pro SoC with 16 GB of unified memory. We ran the experiments in Sections 4.5.4 and 4.5.5 on a 32-core AMD Ryzen Threadripper PRO 3975WX CPU and an NVIDIA RTX A5000 GPU with 512 GB RAM. In all experiments, as baselines, we use AD with the interval domain [34] and AD with the zonotope domain [35], which collectively comprise the state of the art for abstracting AD.

Implementation We implement Pasado in Python, using a combination of the `affapy` library [132], the `micrograd` library [133] and PyTorch [37]. Additionally, select routines from Numpy [134] and Scikit-learn [135] were used to process results. The full artifact can be found at: `https://github.com/uiuc-arc/Pasado`. Pasado's implementation assumes ideal real arithmetic and is thus not floating-point sound (though floating-point sound versions of the operations exist [48]). While the part of the implementation using `affapy` supports all cases, the part written in PyTorch leverages the specific structure of DNNs and is thus only applicable to DNN benchmarks. For the experiments in Sections 4.5.2, 4.5.3, and 4.5.5 we implemented the baselines ourselves, as there was no existing code-base to use, however for Section 4.5.4 we used the implementation of [35] directly.

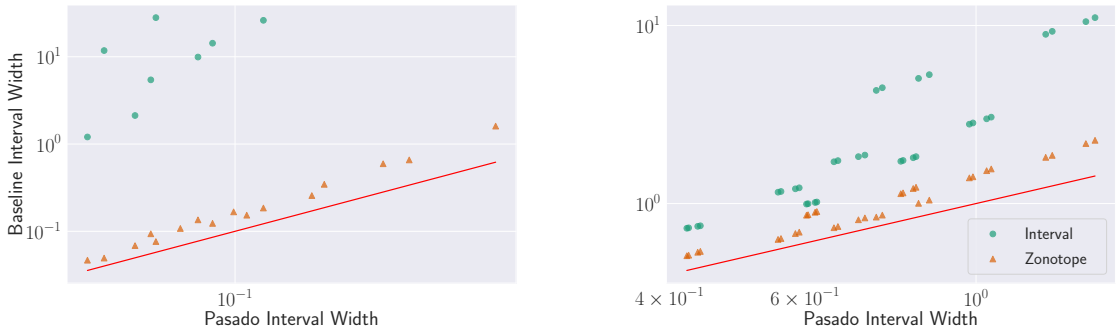
4.5.2 Robust Sensitivity Analysis of Ordinary Differential Equations

This first case study involves performing provable sensitivity analysis on the solutions of ODEs. As mentioned in the example section, to perform sensitivity analysis on the numerical solution of any ODE, one must automatically differentiate through the ODE solver. For these experiments we instantiate our technique with forward-mode AD. In our experiments we use a 4th-order Runge-Kutta numerical solver which is more complicated than the Euler method, but a more commonly used and accurate solver in practice. Hence for this evaluation `ODESolve` does not use Euler integration (unlike the example in Section 4.2). We now examine the following ODEs:

Chemistry ODE. This ODE is taken from [136, 137] and models the concentration of chemical species C_A as a function of time t. The ODE is parameterized by rate constants k_1 and k_{-1}.

$$\frac{dC_A}{dt} = -k_1 \cdot C_A + k_{-1} \cdot (C_0 - C_A) \tag{4.29}$$

However, instead of numerically solving the ODE given in Eq. 4.29, we train a neural network, NN, to learn the dynamics such that $NN(k_1, k_{-1}, C_0, C_A) \approx -k_1 \cdot C_A + k_{-1} \cdot (C_0 -$



(a) (b)

Figure 4.12: Scatter plots in *logarithmic* scales comparing the interval widths of the derivative bounds of (a) $\texttt{ODESolve}_{NN}$ with respect to k_1 and (b) $\texttt{ODESolve}_f$ with respect to T_0.

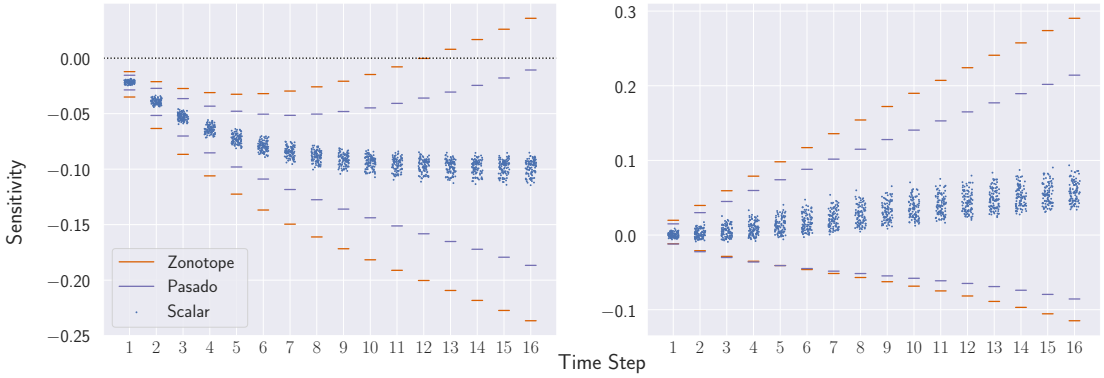


Figure 4.13: Bounds on the sensitivities of $\texttt{ODESolve}_{NN}$ with respect to k_1 (left) and k_{-1} (right). Each region between line segments of the same color represents the interval bound computed by that respective method's abstract AD. The dots represent the sensitivities evaluated at points sampled from the input intervals.

C_A), hence we will actually numerically solve the following *Neural* ODE:

$$\frac{dC_A}{dt} = NN(k_1, k_{-1}, C_0, C_A) \tag{4.30}$$

The neural ODE approximation produces nearly identical results, hence it serves as a useful surrogate model and also as a representative workload for ODEs where the underlying dynamics are some learned model. In [136, 137], the authors perform sensitivity analysis on this chemistry model using AD. However, those works computed sensitivities at scalar points only, hence we are the first to abstractly compute this derivative-based sensitivity analysis for *sets* of points. A key reason for performing the sensitivity analysis is to understand how sensitive the final concentration is to the rate constant parameters k_1 and k_{-1}. For this evaluation, we parameterize the ODE solver by the neural network dynamics function NN,

thus $\texttt{ODESolve}_{NN}$ denotes a 4^{th} order Runge-Kutta solver for Eq. 4.30. Thus we abstractly interpret AD to compute precise bounds on both $\frac{\partial}{\partial k_1}\texttt{ODESolve}_{NN}(k_1, k_{-1}, t_0, C_0, C_{A0}, h, n)$ and $\frac{\partial}{\partial k_{-1}}\texttt{ODESolve}_{NN}(k_1, k_{-1}, t_0, C_0, C_A, h, n)$, where h is the step size and n is the number of time steps.

In Fig. 4.12a, we plot the point $(u_{ours} - l_{ours}, u_{other} - l_{other})$, where the bounds l_{ours} and u_{ours} are with respect to k_1 and obtained from Pasado and l_{other} and u_{other} are the respective lower and upper bounds computed by the other method (regular intervals, regular zonotopes), for each input configuration and for each method. The red line denotes the identity function $y = x$, hence any plotted point that lies *above* the red line signifies that Pasado's bounds were tighter, as a point will lie above the line if and only if $\frac{u_{other}-l_{other}}{u_{ours}-l_{ours}} > 1$. Furthermore, both the x- and y-axes use logarithmic scales, hence even if points visually appear close together, the difference in precision may be substantial. The input ranges we use to generate this plot are all the 512 combinations of $k_1 = 3 \pm \delta_{k_1}$, $k_{-1} = 3 \pm \delta_{k_{-1}}$, $C_0 = 1 \pm \delta_{C_0}$, $C_A = 1 \pm \delta_{C_A}$, $t_0 = 0$, $n = \{8, 10, 12, 16\}$ and $h \in \{0.025, 0.1\}$, where $\delta_{k_1} \in \{0.05, 0.1, 0.15, 0.2\}$, $\delta_{k_{-1}} \in \{0.1, 0.2, 0.25, 0.3\}$, $\delta_{C_0} \in \{0.1, 0.2\}$, $\delta_{C_A} \in \{0.1, 0.2\}$, which are based on the ranges considered in [136].

Fig. 4.12a shows that in all input configurations, all the points for the baseline approaches lie above the red line, meaning that Pasado produces the most precise results. In all 512 cases the derivative bound computed with Pasado is strictly contained inside the bound computed via interval AD, and likewise in 497/512 cases the bound computed via Pasado is contained strictly inside the bound computed via zonotope AD (the bounds are incomparable in 15/512 cases). Only 20/512 green points are shown since in the other cases, the interval analysis generates results that are too over-approximate ($> 10^{20}$) to be meaningful. The *geometric mean* of precision improvement over all green points (comparing Pasado to interval AD) is 60.89 times, and the geometric mean of precision improvement over all orange points (comparing Pasado to zonotope AD) is 1.65 times.

We next focus on a specific input configuration for finer granularity. For our specific configuration we use the following ranges to perform the sensitivity analysis of the Chemical ODE: $k_1 \in [2.95, 3.05]$, $k_{-1} \in [2.8, 3.2]$, $t_0 = 0$, $C_0 \in [0.9, 1.1]$, $C_A \in [0.8, 1.2]$, $h = 0.025$, and $n = 16$, again based on the ranges considered in [136]. Fig. 4.13 illustrates the bounds on these sensitivities for the numerical solution of Eq. 4.29 with respect to k_1 and k_{-1}, with the x-axis representing the time steps and the y-axis representing the sensitivities. For each color, the upper and lower line segments represent the upper and lower bounds obtained by that method, respectively. We can see from both figures that Pasado always produce narrower bounds compared to the standard zonotope analysis. Additionally, within all the 16 time steps, as is shown in the left subfigure of Fig. 4.13, Pasado can formally prove

the monotonicity of $\texttt{ODESolve}_{NN}$ with respect to k_1 in this configuration (monotonically decreasing since the sensitivity is strictly less than 0), while the zonotope analysis cannot prove the monotonicity after the 12th step.

We measure the performance of the three methods by calculating the average runtimes for one specific input configuration over 16 time steps, since varying input bounds has negligible effects on runtimes. The average runtimes for the interval AD and zonotope AD are 0.12 and 81 seconds, respectively. Pasado takes 47 seconds on average, which is faster than zonotope AD - this improvement directly results from Pasado storing fewer noise symbols, which we found to be the biggest computational bottleneck of the `affapy` library.

Climate ODE. The Climate ODE is the same as in the example section and is taken from [123, 124] and models the global mean temperature through an energy balance model. In Fig. 4.12b, we plot the points representing $(u_{ours} - l_{ours}, u_{other} - l_{other})$ for the derivative of the numerical solution of Eq. 4.1 with respect to the initial condition T_0 following the same format as Fig. 4.12a. The different input configurations we use to generate this plot are all the 32 combinations of $T_0 \in \{337.5 \pm 37.5, 337.5 \pm 62.5\}$, $R = [2.65, 2.95]$, $Q \in \{342, 360 \pm 90\}$, $\alpha \in \{0.35, 0.325 \pm 0.025\}$, $e = [3.402224652 \times 10^{-8}, 5.103336978 \times 10^{-8}]$, $n \in \{8, 12\}$ and $h \in \{0.025, 0.05\}$, which are based on realistic ranges discussed in [124].

All of the points for the baseline approaches in Fig. 4.12b lie above the red line (Pasado), indicating that Pasado yields more accurate results in all cases, and additionally for all 32 of the configurations, the derivative bounds computed by Pasado are strictly contained within the bounds computed via interval AD and zonotope AD. The geometric mean of the precision improvement over all green points (comparing Pasado to interval AD) is approximately 2.85 times, and the geometric mean of the precision improvement over all orange points (comparing Pasado to zonotope AD) is approximately 1.31 times. Additionally, we observe that the benefit gained from Pasado becomes more pronounced for larger input intervals, as demonstrated by the greater vertical distance between the data points and the red line as we move further right along the x-axis.

Fig. 4.4 in Section 4.2 represents a plot of a specific input configuration and shows the bounds on the sensitivities of numerical solution of Eq. 4.1 with respect to the initial condition input T_0, formatted identically to each subfigure of Fig. 4.13. The specific input configuration used for the plot in Fig. 4.4 is $T_0 \in [275, 400]$, $R \in [2.65, 2.95]$, $Q = 342$, $\alpha = 0.3$, $e \in [3.402224652 \times 10^{-8}, 5.103336978 \times 10^{-8}]$, $h = 0.025$, and $n = 12$. Again, Pasado produces the most precise bounds among the three methods. Within all the 12 iterations, Pasado can show the monotonicity of the numerical solution with respect to the initial condition T_0.

We measure the performance of the three methods by calculating the average runtimes for

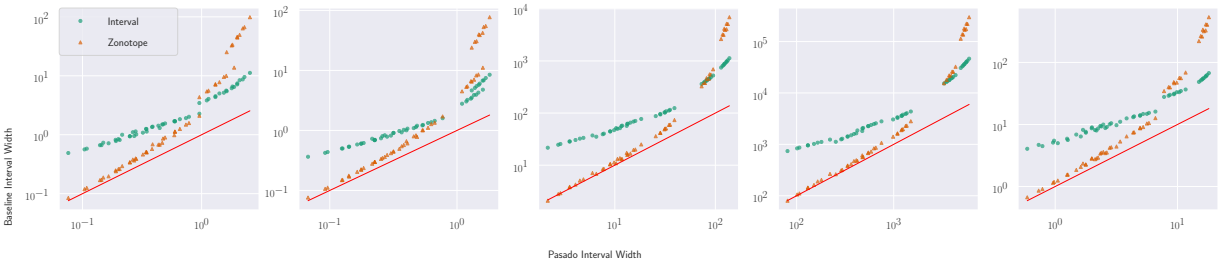


Figure 4.14: Scatter plots in *logarithmic* scales comparing the interval widths of the output bounds on the reverse-mode derivatives of the Black-Scholes solution with respect to K, S, σ, τ, and r, from left to right, respectively.

one specific input configuration over 12 time steps, since varying input bounds has negligible effects on runtimes. The average runtimes for the interval AD, zonotope AD, and Pasado are 0.0044, 0.068, and 0.25 seconds, respectively.

4.5.3 Black Scholes

In the next case study, we use our synthesized AD abstractions to compute bounds on the derivatives of the Black-Scholes solution with respect to the different parameters. The Black-Scholes model is a solution to a Partial Differential Equation (PDE) modeling financial option values [138]. The parameters of the Black-Scholes model are volatility σ, time τ, strike price K, spot price S and interest rate r. The derivatives of the Black-Scholes solution with respect to these parameters are commonly known as the greeks and while prior work has computed bounds on the greeks [116], only the interval domain was used, hence this experiment demonstrates the precision improvement obtained from using Pasado's synthesized abstract transformers. Since there are multiple inputs but only a single output, we elect for (abstract) reverse-mode AD.

For this experiment we compare our approach against both interval and zonotope abstract AD. We perform the evaluation for different ranges of the parameter values to study how varying the size of the input intervals affects the precision. The input ranges we use are all the 54 combinations of $K = 100 \pm \delta_K$, $S = 105 \pm \delta_S$, $\sigma = 5 \pm \delta_\sigma$, $\tau = 0.08219 \pm \delta_\tau$, $r = 0.0125 \pm \delta_r$, where $\delta_K \in \{1, 5, 10\}$, $\delta_\tau \in \{1, 5, 10\}$, $\delta_\sigma \in \{0.5, 1, 2\}$, $\delta_\tau \in \{0.001, 0.01\}$, $\delta_r = 0.001$. Furthermore, there are 5 greeks and 54 different configurations, hence there are 270 total derivative bounds computed.

We plot the points $(u_{ours} - l_{ours}, u_{other} - l_{other})$ in Fig. 4.14 for all the derivatives of the Black-Scholes solution with respect to the five greeks with the same layout as Fig. 4.12a. As can be seen, in all cases all points lie above the red line, hence demonstrating that Pasado

produces the most precise results. The bounds computed with Pasado are strictly contained inside the bounds computed with zonotope AD in 263/270 cases (rest are incomparable). The geometric mean of the precision improvement over all green points (comparing Pasado to interval AD) is approximately 4.03 times, and the geometric mean of the precision improvement over all orange points (comparing Pasado to zonotope AD) is approximately 2.81 times. Furthermore, for larger input intervals (further right along the x-axis) the improvement obtained from Pasado becomes more substantial, as evidenced by the higher vertical distance of the points above the line.

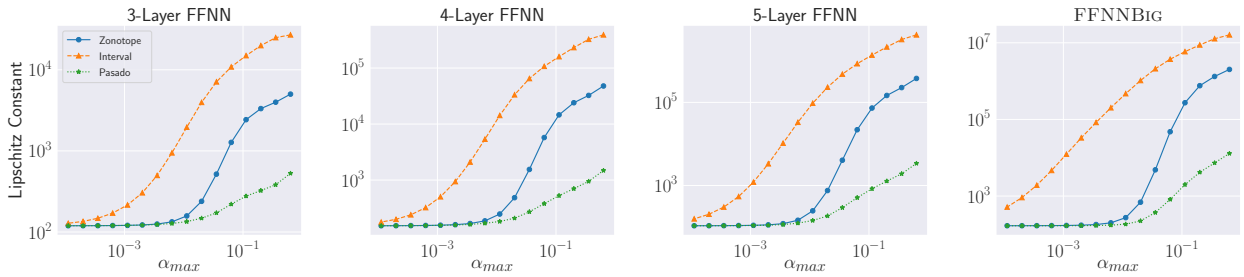
We measure the runtime performance of each method by calculating the average runtimes across all input configurations. The average runtimes (per configuration) for the interval AD, zonotope AD, and Pasado are 0.0015, 0.01, and 0.05 seconds, respectively.

4.5.4 Lipschitz Robustness of Neural Networks

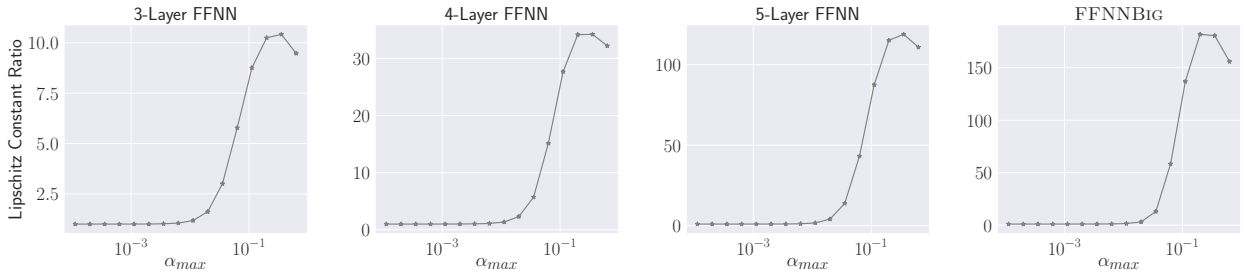
In this case study, we consider the task of computing the local Lipschitz constant of a neural network's output with respect to an adversarial perturbation parameter as in [34, 35]. We study DNNs trained on the MNIST dataset and consider the Haze perturbation given as $p_\alpha(x_i; \alpha) = (1 - \alpha)x_i + \alpha$, where α represents the amount of haze effects and x_i corresponds to the i^{th} input pixel. For the sake of direct comparison, we evaluate on the same range of values for the perturbation parameter α as in [35].

Fig. 4.15 shows the results of Lipschitz certification for feed-forward neural networks (referred to as FFNNs) with 3, 4, and 5 affine layers. The test accuracies for these three FFNNs are 92.7%, 89.2%, and 86.7%, respectively. Fig. 4.15a illustrates the bounds on the Lipschitz constants, where the x-axis represents the values of α_{max} and the y-axis represents the average upper bounds on the corresponding Lipschitz constants (smaller being preferable) for interval AD, zonotope AD, and Pasado. Fig. 4.15b exhibits the increase in precision of Pasado over zonotope AD, with the x-axis showing the values of α_{max} and the y-axis showing the *ratio* of the zonotope-bounded Lipschitz constants over the Pasado-bounded Lipschitz constants (larger being preferable). Given that the results produced by zonotope AD and Pasado are orders of magnitude smaller than those from interval AD for $\alpha_{max} > 10^{-2}$, we employ logarithmic scales for both the x- and the y-axes to enable a more visually clear separation between the curves representing zonotope AD and Pasado.

As shown in Fig. 4.15, there is a significant improvement in precision when using Pasado in comparison to using the interval AD or zonotope AD. In particular, for sufficiently large values of α_{max}, specifically where $\alpha_{max} > 10^{-2}$, the bounds on the Lipschitz constants for 5-layer network that we compute are between $4 - 100\times$ smaller than the bounds computed



(a) Average upper bounds on the local Lipschitz constants for the interval AD, zonotope AD, and Pasado.



(b) Increase in precisions of Pasado over zonotope AD.

Figure 4.15: Lipschitz robustness of FFNNs with 3 (far left), 4 (left-center), or 5 (right-center) affine layers and FFNNBIG (far right) against the haze perturbation on 1000 correctly classified test images. The top row (Fig. 4.15a) shows the average upper bounds on the local Lipschitz constant with respect to different α_{max} for the interval AD, zonotope AD, and Pasado. The bottom row (Fig. 4.15b) shows Pasado's precision increase, which is the ratio of the zonotope-bounded Lipschitz constant over the Pasado-bounded Lipschitz constant.

by the zonotope AD. In contrast, for smaller values of α_{max}, particularly when $\alpha_{max} < 10^{-3}$, Pasado and zonotope AD produce Lipschitz constants of nearly identical magnitudes.

We measure the performance of the three methods by calculating the average runtime across all input configurations used to generate the plots. For the 3-layer network, the average runtimes for the interval AD, zonotope AD, and Pasado are 0.00338, 0.00210, and 0.0636 seconds, respectively. For the 4-layer network, the average runtimes are 0.00383, 0.00352, and 0.0969 seconds, respectively. For the 5-layer network, the average runtimes are 0.00469, 0.00533, and 0.135 seconds, respectively.

The far right subfigure in Fig. 4.15 shows that Pasado is scalable to a larger FFNN, specifically the FFNNBIG architecture from [53]. The test accuracy for our FFNNBIG instance is 95.8%. As in previous plots, both the x- and the y-axes use logarithmic scales. Compared to the zonotope analysis, Pasado can be up to 182× more precise and the local Lipschitz constants can be up to 2.01×10^6 smaller. The average runtimes for interval AD, Zonotope AD and Pasado on the FFNNBIG network are 0.0542, 0.361, and 1.49 seconds,

respectively.

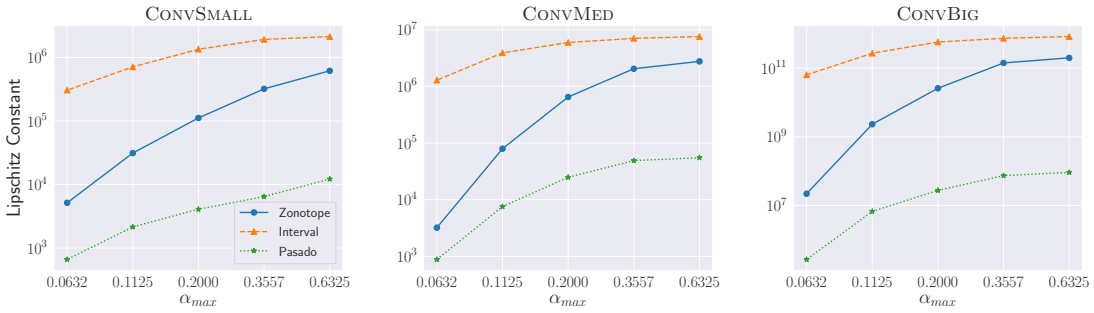
To further illustrate the scalability and versatility of Pasado, we also evaluate it on three convolutional neural networks (CNNs), namely ConvSmall, ConvMed, and ConvBig, as defined in [52], which are state-of-the-art CNN benchmarks for verification. When the convolutional layers of these networks are unrolled into an equivalent affine layer, the corresponding number of intermediate neurons is more than $25,000$, which means Pasado's chain rule abstract transformer will be called that number of times, hence these benchmarks highlight Pasado's scalability.

Fig. 4.16 presents the results of Lipschitz robustness analysis for ConvSmall, ConvMed, and ConvBig. The test accuracies for these three CNNs are 98.5%, 99.2%, and 98.9%, respectively. As in previous plots, we use logarithmic scales for clarity. Across all three CNNs, Pasado generates much more precise bounds. Notably, in the ConvBig network, Pasado can offer up to 2750× greater precision, and the local Lipschitz constants can be up to 1.99×10^{11} smaller. For ConvSmall, the average runtimes for the interval analysis, zonotope analysis, and Pasado are 3.38, 3.64, and 4.70 seconds, respectively. For ConvMed, the average runtimes are 5.11, 5.70, and 7.27 seconds, respectively. For ConvBig, the average runtimes are 115, 212, and 191 seconds, respectively.

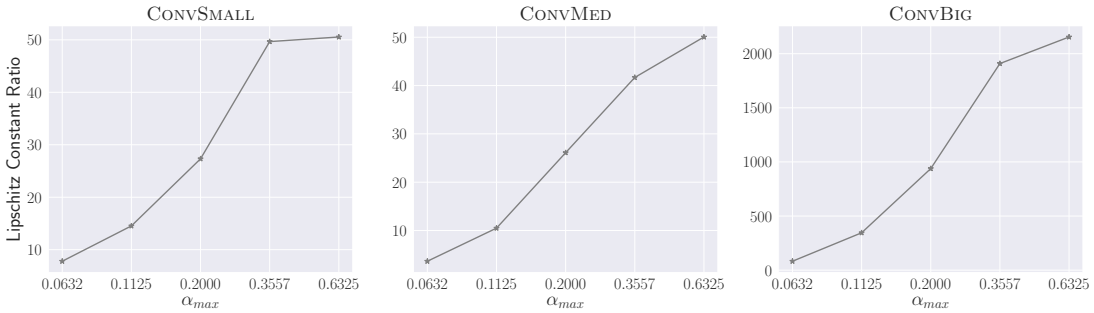
In summary, we observe that as the network architectures become larger, the precision improvements offered by Pasado become more pronounced. Furthermore, in the largest convolutional benchmark, ConvBig, the runtime of Pasado was actually *faster* than the standard zonotope AD analysis baseline. This behavior is the consequence of Pasado generating fewer noise symbols compared to standard zonotope transformers, similar to the observation in the ODE benchmarks of Section 4.5.2. Hence, for sufficiently large benchmarks (like CNNs), the computational savings obtained by propagating fewer noise symbols (as Pasado does) outweigh the costs (e.g. due to the linear regression) associated with Pasado's more precise abstract transformers.

4.5.5 Monotonicity Analysis of an Adult Income Network

In this case study, we conduct a monotonicity analysis on a multilayer perceptron (MLP) trained on the Adult dataset [139]. Our MLP takes 87 input features (where 81 of the 87 features result from one-hot encodings of the original dataset's categorical variables), passes these features through two hidden layers (each containing 10 neurons and applying tanh activation), and outputs a single binary classification score predicting the income level. Our goal is to verify the monotonicity (both increasing and decreasing) of the MLP's output with respect to 5 continuous input features which are: *Age*, *Education-Num*, *Capital*



(a) Average upper bounds on the local Lipschitz constants for the interval AD, zonotope AD, and Pasado.



(b) Increase in precisions of Pasado over zonotope AD.

Figure 4.16: Lipschitz robustness of ConvSmall (left), ConvMed (center), and ConvBig (right) against the haze perturbation on 30 correctly classified test-set images. The top row (Fig. 4.16a) presents the average upper bounds on the local Lipschitz constant with respect to different α_{max} for the interval AD, zonotope AD, and Pasado. The bottom row (Fig. 4.16b) presents the increase in precision of the Pasado domain, computed as the ratio of the zonotope-bounded Lipschitz constant over the Pasado-bounded Lipschitz constant.

Gain, *Capital Loss*, and *Hours per week*. Whereas prior work [22] varied one feature at a time while holding the value of all other features as fixed, our experiments allow all 5 of the aforementioned continuous features to simultaneously vary within interval bounds. Hence we abstractly interpret the continuous features with a 5D L_∞-ball, with a radius $\epsilon \in [0, 1]$, while holding all the remaining features as fixed. Since training data is normalized to have zero mean and unit variance, passing a 5D L_∞-ball with $\epsilon = 0.4$ through the MLP is equivalent to exploring an infinite set of inputs that satisfy *Age* $\in [33.2, 44.1]$, *Education-Num* $\in [9.05, 11.1]$, *Capital Gain* $\in [-1900, 4060]$, *Capital Loss* $\in [-73.7, 249]$, and *Hours per week* $\in [35.5, 45.4]$. For this analysis, we used Pasado's reverse-mode AD abstract transformers. Hence in a single (abstract) pass, Pasado computes bounds on the partial derivatives of the output with respect to each of the five input features.

For each L_∞-ball radius ϵ, Pasado abstractly computes bounds on the five partial derivatives when the original input is perturbed by the L_∞-ball for 100 different inputs, computing 500 partial derivative bounds in total. In addition, we compare Pasado against interval AD

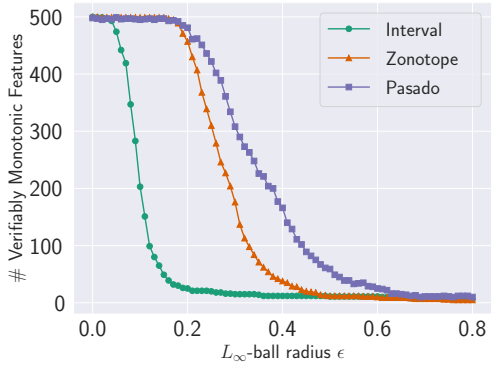


Figure 4.17: Counts of verifiably monotonic features of Adult MLP over 100 test-set inputs.

and zonotope AD. For a given input L_∞-ball, to verify monotonicity with respect to a chosen feature, the partial derivative bound with respect to that feature should provably exclude 0, meaning the interval should be strictly positive (monotonically increasing) or strictly negative (monotonically decreasing). This condition ensures that the MLP is monotonic with respect to that feature for *all* input points in the given L_∞-ball. Hence in Fig. 4.17, we show the total number of partial derivative bounds that exclude 0 over 100 test inputs, for different-sized L_∞-balls.

Fig. 4.17 shows that the ability of interval AD to prove monotonicity sharply decreases for $\epsilon \geq 0.05$ due to the inherent imprecision of the interval domain. For small ϵ such as $0 \leq \epsilon \leq 0.2$, zonotope AD and Pasado produce similar counts, meaning both can prove monotonicity. However, their respective performances diverge as ϵ increases. When $0.2 \leq \epsilon \leq 0.6$, the counts for zonotope AD decline rapidly to nearly zero, whereas the counts for Pasado remain high. Hence, in these cases Pasado can prove monotonicity in significantly more instances. For $\epsilon > 0.6$, all three analyses struggle to prove monotonicity for most continuous input features. The average runtimes for the interval AD, zonotope AD, and Pasado are 0.079, 12, and 39 seconds, respectively. In summary, Pasado proves the most monotonicity specifications across all inputs.

4.6 RELATED WORK

Composite and Synthesized Abstractions. The idea of abstracting a composite numeric expression all at once (as Pasado does) to obtain better precision than sub-optimally composing individual abstract transformers for nonlinear primitives has emerged in the literature. While beneficial for improving precision, abstracting composite expressions is challenging because soundly bounding a composite expression typically involves solving a non-

convex, multivariable optimization problem. As discussed in Fryazinov et al. [111], these problems typically cannot be solved by hand due to the need to consider interior critical points (as Pasado does). Hence recent works [109, 121, 140, 141, 142, 143] have tried to *synthesize* precise composite abstractions by solving an optimization problem (using gradient methods or SMT solvers). However, none of these works target AD, thus none of these techniques can leverage the structure of an AD computation as Pasado does. Further, unlike [141, 142] we do not use an SMT solver, as we can solve our optimization problems directly.

Abstract Interpretation of AD. Abstract interpretation has also been applied to AD [24, 34, 35, 93, 116], however, all of these prior works use either the standard interval domain or standard zonotope domain abstract transformers, and compose them naively instead of jointly abstracting multiple AD operations as we do. Furthermore, [34, 93, 116] are only formalized for a single abstract domain and thus cannot immediately produce general static analyzers for a more general set of abstract domains as Pasado can. While [35] can be instantiated with different abstract domains, that work uses standard abstract transformers, cannot dynamically synthesize new abstractions, and only supports forward mode AD.

4.7 SUMMARY

We presented Pasado, the first technique for synthesizing precise static analyzers tailored specifically to Automatic Differentiation. We show the generality of Pasado by instantiating it for the Product Rule, Quotient Rule and Chain Rule patterns, with the latter supporting multiple different non-linear functions. Pasado's generality also extends to multiple different abstract domains and both forward-mode and reverse-mode AD. Our evaluation on multiple challenging scenarios from machine learning and scientific computing shows that Pasado significantly improves precision compared to prior techniques while simultaneously offering scalability to large computations including CNNs.

CHAPTER 5: CONCLUSIONS AND FUTURE WORK

5.1 CONCLUSIONS

The need to compute derivatives pervades Computer Science: from Machine Learning, to Scientific Computing to Graphics and beyond. To compute these derivatives that are so ubiquitous, programmers rely upon differentiable programming languages and automatic differentiation frameworks. However, programmers have a responsibility to ensure that the differentiable programs they write satisfy their desired properties. In light of this obligation, there exists a core need for automated program analyses for differentiable programming.

This need for automated differentiable program analyses is complicated by the fact that differentiable programs have challenging semantics, use highly nonlinear operations, and involve thousands to millions of program variables. Thus, ensuring that the analysis is general, precise and scalable becomes the primary concern. To tackle these challenges and address this need, this dissertation presented the first steps towards building a unified framework to obtain precise, general and scalable static analyses of differentiable programs.

First, in Chapter 2, I described DeepJ which is the first abstract interpretation of Clarke Generalized Jacobians. By defining an abstract semantics based upon generalized derivatives, DeepJ can analyze differentiable programs containing points of non-differentiability caused by branches in the program's control flow.

Next, in Chapter 3, I described a general construction for building static analyzers for differentiable programs with higher derivatives. Due to the generality of this construction, it can be instantiated for any order of derivative and for a broad class of numeric abstract domains (intervals, zonotopes, polyhedra).

Lastly, in Chapter 4, I described Pasado, the first framework for synthesizing precise static analyzers for the chain rule, product rule and quotient rule operations of automatic differentiation. Not only is Pasado orders of magnitude more precise than prior work, but Pasado scalably analyzes derivatives from large convolutional neural networks.

5.2 FUTURE WORK

While this dissertation represents a first step toward the goal of precise, general and scalable program analyses for differentiable programming, many more interesting questions remain. Moving forward, there are opportunities to further the scope of foundational differentiable program analysis techniques and opportunities to apply the existing techniques

outlined in this thesis towards new, emerging application domains. We describe these future directions below.

5.2.1 Program Analyses for Differentiable Scientific Computing

Given that scientific computing researchers were the original developers of AD, scientific computing which includes scientific ML, represents an attractive application for the program analyses developed in this dissertation. Chapter 4 which describes a robust AD-based sensitivity analysis of ODE solutions for proving monotonicity with respect to physical (e.g., atmospheric) parameters represents a step in this direction. On a similar note, other researchers have begun using derivative bounds to certify the correctness of scientific ML models used to simulate physics phenomena [25], and showed a direct connection between the derivatives and the model's desired physical behavior. However, a gap still exists between the programming languages and formal methods community and the scientific computing community in terms of how to best specify the desired formal properties of scientific models. For instance, scientific computing requires new notions of robustness and formal correctness that differ from the previous robustness and correctness goals of DNN verification. This difference stems from the fact that scientific computing programs are susceptible to different issues (e.g., numerical blowups) than DNNs are. Furthermore, scientific computations must obey properties that capture the underlying physics of the model. The question of how to embed such requirements into an automated analysis remains severely understudied. Thus establishing correctness properties of scientific applications in terms of derivatives will become an important problem moving forward.

Additionally, scientific computing software often contains complicated nonlinear, numerical expressions. While those expressions share similarities with the patterns described in this dissertation, there are key differences, such as complex control flow and sparsity, that present new challenges. Hence, naively applying an abstract interpreter to these challenging, nonlinear expressions, would inevitably lead to large imprecision. Hence this problem remains open and an interesting future application of the techniques described in this dissertation.

5.2.2 Foundational Differentiable Program Analyses

In addition to the abstract interpreters I developed, AD can benefit from other verification techniques. For instance, Hoare logic lets programmers *work backwards* and reason about the set of inputs that lead to desired outputs. Thus, by adapting Hoare logic to AD, one could answer questions like "*what is the largest set of inputs for which a neural network*

is still guaranteed to behave monotonically?". This effort will likely require incorporating derivative information into solvers. Hence, verification techniques such as Hoare logics, type systems, or SMT solvers offer additional pathways to develop foundational program analysis of differentiable programs.

In addition, expanding the scope of static analysis techniques to support more expressive features of AD represents another interesting direction. For instance, popular AD functionalities like checkpointing schemes remain understudied by the PL community. Additionally, due to their complicated and often undefined semantics, it remains an open question how to best adapt existing static analyses to support these features.

5.2.3 Improving Differentiable Programming Software

In addition to the program analysis approaches developed in this dissertation, many opportunities exist to develop more empirical, testing based-approaches. For instance the results of abstractly interpreting an AD computation could then be used as an oracle for a testing tool. While some work on software testing of differentiable programming exists, in particular [144, 145], the intersection of Software Engineering, Differentiable Programming, and Automatic Differentiation remains severely understudied.

REFERENCES

[1] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, no. POPL, 1977.

[2] R. E. Wengert, "A simple automatic derivative evaluation program," *Communications of the ACM*, vol. 7, no. 8, pp. 463–464, 1964.

[3] B. Speelpenning, *Compiling fast partial derivatives of functions given by algorithms*. University of Illinois at Urbana-Champaign, 1980.

[4] S. Linnainmaa, "Taylor expansion of the accumulated rounding error," *BIT Numerical Mathematics*, vol. 16, no. 2, pp. 146–160, 1976.

[5] A. Griewank and A. Walther, *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008.

[6] A. Mametjanov, B. Norris, X. Zeng, B. Drewniak, J. Utke, M. Anitescu, and P. Hovland, "Applying automatic differentiation to the community land model," in *Recent Advances in Algorithmic Differentiation*, 2012.

[7] C. Bendtsen and O. Stauning, "Fadbad, a flexible c++ package for automatic differentiation," 1996.

[8] Y. Ma, V. Dixit, M. J. Innes, X. Guo, and C. Rackauckas, "A comparison of automatic differentiation and continuous sensitivity analysis for derivatives of differential equation solutions," in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2021, pp. 1–9.

[9] P. D. Hovland, B. Norris, M. M. Strout, S. Bhowmick, and J. Utke, "Sensitivity analysis and design optimization through automatic differentiation," in *Journal of Physics: Conference Series*, 2005.

[10] J. Lin, C. Gan, and S. Han, "Defensive quantization: When efficiency meets robustness," in *International Conference on Learning Representations*, 2019.

[11] Y. Tsuzuku, I. Sato, and M. Sugiyama, "Lipschitz-margin training: scalable certification of perturbation invariance for deep neural networks," in *Neural Information Processing Systems*, 2018.

[12] D. Alvarez-Melis and T. S. Jaakkola, "Towards robust interpretability with self-explaining neural networks," in *Neural Information Processing Systems*, 2018.

[13] C. Dwork, M. Hardt, T. Pitassi, O. Reingold, and R. Zemel, "Fairness through awareness," in *Proceedings of the 3rd innovations in theoretical computer science conference*, 2012.

[14] C. Dwork, F. McSherry, K. Nissim, and A. Smith, "Calibrating noise to sensitivity in private data analysis," in *Theory of cryptography conference*, 2006.

[15] H. Zhang, P. Zhang, and C. Hsieh, "Recurjac: An efficient recursive algorithm for bounding jacobian matrix of neural networks and its applications," in *The 33rd AAAI Conference on Artificial Intelligence, (AAAI)*, 2019.

[16] J. Deussen, "Global derivatives," Ph.D. dissertation, 2021.

[17] Y. Wang, T. Zhang, X. Guo, and Z. Shen, "Gradient based feature attribution in explainable ai: A technical review," *arXiv preprint arXiv:2403.10415*, 2024.

[18] M. Ancona, E. Ceolini, C. Öztireli, and M. Gross, "Towards better understanding of gradient-based attribution methods for deep neural networks," in *6th International Conference on Learning Representations (ICLR)*, 2018.

[19] J. D. Janizek, P. Sturmfels, and S.-I. Lee, "Explaining explanations: Axiomatic feature interactions for deep networks," *Journal of Machine Learning Research*, vol. 22, 2021.

[20] S. Lerman, C. Venuto, H. Kautz, and C. Xu, "Explaining local, global, and higher-order interactions in deep learning," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021.

[21] X. Liu, X. Han, N. Zhang, and Q. Liu, "Certified monotonic neural networks," *Advances in Neural Information Processing Systems*, vol. 33, pp. 15 427–15 438, 2020.

[22] Z. Shi, Y. Wang, H. Zhang, Z. Kolter, and C.-J. Hsieh, "Efficiently computing local lipschitz constants of neural networks via bound propagation," in *Advances in Neural Information Processing Systems*, 2022.

[23] A. Sivaraman, G. Farnadi, T. Millstein, and G. Van den Broeck, "Counterexample-guided learning of monotonic neural networks," *Neural Information Processing Systems*, 2020.

[24] A. Misra, J. Laurel, and S. Misailovic, "Vix: Analysis-driven compiler for efficient low-precision variational inference," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2023.

[25] F. Eiras, A. Bibi, R. R. Bunel, K. D. Dvijotham, P. Torr, and M. P. Kumar, "Efficient error certification for physics-informed neural networks," in *Forty-first International Conference on Machine Learning*.

[26] Z. Qin, T.-W. Weng, and S. Gao, "Quantifying safety of learning-based self-driving control using almost-barrier functions," in *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2022, pp. 12 903–12 910.

[27] H. G. Rice, "Classes of recursively enumerable sets and their decision problems," *Transactions of the American Mathematical society*, vol. 74, no. 2, pp. 358–366, 1953.

[28] A. Miné, "Tutorial on static inference of numeric invariants by abstract interpretation," *Foundations and Trends in Programming Languages*, vol. 4, 2017.

[29] P. Cousot and N. Halbwachs, "Automatic discovery of linear restraints among variables of a program," in *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, no. POPL, 1978.

[30] G. Singh, M. Püschel, and M. Vechev, "Fast polyhedra abstract domain," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, 2017.

[31] M. Abadi and G. D. Plotkin, "A simple differentiable programming language," *Symposium on Principles of Programming Languages (POPL)*, 2019.

[32] W. Lee, H. Yu, X. Rival, and H. Yang, "On correctness of automatic differentiation for non-differentiable functions," in *Neural Information Processing Systems*, 2020.

[33] J. Laurel, S. B. Qian, G. Singh, and S. Misailovic, "Abstract interpretation of automatic differentiation," *Languages for Inference Workshop (LAFI)*, 2024.

[34] J. Laurel, R. Yang, G. Singh, and S. Misailovic, "A dual number abstraction for static analysis of clarke jacobians," *Proceedings of the ACM on Programming Languages*, no. POPL, 2022.

[35] J. Laurel, R. Yang, S. Ugare, R. Nagel, G. Singh, and S. Misailovic, "A general construction for abstract interpretation of higher-order automatic differentiation," *Proceedings of the ACM on Programming Languages*, vol. 6, no. OOPSLA2, pp. 1007–1035, 2022.

[36] J. Laurel, S. B. Qian, G. Singh, and S. Misailovic, "Synthesizing precise static analyzers for automatic differentiation," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA2, pp. 1964–1992, 2023.

[37] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga et al., "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[38] B. Sherman, J. Michel, and M. Carbin, "λ_s: Computable semantics for differentiable programming with higher-order functions and datatypes," in *Symposium on Principles of Programming Languages (POPL)*, 2021.

[39] F. Clarke, "2. generalized gradients," in *Optimization and Nonsmooth Analysis*. Society for Industrial and Applied Mathematics, 1990, pp. 24–109.

[40] A. Edalat and M. Maleki, "Differentiation in logical form," in *32nd ACM/IEEE Symposium on Logic in Computer Science (LICS)*, 2017.

[41] R. Mangal, K. Sarangmath, A. V. Nori, and A. Orso, "Probabilistic lipschitz analysis of neural networks," in *International Static Analysis Symposium*, 2020.

[42] M. Jordan and A. G. Dimakis, "Exactly computing the local lipschitz constant of relu networks," *Neural Information Processing Systems*, 2020.

[43] K. A. Khan and P. I. Barton, "Evaluating an element of the clarke generalized jacobian of a composite piecewise differentiable function," *ACM Transactions on Mathematical Software (TOMS)*, vol. 39, no. 4, 2013.

[44] P. Di Gianantonio and A. Edalat, "A language for differentiable functions," in *International Conference on Foundations of Software Science and Computational Structures*, 2013.

[45] A. Edalat and M. Maleki, "Differential calculus with imprecise input and its logical framework," in *International Conference on Foundations of Software Science and Computation Structures*, 2018.

[46] A. Edalat, A. Lieutier, and D. Pattinson, "A computational model for multi-variable differential calculus," *Information and Computation*, 2013.

[47] A. Edalat and A. Lieutier, "Domain theory and differential calculus (functions of one variable)," *Mathematical Structures in Computer Science*, 2004.

[48] A. Miné, "Relational abstract domains for the detection of floating-point run-time errors," in *European Symposium on Programming*, 2004.

[49] A. Krizhevsky, G. Hinton et al., "Learning multiple layers of features from tiny images," 2009.

[50] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, 1998.

[51] H. Gouk, E. Frank, B. Pfahringer, and M. J. Cree, "Regularisation of neural networks by enforcing lipschitz continuity," *Machine Learning*, vol. 110, 2021. [Online]. Available: https://github.com/henrygouk/keras-lipschitz-networks

[52] G. Singh, T. Gehr, M. Mirman, M. Püschel, and M. T. Vechev, "Fast and effective robustness certification." *NeurIPS*, 2018.

[53] G. Singh, T. Gehr, M. Püschel, and M. Vechev, "An abstract domain for certifying neural networks," *Symposium on Principles of Programming Languages (POPL)*, 2019.

[54] C. Urban and A. Miné, "A review of formal methods applied to machine learning," *arXiv preprint arXiv:2104.02466*, 2021.

[55] C. Müller, F. Serre, G. Singh, M. Püschel, and M. Vechev, "Scaling polyhedral neural network verification on gpus," in *Machine Learning and Systems (MLSys)*, vol. 3, 2021.

[56] M. Mirman, T. Gehr, and M. Vechev, "Differentiable abstract interpretation for provably robust neural networks," in *International Conference on Machine Learning*, 2018.

[57] M. Balunović, M. Baader, G. Singh, T. Gehr, and M. Vechev, "Certifying geometric robustness of neural networks," *Neural Information Processing Systems*, 2019.

[58] M. Mirman, G. Singh, and M. T. Vechev, "A provable defense for deep residual networks," *CoRR*, vol. abs/1903.12519, 2019.

[59] H. Zhang, H. Chen, C. Xiao, S. Gowal, R. Stanforth, B. Li, D. Boning, and C.-J. Hsieh, "Towards stable and efficient training of verifiably robust neural networks," in *International Conference on Learning Representations*, 2020.

[60] B. Zhang, T. Cai, Z. Lu, D. He, and L. Wang, "Towards certifying l-infinity robustness using neural networks with l-inf-dist neurons," in *International Conference on Machine Learning*, 2021.

[61] C. Paterson, H. Wu, J. Grese, R. Calinescu, C. S. Pasareanu, and C. Barrett, "Deepcert: Verification of contextually relevant robustness for neural network image classifiers," 2021.

[62] W. Clifford, "Preliminary sketch of biquaternions," *Proceedings of The London Mathematical Society*, 1873.

[63] J. Grünwald, "Über duale zahlen und ihre anwendung in der geometrie," *Monatshefte für Mathematik und Physik*, vol. 17, pp. 81–136, 1906.

[64] S. Scholtes, "Piecewise differentiable functions," in *Introduction to Piecewise Differentiable Equations*, 2012.

[65] A. Griewank, "On stable piecewise linearization and generalized algorithmic differentiation," *Optimization Methods and Software*, 2013.

[66] S. Chaudhuri, S. Gulwani, and R. Lublinerman, "Continuity analysis of programs," in *Symposium on Principles of Programming Languages (POPL)*, 2010.

[67] D. Richardson, "Some undecidable problems involving elementary functions of a real variable," *The Journal of Symbolic Logic*, 1969.

[68] T. Beck and H. Fischer, "The if-problem in automatic differentiation," *Journal of Computational and Applied Mathematics*, 1994.

[69] R. Moore, R. B. Kearfott, and M. Cloud, *Introduction to Interval Analysis*, ch. 7. Interval Matrices.

[70] L. H. De Figueiredo and J. Stolfi, "Affine arithmetic: concepts and applications," *Numerical Algorithms*, 2004.

[71] E. Darulova and V. Kuncak, "Towards a compiler for reals," *ACM Trans. Program. Lang. Syst.*, 2017.

[72] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen, "The essence of compiling with continuations," in *Proceedings of the 1993 Conference on Programming Language Design and Implementation (PLDI)*, 1993.

[73] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[74] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Reluplex: An efficient smt solver for verifying deep neural networks," in *International Conference on Computer Aided Verification*, 2017.

[75] V. Fernando, K. Joshi, J. Laurel, and S. Misailovic, "Diamont: dynamic monitoring of uncertainty for distributed asynchronous programs," *International Journal on Software Tools for Technology Transfer*, vol. 25, no. 4, pp. 521–539, 2023.

[76] R. Ehlers, "Formal verification of piece-wise linear feed-forward neural networks," in *International Symposium on Automated Technology for Verification and Analysis*, 2017.

[77] J. Laurel, R. Yang, A. Sehgal, S. Ugare, and S. Misailovic, "Statheros: Compiler for efficient low-precision probabilistic programming," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 787–792.

[78] V. Fernando, K. Joshi, J. Laurel, and S. Misailovic, "Diamont: Dynamic monitoring of uncertainty for distributed asynchronous programs," *2021 in Runtime Verification*, 2021.

[79] Z. Huang, S. Dutta, and S. Misailovic, "Aqua: Automated quantized inference for probabilistic programs," in *Automated Technology for Verification and Analysis: 19th International Symposium, ATVA 2021, Gold Coast, QLD, Australia, October 18–22, 2021, Proceedings 19*. Springer, 2021, pp. 229–246.

[80] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu, "Safety verification of deep neural networks," in *International Conference on Computer Aided Verification*, 2017.

[81] G. Katz, D. A. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljić et al., "The marabou framework for verification and analysis of deep neural networks," in *International Conference on Computer Aided Verification*, 2019.

[82] M. Sotoudeh and A. V. Thakur, "Abstract neural networks," in *International Static Analysis Symposium*, 2020.

[83] K. A. Khan and P. I. Barton, "Evaluating an element of the clarke generalized jacobian of a piecewise differentiable function," in *Recent Advances in Algorithmic Differentiation*, 2012.

[84] S. Chaudhuri, S. Gulwani, R. Lublinerman, and S. Navidpour, "Proving programs robust," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, 2011.

[85] S. Chaudhuri and A. Solar-Lezama, "Smoothing a program soundly and robustly," in *International Conference on Computer Aided Verification*, 2011.

[86] S. Chaudhuri and A. Solar-Lezama, "Smooth interpretation," in *Proceedings of the 31st Conference on Programming Language Design and Implementation (PLDI)*, 2010.

[87] J. Laurel and S. Misailovic, "Continualization of probabilistic programs with correction." in *29th European Symposium on Programming (ESOP), 2020.*, 2020.

[88] K. Scaman and A. Virmaux, "Lipschitz regularity of deep neural networks: analysis and efficient estimation," in *Neural Information Processing Systems*, 2018.

[89] T.-W. Weng, H. Zhang, P.-Y. Chen, J. Yi, D. Su, Y. Gao, C.-J. Hsieh, and L. Daniel, "Evaluating the robustness of neural networks: An extreme value theory approach," in *International Conference on Learning Representations*, 2018.

[90] T.-W. Weng, H. Zhang, H. Chen, Z. Song, C.-J. Hsieh, D. Boning, I. S. Dhillon, and L. Daniel, "Towards fast computation of certified robustness for relu networks," in *International Conference on Machine Learning*, 2018.

[91] J. Hückelheim, Z. Luo, S. H. K. Narayanan, S. Siegel, and P. D. Hovland, "Verifying properties of differentiable programs," in *International Static Analysis Symposium*, 2018.

[92] F. Krawiec, N. Krishnaswami, S. Peyton Jones, T. Ellis, A. Fitzgibbon, and R. Eisenberg, "Provably correct, asymptotically efficient, higher-order reverse-mode automatic differentiation," *Proceedings of the ACM on Programming Languages*, no. POPL, 2022.

[93] M. Jordan and A. Dimakis, "Provable lipschitz certification for generative models," in *International Conference on Machine Learning*. PMLR, 2021, pp. 5118–5126.

[94] F. Immler, "A verified ode solver and smale's 14th problem," Ph.D. dissertation, Technische Universität München, 2018.

[95] J. Karczmarczuk, "Functional differentiation of computer programs," *Higher-order and symbolic computation*, 2001.

[96] B. A. Pearlmutter and J. M. Siskind, "Lazy multivariate higher-order forward-mode ad," in *Symposium on Principles of Programming Languages*, no. POPL, 2007.

[97] J. Laurel, R. Yang, S. Ugare, R. Nagel, G. Singh, and S. Misailovic, "Artifact for a general construction for abstract interpretation of higher-order automatic differentiation," 2022.

[98] F. F. Di Bruno, "Note sur une nouvelle formule de calcul différentiel," *Quarterly J. Pure Appl. Math*, 1857.

[99] P. J. Olver, *Applications of Lie groups to differential equations*. Springer Science & Business Media, 1993, vol. 107.

[100] T. Fel, M. Ducoffe, D. Vigouroux, R. Cadène, M. Capelle, C. Nicodème, and T. Serre, "Don't lie to me! robust and efficient explainability with verified perturbation analysis," *arXiv preprint arXiv:2202.07728*, 2022.

[101] K. Ghorbal, E. Goubault, and S. Putot, "The zonotope abstract domain taylor1+," in *International Conference on Computer Aided Verification*, no. CAV, 2009.

[102] J. Stolfi and L. H. de Figueiredo, "An introduction to affine arithmetic," *Trends in Computational and Applied Mathematics*, vol. 4, 2003.

[103] A. Griewank, J. Utke, and A. Walther, "Evaluating higher derivative tensors by forward propagation of univariate taylor series," *Mathematics of computation*, 2000.

[104] A. Paszke, D. D. Johnson, D. Duvenaud, D. Vytiniotis, A. Radul, M. J. Johnson, J. Ragan-Kelley, and D. Maclaurin, "Getting to the point: index sets and parallelism-preserving autodiff for pointful array programming," *Proceedings of the ACM on Programming Languages*, no. ICFP, 2021.

[105] J. Fike and J. Alonso, "The development of hyper-dual numbers for exact second-derivative calculations," *AIAA*, 2011.

[106] H. He, "The state of machine learning frameworks in 2019," *The Gradient*, 2019.

[107] A. Miné, "Symbolic methods to enhance the precision of numerical abstract domains," in *International Workshop on Verification, Model Checking, and Abstract Interpretation*, 2006.

[108] Z. Shi, H. Zhang, K.-W. Chang, M. Huang, and C.-J. Hsieh, "Robustness verification for transformers," in *International Conference on Learning Representations*, no. ICLR, 2019.

[109] W. Ryou, J. Chen, M. Balunovic, G. Singh, A. Dan, and M. Vechev, "Scalable polyhedral verification of recurrent neural networks," in *International Conference on Computer Aided Verification*, 2021.

[110] A. Albarghouthi, "Introduction to neural network verification," *Foundations and Trends® in Programming Languages*, 2021.

[111] O. Fryazinov, A. Pasko, and P. Comninos, "Fast reliable interrogation of procedurally defined implicit surfaces using extended revised affine arithmetic," *Computers & Graphics*, vol. 34, no. 6, 2010.

[112] B. Jeannet and A. Miné, "Apron: A library of numerical abstract domains for static analysis," in *International Conference on Computer Aided Verification*, 2009, pp. 661–667.

[113] R. Yang, J. Laurel, S. Misailovic, and G. Singh, "Provable defense against geometric transformations," in *11th International Conference on Learning Representations*, 2023.

[114] J. Bettencourt, M. J. Johnson, and D. Duvenaud, "Taylor-mode automatic differentiation for higher-order derivatives in jax," 2019.

[115] M. Huot, S. Staton, and M. Vákár, "Higher order automatic differentiation of higher order functions," *arXiv preprint arXiv:2101.06757*, 2021.

[116] V. Vassiliadis, J. Riehme, J. Deussen, K. Parasyris, C. D. Antonopoulos, N. Bellas, S. Lalis, and U. Naumann, "Towards automatic significance analysis for approximate computing," in *2016 IEEE/ACM International Symposium on Code Generation and Optimization*, no. CGO, 2016.

[117] O. Stauning, "Automatic validation of numerical solutions," 1997.

[118] T. Reps and A. Thakur, "Automating abstract interpretation," in *International Conference on Verification, Model Checking, and Abstract Interpretation*, 2016.

[119] G. Singh, M. Püschel, and M. Vechev, "A practical construction for decomposing numerical abstract domains," *Proceedings of the ACM on Programming Languages*, no. POPL, 2018.

[120] P. Cousot, R. Giacobazzi, and F. Ranzato, "A^{2}i: abstract^2 interpretation," *Proceedings of the ACM on Programming Languages*, no. POPL, 2019.

[121] G. Singh, R. Ganvir, M. Püschel, and M. Vechev, "Beyond the single neuron convex barrier for neural network certification," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[122] T. Du, S. Ji, L. Shen, Y. Zhang, J. Li, J. Shi, C. Fang, J. Yin, R. Beyah, and T. Wang, "Cert-rnn: Towards certifying the robustness of recurrent neural networks." in *CCS*, 2021.

[123] H. Kaper and H. Engler, *Mathematics and climate*. SIAM, 2013.

[124] J. Walsh, "Climate modeling in differential equations," *The UMAP Journal*, vol. 36, no. 4, pp. 325–363, 2015.

[125] Y. Wang, Q. Gao, and M. Pajic, "Learning monotone dynamics by neural networks," in *2022 American Control Conference (ACC)*. IEEE, 2022, pp. 1485–1490.

[126] A. Adjé, S. Gaubert, and E. Goubault, "Coupling policy iteration with semi-definite relaxation to compute accurate numerical invariants in static analysis," in *European Symposium on Programming*, 2010.

[127] J. P. Turner, "Analysing and bounding numerical error in spiking neural network simulations," Ph.D. dissertation, University of Sussex, 2020.

[128] G. Singh, T. Gehr, M. Püschel, and M. Vechev, "Boosting robustness certification of neural networks," in *International conference on learning representations*, 2019.

[129] C. Laneve, T. A. Lascu, and V. Sordoni, "The interval analysis of multilinear expressions," *Electronic Notes in Theoretical Computer Science*, vol. 267, no. 2, pp. 43–53, 2010.

[130] Z. Shi, H. Zhang, K.-W. Chang, M. Huang, and C.-J. Hsieh, "Robustness verification for transformers," in *ICLR*, 2020.

[131] J. Stolfi and L. H. De Figueiredo, "Self-validated numerical methods and applications," in *Monograph for 21st Brazilian Mathematics Colloquium, IMPA, Rio de Janeiro. Citeseer*, vol. 5, no. 1. Citeseer, 1997.

[132] T. Helaire et al, "affapy library," 2021.

[133] A. Karpathy et al., "micrograd library," 2020.

[134] C. R. Harris, K. J. Millman, S. J. Van Der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith et al., "Array programming with numpy," *Nature*, vol. 585, no. 7825, pp. 357–362, 2020.

[135] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[136] A. Saltelli, M. Ratto, S. Tarantola, and F. Campolongo, "Sensitivity analysis for chemical models," *Chemical reviews*, vol. 105, no. 7, pp. 2811–2828, 2005.

[137] J. Kitchin, "A differentiable ode integrator for sensitivity analysis," 2018.

[138] F. Black and M. Scholes, "The pricing of options and corporate liabilities," *Journal of political economy*, vol. 81, no. 3, pp. 637–654, 1973.

[139] B. Becker and R. Kohavi, "Adult," UCI Machine Learning Repository, 1996, DOI: https://doi.org/10.24432/C5XW20.

[140] C.-Y. Ko, Z. Lyu, L. Weng, L. Daniel, N. Wong, and D. Lin, "Popqorn: Quantifying robustness of recurrent neural networks," in *International Conference on Machine Learning*. PMLR, 2019, pp. 3468–3477.

[141] B. Paulsen and C. Wang, "Linsyn: Synthesizing tight linear bounds for arbitrary neural network activation functions," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2022, pp. 357–376.

[142] B. Paulsen and C. Wang, "Example guided synthesis of linear approximations for neural network verification," in *International Conference on Computer Aided Verification*, 2022.

[143] N. Kochdumper, C. Schilling, M. Althoff, and S. Bak, "Open-and closed-loop neural network verification using polynomial zonotopes," *arXiv preprint arXiv:2207.02715*, 2022.

[144] C. Yang, Y. Deng, J. Yao, Y. Tu, H. Li, and L. Zhang, "Fuzzing automatic differentiation in deep-learning libraries," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1174–1186.

[145] X. Zhang, J. Zhai, S. Ma, and C. Shen, "Autotrainer: An automatic dnn training problem detection and repair system," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 359–371.