


# A General Construction for Abstract Interpretation of Higher-Order Automatic Differentiation

JACOB LAUREL, University of Illinois Urbana-Champaign, USA
REM YANG, University of Illinois Urbana-Champaign, USA
SHUBHAM UGARE, University of Illinois Urbana-Champaign, USA
ROBERT NAGEL, University of Illinois Urbana-Champaign, USA
GAGANDEEP SINGH, University of Illinois Urbana-Champaign and VMware Research, USA
SASA MISAILOVIC, University of Illinois Urbana-Champaign, USA

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.

CCS Concepts: • **Theory of computation → Program analysis**; **Abstraction**; • **Mathematics of computing → Automatic differentiation**.

Additional Key Words and Phrases: Abstract Interpretation, Differentiable Programming

**ACM Reference Format:**
Jacob Laurel, Rem Yang, Shubham Ugare, Robert Nagel, Gagandeep Singh, and Sasa Misailovic. 2022. A General Construction for Abstract Interpretation of Higher-Order Automatic Differentiation. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 161 (October 2022), 29 pages. https://doi.org/10.1145/3563324

## 1 INTRODUCTION

Recent years have seen a resurgence in popularity of Automatic Differentiation (AD) and Differentiable Programming – not only from machine learning applications requiring derivatives (e.g., for training neural networks [Abadi et al. 2016]), but also in areas as diverse as Computer

Authors' addresses: Jacob Laurel, jlaurel2@illinois.edu, University of Illinois Urbana-Champaign, USA; Rem Yang, remyang2@illinois.edu, University of Illinois Urbana-Champaign, USA; Shubham Ugare, sugare2@illinois.edu, University of Illinois Urbana-Champaign, USA; Robert Nagel, rjnagel2@illinois.edu, University of Illinois Urbana-Champaign, USA; Gagandeep Singh, ggnds@illinois.edu, University of Illinois Urbana-Champaign and VMware Research, USA; Sasa Misailovic, misailo@illinois.edu, University of Illinois Urbana-Champaign, USA.


2475-1421/2022/10-ART161
https://doi.org/10.1145/3563324

Graphics [Li et al. 2018], Physical Simulation [Hu et al. 2020] and Scientific Computing [Walther and Griewank 2012]. However, despite the widespread adoption of AD, as pointed out in Hückelheim et al. [2018], there is surprisingly little work on *formally verifying* that these programs compute correctly. While there have been several works [Di Gianantonio and Edalat 2013; Krawiec et al. 2022; Sherman et al. 2021] on proving correct the *concrete* semantics of AD, there is almost no work on defining different *abstract* AD semantics for the purpose of verifying program properties expressed over derivatives.

Abstractly interpreting AD programs is further complicated by the fact that these programs do not have the typical concrete semantics of standard numerical programs. One either overloads all operators to have a non-standard interpretation over a *tuple* of numbers corresponding to the derivatives, such as with dual numbers [Griewank and Walther 2008], or one defines the semantics of the differentiable program using reverse-mode program execution. Additionally, most semantics for computing higher-order derivatives are defined over complicated structures, such as derivative towers [Karczmarczuk 2001] or lazily evaluated data structures as in Pearlmutter and Siskind [2007]. Hence, given the complexities in existing AD program semantics, adapting existing static analysis techniques, specifically abstract interpretation [Cousot and Cousot 1977], to soundly reason about differentiable programs is a notable challenge.

While there has been limited work for soundly reasoning about AD, the few techniques that have been developed in this area are significantly restricted: they are either limited to the interval domain [Bendtsen and Stauning 1996; Deussen 2021; Laurel et al. 2022a; Sherman et al. 2021] or do not support higher-order derivatives [Jordan and Dimakis 2021; Laurel et al. 2022a]. Thus, precisely tracking relational information *across* higher derivatives is out of the realm of existing abstract interpreters, despite the fact that this precision is needed in tasks such as improving the performance of verified ODE solvers [Immler 2018]. Furthermore, 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 differences in the precision of different domains, have not been studied – especially for AD involving higher-order derivatives.

**Challenges.** We focus 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 Bendtsen and Stauning [1996]; Deussen [2021]; Laurel et al. [2022a]. 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.** We present 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 single 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 transformers 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 paper 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 [Laurel et al. 2022c].
(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 Laurel et al. [2022a].

## 2 EXAMPLE

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

**Running Example.** Fig. 1 presents an example neural network (the reader can find a simpler example in the Appendix [Laurel et al. 2022b]). 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 Janizek et al. [2021]) or if we want to certify input regions where the function is locally concave or convex [Deussen 2021]. For this example, we study the impact of the interaction between $x_1$ and $x_2$ on the

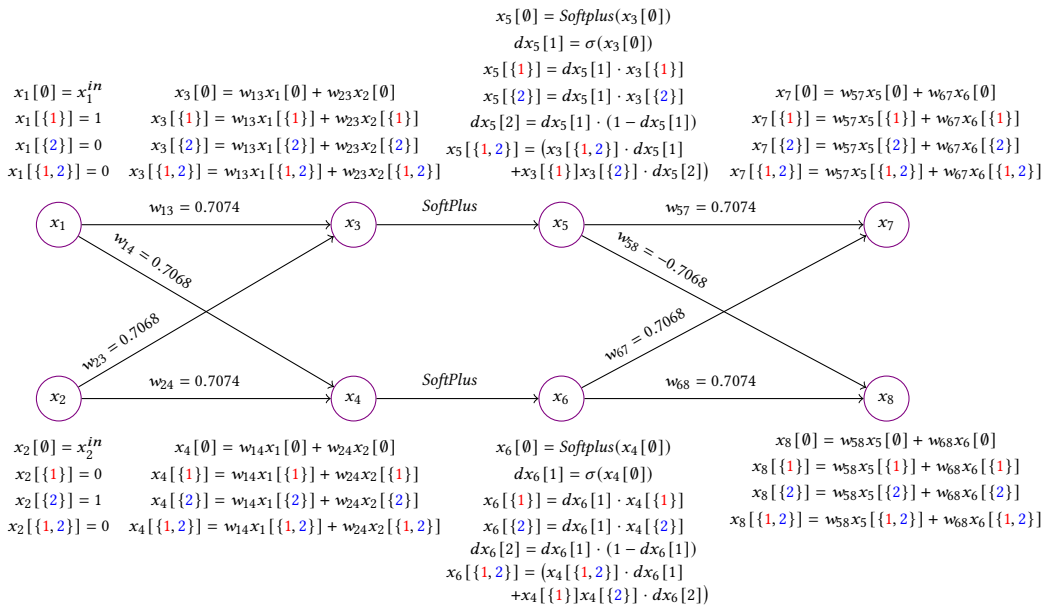



Fig. 1. Concrete $2^{nd}$ order forward-mode AD evaluation

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 4).

**Second Derivative Analysis.** While our prior work [Laurel et al. 2022a] 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 4.11. 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. 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 [Di Bruno 1857], and the product rule must be generalized using the general Leibniz rule.

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 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 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

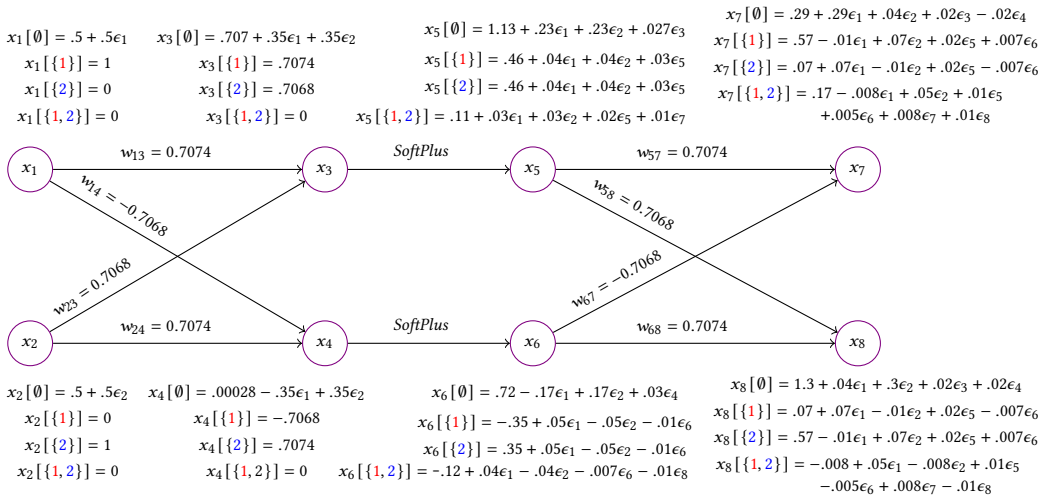



Fig. 2. Abstract $2^{nd}$ order forward-mode AD evaluation

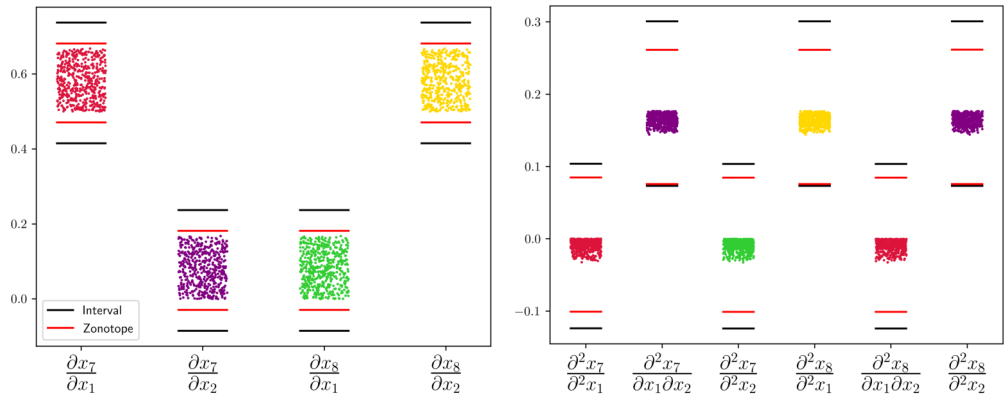
$x_7[\{1,2\}] = \frac{\partial^2 x_7}{\partial x_1 \partial x_2}\Big|_{(x_1^{in}, x_2^{in}) \in \mathbb{R}^2}$, where the notation $\frac{\partial x_i}{\partial x_j, ..., \partial x_k}\Big|_{(x_1, ..., x_m) \in \mathbb{R}^m}$ denotes the partial derivative of some variable $(x_i)$ with respect to other 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 [Alvarez-Melis and Jaakkola 2018], hence why recent work [Fel et al. 2022] 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 [Ghorbal et al. 2009], 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 preserved to a high-degree; 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 ($\epsilon_2$ instead of $\epsilon_1$) 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




(a) Bounds on First Derivatives (b) Bounds on Second Derivatives

Fig. 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.

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 $SoftPlus(x)$, 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 $SoftPlus(x)$, $\sigma(x)$, and multiplication. In Section 6.2, we describe how to automatically construct zonotope transformers for functions like $SoftPlus(x)$ that other works [Jordan and Dimakis 2021; Singh et al. 2018a] cannot support, using the Chebyshev construction [Stolfi and de Figueiredo 2003]. The core idea is to add new noise symbols for each application of a non-linear function. The resulting affine forms after applying the $SoftPlus(x)$ are shown above variable $x_5$ and below variable $x_6$ in Fig. 2. Since the $dx_i[1]$ and $dx_i[2]$ are merely intermediate variables for the computation 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. 2. Furthermore, we can ultimately convert this output zonotope to interval bounds for each variable. Fig. 3a presents in red the bounds for all first derivatives computed from the zonotopes and likewise Fig. 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. 3b, as second derivatives require more computations, compounding the over-approximation inherent to the abstraction. For illustration simplicity, the

example of Fig. 2 corresponds to a single pass of (abstract) forward-mode AD. However, to obtain *all* entries plotted in Figs. 3a and 3b, we need to rerun forward-mode AD for multiple passes. Theorem 4.11 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 7.2.

## 3 PRELIMINARIES

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

### 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 $Part(S)$.

### 3.2 Forward-Mode Automatic Differentiation

The most basic implementation of first-order forward-mode AD is operator overloading with dual numbers [Griewank et al. 2000]. 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 $\epsilon$ being an infinitesimal part, similar to the imaginary part of complex numbers. However, because our work later uses $\epsilon$ 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 $f : \mathbb{R} \to \mathbb{R}$, 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 Griewank et al. [2000]. 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 "de-package" 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 Paszke et al. [2021], 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 [Fike and Alonso 2011]. 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 4.4.

*Example 3.3.* 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 $T[\{1, 2\}] = \frac{\partial^2 x_o}{\partial x_1 \partial x_2}$ .

## 4 LANGUAGE SYNTAX AND SEMANTICS

### 4.1 Syntax

Figure 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

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

Fig. 4. Differentiable Function Syntax

original program's syntax (e.g. $\mathsf{x}_1$,...,$\mathsf{x}_k$) 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. 4.1 the syntactic restrictions on programs needed for the resulting derivatives to be correctly computed.

*Definition 4.1.* 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 4.2.* 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 [Flanagan et al. 1993].

### 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 3.2) 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 more easily track correlations *between* individual variables when later using a relational abstract domain.

Hence, for forward-mode AD, our concrete states $\sigma$ 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 $O(m{\cdot}2^D)$ augmented variables, e.g. $x_1[\emptyset], ..., x_1[\{1, ..., D\}], ..., x_m[\emptyset], ..., x_m[\{1, ..., D\}]$. Having now defined a concrete AD program state $\sigma$ (which is what the semantics will ultimately return), we can formally define the concrete domain upon which the meta-semantics will be built.

*Definition 4.3.* 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 $\sigma$ we can just enumerate all $O(m \cdot 2^D)$ augmented variables into a single state vector with their corresponding Taylor coefficient value, thus this state vector $\sigma$ is equivalently an element of $\mathbb{R}^{|AugVars|}$. Because our AD states are now standard mappings

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

Fig. 5. Semantic rules for the base statement and expression interpreter $\llbracket \cdot \rrbracket$ ($\bot$ is the error state).

of (augmented) variables to real numbers, we can construct an interpreter for $\llbracket \cdot \rrbracket_D$ using only a standard interpreter for real-valued imperative programs $\llbracket \cdot \rrbracket : (P, \mathcal{D}) \to \mathcal{D}$. We now formally define $\llbracket \cdot \rrbracket_D$ in terms of the base interpreter $\llbracket \cdot \rrbracket$.

*Definition 4.4.* The base interpreter $\llbracket \cdot \rrbracket : (P, \mathcal{D}) \to \mathcal{D}$ for assignment statements, and $\llbracket Expr \rrbracket : (Expr, \mathcal{D}) \to \mathbb{R}$ for arithmetic sub-expressions, is given by the rules of Fig. 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 $\llbracket \cdot \rrbracket$ returns $\bot$. Likewise, any arithmetic operation with $\bot$ returns $\bot$. We can now use $\llbracket \cdot \rrbracket$ as a *subroutine* to build an interpreter for performing concrete $D^{th}$ degree forward-mode AD. Lastly, while $\llbracket \cdot \rrbracket$ is defined in Fig. 5 only for elementary expressions (binary arithmetic and unary functions) we will still notationally write $\llbracket Expr \rrbracket$ 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 4.5.* The concrete meta-semantics for performing forward-mode AD are a parametric semantics given by $\llbracket \cdot \rrbracket_D : (P, \mathcal{D}) \to \mathcal{D}$, where the parameter $D \in \mathbb{N}_{>0}$ is the order of 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 $\llbracket \cdot \rrbracket_D$. We immediately see why the concrete meta-semantics are parametric in $D$ – choosing a $D$ governs how many times $\llbracket \cdot \rrbracket$ 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. 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 $\sigma$ after each application of $\llbracket \cdot \rrbracket$.

***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 $x_k$.

$$\begin{array}{l} \llbracket x_i = x_j + x_k \rrbracket_D(\sigma) \triangleq \; \textbf{for} \;\; S \in \mathcal{P}(\{1, ..., D\}) : \\ \qquad\qquad\qquad\qquad \sigma = \llbracket x_i[S] = x_j[S] + x_k[S] \rrbracket(\sigma) \\ \qquad\qquad\qquad \textbf{return} \;\; \sigma \end{array}$$

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

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

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

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. 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.

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

**Unary Functions.** For compositions with unary functions $f: \mathbb{R} \to \mathbb{R}$, we will need to use Faa di Bruno's formula [Di Bruno 1857] 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. 4.6:

*Definition 4.6.* ([Di Bruno 1857]) Let $f: \mathbb{R} \to \mathbb{R}$ and $y = g(x_1, ..., x_D)$ where $g: \mathbb{R}^D \to \mathbb{R}$. 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. 4.6 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. 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 $\sigma$, 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.

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

***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.

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

***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.

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

***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.

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

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

Furthermore, because 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.

***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.

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

Remark 1. *When $D = 1$, $[\![\cdot]\!]_D$ computes exactly the dual numbers [Griewank and Walther 2008] and when $D = 2$, $[\![\cdot]\!]_D$ computes exactly the hyper-dual numbers [Fike and Alonso 2011].*

### 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 $\sigma_a$ 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 7.

### 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 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 [Griewank and Walther 2008]. We generalize this idea to arbitrary order derivatives, as described below.

*Definition 4.7.* For $[\![\cdot]\!]_D$, an initial state $\sigma$ 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 4.8.* For $D = 2$ and $P \triangleq x_3 = x_1^{in} + x_2^{in}$, and state $\sigma$ 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 $\sigma$ 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 $\sigma$ 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 4.9. *Let $\sigma$ 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*

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

*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. See Appendix. □

**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 4.10. *Let $\sigma$ 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 $\sigma'$ and any non-empty set $S \in \mathcal{P}(D) \setminus \emptyset$, we have that*

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

Proof. (Sketch) We show select cases with all other cases shown in the Appendix.

• *Constants.* For any non-empty set $S$, and constant $c \in \mathbb{R}$, $\frac{\partial^{|S|} c}{\prod_{j \in Active(S)} \partial x_j^{in}}\Big|_{(x_1^{in}[\emptyset],\ldots,x_m^{in}[\emptyset]) \in \mathbb{R}^m} = 0$. However, for any non-empty $S$, the interpreter always assigns 0 to $\sigma'[x_i[S]]$.

• *Addition.* As the current state $\sigma$ has been correctly initialized, before executing $[\![x_i = x_j + x_k]\!]_D(\sigma)$ we inductively assume (by Lemma 4.9) that for each $S$, $\sigma[x_j[S]] = \frac{\partial^{|S|} x_j}{\prod_{l \in Active(S)} \partial x_l^{in}}\Big|_{(x_1^{in}[\emptyset],\ldots,x_m^{in}[\emptyset]) \in \mathbb{R}^m}$ and $\sigma[x_k[S]] = \frac{\partial^{|S|} x_k}{\prod_{l \in Active(S)} \partial x_l^{in}}\Big|_{(x_1^{in}[\emptyset],\ldots,x_m^{in}[\emptyset]) \in \mathbb{R}^m}$. Furthermore, by linearity of derivatives, we know that

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

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. 5. Thus, by substitution $\sigma'[x_i[S]] = \frac{\partial^{|S|} x_j}{\prod_{l \in Active(S)} \partial x_l^{in}}\Big|_{(x_1^{in}[\emptyset],\ldots,x_m^{in}[\emptyset]) \in \mathbb{R}^m} + \frac{\partial^{|S|} x_k}{\prod_{l \in Active(S)} \partial x_l^{in}}\Big|_{(x_1^{in}[\emptyset],\ldots,x_m^{in}[\emptyset]) \in \mathbb{R}^m}$, hence $\sigma'[x_i[S]] = \frac{\partial^{|S|}(x_j + x_k)}{\prod_{l \in Active(S)} \partial x_l^{in}}\Big|_{(x_1^{in}[\emptyset],\ldots,x_m^{in}[\emptyset]) \in \mathbb{R}^m} = \frac{\partial^{|S|} x_i}{\prod_{l \in Active(S)} \partial x_l^{in}}\Big|_{(x_1^{in}[\emptyset],\ldots,x_m^{in}[\emptyset]) \in \mathbb{R}^m}$. □

**Complexity.** While Theorem 4.10 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 $\sigma$. 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 4.11. *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. □

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 Griewank and Walther [2008], and more recent work [He 2019] has noted that specifically for higher-order derivatives, forward-mode AD is better suited than reverse mode.

## 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 5.1.* [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. 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 5.2.* [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. 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. 5.

### 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. 5.1 and 5.2, 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. 4, we require a sound transformer for assignments with that expression, $[\![x_i = Expr]\!]^\# : \mathcal{D}^\# \to \mathcal{D}^\#$.
(5) (Optionally) we need an abstract test transformer $[\![x_i > c]\!]^\# : \mathcal{D}^\# \to \mathcal{D}^\#$ 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. *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. 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 Stolfi and de Figueiredo [2003] and Miné [2006] to construct the necessary transformers. Shi et al. [2019] has also shown how to construct DeepPoly abstract transformers for $\cdot$ and $/$.

*Differentiable Unary Functions.* As our language (Fig. 4) includes unary functions, the chosen abstract domain must provide sound transformers for the following functions: $\sigma_a$, *SoftPlus*, *exp*, and *log*. As mentioned in Section 4.5, 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 [Stolfi and de Figueiredo 2003], the polyhedral linearization of Miné [2006], or using the methods of Ryou et al. [2021] 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 $\sigma_a$ 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), hence why this step is optional. By requiring $\gamma([\![x_i > c]\!]^\#(\sigma^\#)) \subseteq \gamma(\sigma^\#)$, this step never loses precision.

Remark 4. *We do not need a widening operator, since all the programs expressible in the syntax of Fig. 4 are loop-free. We also do not require a join operator $\sqcup$ or meet operator $\sqcap$.*

### 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 (Definition 4.5). 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 5.3.* 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. 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 4.5 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 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 5.1, $[\![\cdot]\!]^{\#}$ has sound transformers for (abstract) addition of two variables.

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

***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.

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

***Constants.*** For this case, we need to assign each augmented variable to a constant (either $c$ or 0), hence all this requires is that the abstract domain has sound transformers constant assignment.

*Unary Functions.* As in the concrete meta-semantics, with unary functions, we must use Faa di Bruno's formula (Def. 4.6). 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 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.

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

***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*.

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

***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.

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

***Abstract Sequencing.*** Abstractly interpreting the sequencing of multiple statements follows similarly to the concrete meta-semantics.

$$
[\![P_1; P_2]\!]^{\#}_D(\sigma^{\#}) \triangleq [\![P_2]\!]^{\#}_D\big([\![P_1]\!]^{\#}_D(\sigma^{\#})\big)
$$

Remark 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 relationally tracks dependencies across derivatives and derivative orders.*

### 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 5.4. *(Soundness of Abstraction) For any program $P$ expressible in the syntax of Fig. 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 5.1 are satisfied.

Proof. (Sketch) We provide sketches of representative cases (others detailed in the Appendix):

• *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 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 $[\![\cdot]\!]$ (also by assumption) for every application (regardless of $D$) we know $[\![x_i[S] = x_j[S] + x_k[S]]\!](\sigma) \in \gamma([\![x_i[S] = x_j[S] + x_k[S]]\!]^\#(\sigma^\#))$.

• *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. 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. 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:* It represents an inductive case. 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 $[\![P_1; P_2]\!]_D(\sigma) \in \gamma([\![P_1; P_2]\!]^\#_D(\sigma^\#))$. □

## 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.

### 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 $\mathcal{D}^\# = \mathcal{P}(\mathbb{R}^{|AugVars|})$, is among the simplest numeric abstract domains where in an abstract state $\sigma^\#$ a variable is mapped to an interval $[a, b] \in \mathbb{R}$, hence $\sigma^\# : AugVar \to \mathbb{R}$. The interval domain is *non-relational*.

**First Derivatives.** Our construction can be used to produce the Dual Interval domain of Laurel et al. [2022a]. 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 Fike and Alonso [2011] 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.

### 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, which we denote with $\mathcal{D}^\# = \textbf{Zono}$. The abstract state $\sigma^\# \in \textbf{Zono}$ 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 $\epsilon_i$ are shared across variables, this is a relational domain; furthermore, the set of states encoded by $\sigma^\#$ is exactly a zonotope. This can be seen from the concretization function $\gamma\colon \textbf{Zono} \to \mathcal{P}(\mathbb{R}^{|AugVars|})$ defined as:

$$\gamma(\sigma^\#) = \{(x_0, ..., x_{|AugVars|}) \in \mathbb{R}^{|AugVars|} \,\big|\, \forall j, i :\; x_j = \sigma^\#[x_j][c] + \sum_{i=1}^{|generators|} \sigma^\#[x_j][g_i]\cdot\epsilon_i \wedge \epsilon_i \in [-1, 1]\}.$$

As mentioned in Albarghouthi [2021], $|generators|$ 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 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 [Jordan and Dimakis 2021; Singh et al. 2018a] 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 [Fryazinov et al. 2010; Stolfi and de Figueiredo 2003]:

*Definition 6.1.* 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:

$$y = (\alpha \cdot c + \zeta) + \left(\sum_{i=1}^{|generators|} (\alpha \cdot g_i)\epsilon_i\right) + \delta\epsilon_{new},$$

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 $\delta = \frac{|f(u) - r(u)|}{2}$. 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 Singh et al. [2018a]) or just default to the interval domain transformers. We can then collectively use all of these abstract transformers in the general construction of $[\![\cdot]\!]^{\#}_D$.

Theorem 6.2. *When $D = 2$ and when the abstract domain is the zonotope domain, $[\![\cdot]\!]_D$ computes a zonotope abstraction of the hyper-dual numbers of Fike and Alonso [2011].*

**Precision.** For many functions, the zonotope transformers are strictly more precise than the interval ones. As mentioned in [Fryazinov et al. 2010; Stolfi and de Figueiredo 2003], the Chebyshev construction is *optimal* in the input-output plane, hence it gives the same guarantees as Singh et al. [2018a]. The zonotope transformers in this case are incomparable to the interval ones. We also found that existing abstract test transformers for zonotopes [Jeannet and Miné 2009; Singh et al. 2017] 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.

## 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.

### 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 [LeCun 1998] image classification dataset (which contains 70,000 $28 \times 28$ 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 Singh et al. [2019] 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 Paterson et al. [2021]. 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.

### 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 [Ancona et al. 2018; Janizek et al. 2021]. However, prior work has shown that it is not enough to generate explanations for scalar points; rather, explanations should be robust, hence why Fel et al. [2022] used robust interval explanations. Our case study serves to show how our construction allows a user to compute 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

Table 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 |

Table 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 |

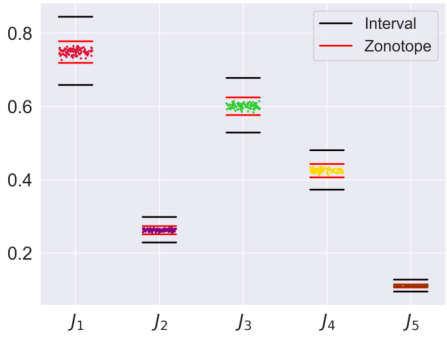



Fig. 6. Bounds on Jacobian entries via the interval and zonotope abstract domains.

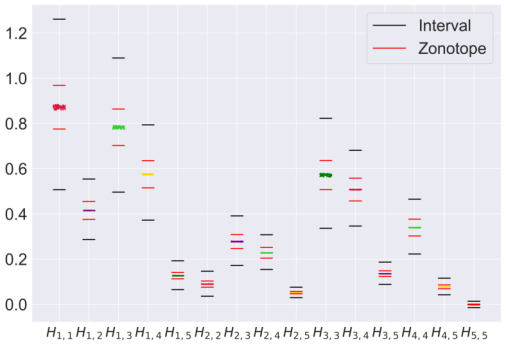


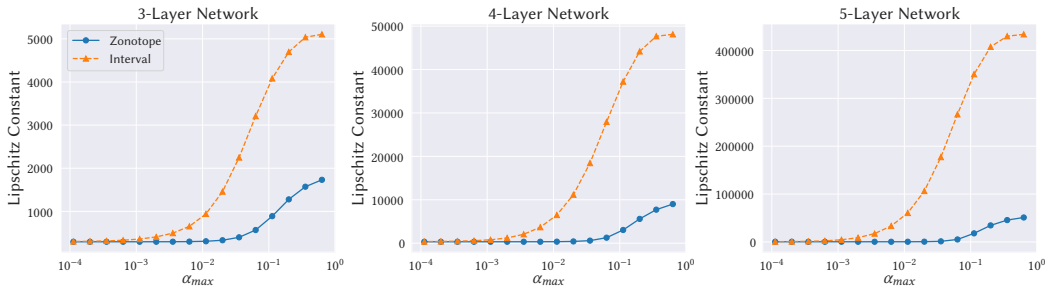

Fig. 7. Bounds on Hessian entries via the interval and zonotope abstract domains.

range $[0, 1)$ and enclose each input in an interval of $\pm 0.01$. We then soundly bound the first and second derivatives of the neural network (trained to approximate a polynomial as described in Section 7.1) with respect to these input ranges. We repeat this experiment for 5 different seeds.

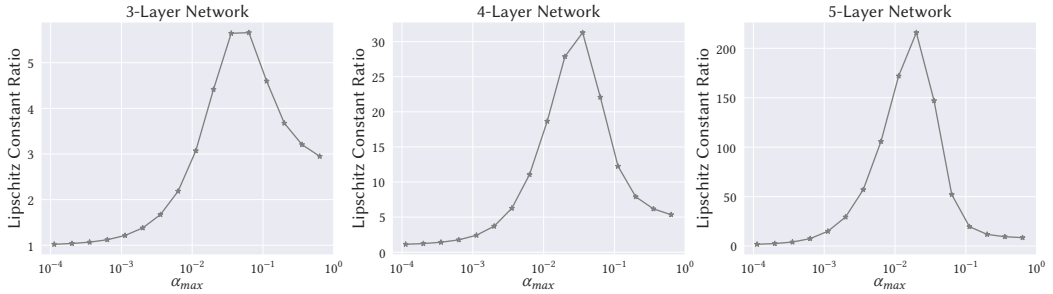
Table 1 and Table 2 demonstrate the precision improvement of the zonotope domain on bounding the Jacobian and the Hessian, respectively. In both tables, each cell presents the geometric mean over 5 trials of the ratio of the interval-bounded derivative's width over its zonotope-bounded counterpart. For 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. 6 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. 7 presents the results (for the same trial as in Fig. 6). The entries along the x-axis represent Hessian terms instead of just individual input derivatives (as in Fig. 6). 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




(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.

Fig. 8. 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. 8a) presents the average upper bound on the local Lipschitz constant for both the zonotope and interval domains. The bottom row (Fig. 8b) 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.

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.

### 7.3 Lipschitz Certification

Our second case study involves bounding the local Lipschitz constant of neural networks with respect to semantic perturbations, as in [Laurel et al. 2022a; Yang et al. 2022]. 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 $\alpha$ 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 $\alpha$ over *a range of* haze values, specifically where $\alpha$ lies in $[0, \alpha_{max}]$. The parameter $\alpha_{max}$ 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 Laurel et al. [2022a] (as no other work can handle this setting).

Fig. 8 shows the Lipschitz certification results for the 3-layer (left), 4-layer (center), and 5-layer (right) MNIST networks. Fig. 8a presents the computed Lipschitz constant bounds. The x-axis shows the value of $\alpha_{max}$, and the y-axis shows the upper bound on the Lipschitz constant computed with the zonotope and interval domains (smaller is better). Fig. 8b presents the increase in precision of the zonotope domain. The x-axis again shows the value of $\alpha_{max}$, 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 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 $\alpha_{max}$. Fig. 8a shows that as $\alpha_{max}$ (i.e., the range of perturbation) increases, the *absolute* difference between the interval- 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. 8b shows that for small values of $\alpha_{max}$, both domains are quite precise, since the input range is very small. For large values of $\alpha_{max}$, 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. 8b exactly correspond to the points in Fig. 8a 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 Laurel et al. [2022a].

Our approach can also scale to larger networks: Fig. 9 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 constants are up to $4.67 \times 10^8$ 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.

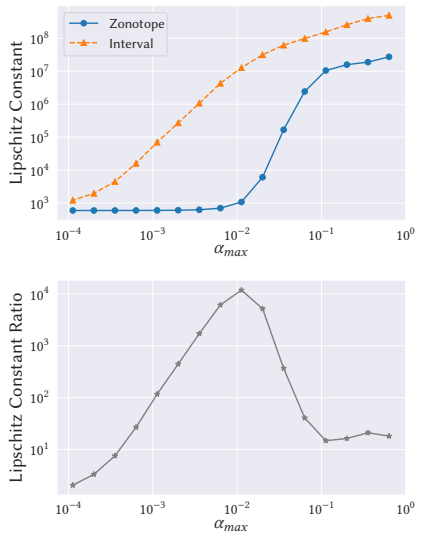



Fig. 9. Top: Average local Lipschitz constant upper bounds (lower is better) for the zonotope and interval domains on FFNNBig. Bottom: the increase in precision of the zonotope over the interval domain.

## 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 Griewank et al. [2000] and Karczmarczuk [2001]. Pearlmutter and Siskind [2007] also proposed a forward-mode scheme; then, later implementations in JAX [Bettencourt et al. 2019], $\lambda_S$ [Sherman et al. 2021], and the work of Huot et al. [2021] also followed in this spirit. In particular, $\lambda_S$ also supports Clarke Jacobians. However, all

of these works only give *concrete* (and not abstract) semantics for AD. Additionally, unlike our concrete semantics, Huot et al. [2021]; Karczmarczuk [2001]; Krawiec et al. [2022]; Pearlmutter and Siskind [2007]; Sherman et al. [2021] all define their concrete semantics in functional styles, whereas we use an imperative semantics (that generalizes Fike and Alonso [2011]) 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 Griewank and Walther [2008], as our tuple coefficients are unscaled by factorial terms.

**Abstract Interpretation of AD.** As pointed out in Laurel et al. [2022a] and Vassiliadis et al. [2016], there is very little work on Abstract Interpretation for AD. Most recently, Laurel et al. [2022a] developed an interval abstract interpretation for bounding Clarke Jacobians; however, they cannot support relational domains or higher derivatives. Jordan and Dimakis [2021] 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 Vassiliadis et al. [2016] and Mangal et al. [2020] use the interval domain to bound AD, but their works cannot use any relational domain (e.g., zonotopes or polyhedra). Deussen [2021] 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. Immler [2018] 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, Bendtsen and Stauning [1996] and Stauning [1997] 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 [Cousot et al. 2019; Reps and Thakur 2016; Singh et al. 2018b; Sotoudeh and Thakur 2020], 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 [Miné 2006], but the precision and scalability of these transformers are usually suboptimal.

## 9 CONCLUSION

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.

## ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their comments. This research was supported in part by NSF Grants No. CCF-1846354, CCF-1956374, CCF-2008883, CNS-2148583, USDA NIFA Grant No. AG NIFA 2021-67021-33449, a gift from Facebook, and a Sloan UCEM Graduate Scholarship.

## REFERENCES


Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX symposium on operating systems design and implementation*.

Aws Albarghouthi. 2021. Introduction to Neural Network Verification. *Foundations and Trends® in Programming Languages* (2021).

David Alvarez-Melis and Tommi S Jaakkola. 2018. Towards robust interpretability with self-explaining neural networks. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*.

Marco Ancona, Enea Ceolini, Cengiz Öztireli, and Markus Gross. 2018. Towards better understanding of gradient-based attribution methods for Deep Neural Networks. In *6th International Conference on Learning Representations (ICLR)*.

Claus Bendtsen and Ole Stauning. 1996. FADBAD, a flexible C++ package for automatic differentiation. (1996).

Jesse Bettencourt, Matthew J Johnson, and David Duvenaud. 2019. Taylor-mode automatic differentiation for higher-order derivatives in JAX. (2019).

Patrick Cousot and Radhia Cousot. 1977. 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*.

Patrick Cousot, Roberto Giacobazzi, and Francesco Ranzato. 2019. $A^2I$: abstract$^2$ interpretation. *Proceedings of the ACM on Programming Languages* POPL (2019).

Jens Deussen. 2021. *Global Derivatives*. Ph. D. Dissertation.

Fr. Faa Di Bruno. 1857. Note sur une nouvelle formule de calcul différentiel. *Quarterly J. Pure Appl. Math* (1857).

Pietro Di Gianantonio and Abbas Edalat. 2013. A language for differentiable functions. In *International Conference on Foundations of Software Science and Computational Structures*.

Thomas Fel, Mélanie Ducoffe, David Vigouroux, Rémi Cadène, Mikael Capelle, Claire Nicodème, and Thomas Serre. 2022. Don't Lie to Me! Robust and Efficient Explainability with Verified Perturbation Analysis. *arXiv preprint arXiv:2202.07728* (2022).

Jeffrey Fike and Juan Alonso. 2011. The Development of Hyper-Dual Numbers for Exact Second-Derivative Calculations. *AIAA* (2011).

Cormac Flanagan, Amr Sabry, Bruce F Duba, and Matthias Felleisen. 1993. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*.

Oleg Fryazinov, Alexander Pasko, and Peter Comninos. 2010. Technical Section: Fast Reliable Interrogation of Procedurally Defined Implicit Surfaces Using Extended Revised Affine Arithmetic. *Comput. Graph.* 34, 6 (2010).

Khalil Ghorbal, Eric Goubault, and Sylvie Putot. 2009. The zonotope abstract domain taylor1+. In *International Conference on Computer Aided Verification*.

Andreas Griewank, Jean Utke, and Andrea Walther. 2000. Evaluating higher derivative tensors by forward propagation of univariate Taylor series. *Mathematics of computation* (2000).

Andreas Griewank and Andrea Walther. 2008. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM.

Horace He. 2019. The State of Machine Learning Frameworks in 2019. https://thegradient.pub/state-of-ml-frameworks-2019-pytorch-dominates-research-tensorflow-dominates-industry/. *The Gradient* (2019).

Yuanming Hu, Luke Anderson, Tzu-Mao Li, Qi Sun, Nathan Carr, Jonathan Ragan-Kelley, and Frédo Durand. 2020. DiffTaichi: Differentiable Programming for Physical Simulation. In *International Conference on Learning Representations*.

Jan Hückelheim, Ziqing Luo, Sri Hari Krishna Narayanan, Stephen Siegel, and Paul D Hovland. 2018. Verifying Properties of Differentiable Programs. In *International Static Analysis Symposium*.

Mathieu Huot, Sam Staton, and Matthijs Vákár. 2021. Higher Order Automatic Differentiation of Higher Order Functions. *arXiv preprint arXiv:2101.06757* (2021).

Fabian Immler. 2018. *A Verified ODE Solver and Smale's 14th Problem*. Ph. D. Dissertation. Technische Universität München.

Joseph D Janizek, Pascal Sturmfels, and Su-In Lee. 2021. Explaining explanations: Axiomatic feature interactions for deep networks. *Journal of Machine Learning Research* 22 (2021).

Bertrand Jeannet and Antoine Miné. 2009. Apron: A library of numerical abstract domains for static analysis. In *International Conference on Computer Aided Verification*. 661–667.

Matt Jordan and Alex Dimakis. 2021. Provable Lipschitz certification for generative models. In *International Conference on Machine Learning*. PMLR, 5118–5126.

Jerzy Karczmarczuk. 2001. Functional differentiation of computer programs. *Higher-order and symbolic computation* (2001).

Faustyna Krawiec, Neel Krishnaswami, Simon Peyton Jones, Tom Ellis, Andrew Fitzgibbon, and R Eisenberg. 2022. Provably correct, asymptotically efficient, higher-order reverse-mode automatic differentiation. *Proceedings of the ACM on Programming Languages* POPL (2022).

Jacob Laurel, Rem Yang, Gagandeep Singh, and Sasa Misailovic. 2022a. A Dual Number Abstraction for Static Analysis of Clarke Jacobians. *Proceedings of the ACM on Programming Languages* POPL (2022).

Jacob Laurel, Rem Yang, Shubham Ugare, Robert Nagel, Gagandeep Singh, and Sasa Misailovic. 2022b. Appendix to A General Construction for Abstract Interpretation of Higher-Order Automatic Differentiation. https://jsl1994.github.io/papers/OOPSLA2022_appendix.pdf

Jacob Laurel, Rem Yang, Shubham Ugare, Robert Nagel, Gagandeep Singh, and Sasa Misailovic. 2022c. Artifact for A General Construction for Abstract Interpretation of Higher-Order Automatic Differentiation. (2022). https://doi.org/10.1145/3554329

Yann LeCun. 1998. The MNIST database of handwritten digits. *http://yann. lecun. com/exdb/mnist/* (1998).

Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. 2018. Differentiable programming for image processing and deep learning in Halide. *ACM Transactions on Graphics (TOG)* 37, 4 (2018).

Ravi Mangal, Kartik Sarangmath, Aditya V Nori, and Alessandro Orso. 2020. Probabilistic Lipschitz Analysis of Neural Networks. In *International Static Analysis Symposium*.

Antoine Miné. 2006. Symbolic methods to enhance the precision of numerical abstract domains. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*.

Adam Paszke, Daniel D Johnson, David Duvenaud, Dimitrios Vytiniotis, Alexey Radul, Matthew J Johnson, Jonathan Ragan-Kelley, and Dougal Maclaurin. 2021. Getting to the point: index sets and parallelism-preserving autodiff for pointful array programming. *Proceedings of the ACM on Programming Languages* ICFP (2021).

Colin Paterson, Haoze Wu, John Grese, Radu Calinescu, Corina S Păsăreanu, and Clark Barrett. 2021. Deepcert: Verification of contextually relevant robustness for neural network image classifiers. In *International Conference on Computer Safety, Reliability, and Security*.

Barak A Pearlmutter and Jeffrey Mark Siskind. 2007. Lazy Multivariate Higher-Order Forward-Mode AD. In *Symposium on Principles of Programming Languages*.

Thomas Reps and Aditya Thakur. 2016. Automating abstract interpretation. In *International Conference on Verification, Model Checking, and Abstract Interpretation*.

Wonryong Ryou, Jiayu Chen, Mislav Balunovic, Gagandeep Singh, Andrei Dan, and Martin Vechev. 2021. Scalable polyhedral verification of recurrent neural networks. In *International Conference on Computer Aided Verification*.

Benjamin Sherman, Jesse Michel, and Michael Carbin. 2021. $\lambda_S$: Computable Semantics for Differentiable Programming with Higher-Order Functions and Datatypes. *Proceedings of the ACM on Programming Languages* POPL (2021).

Zhouxing Shi, Huan Zhang, Kai-Wei Chang, Minlie Huang, and Cho-Jui Hsieh. 2019. Robustness Verification for Transformers. In *International Conference on Learning Representations*.

Gagandeep Singh, Timon Gehr, Matthew Mirman, Markus Püschel, and Martin T Vechev. 2018a. Fast and Effective Robustness Certification. *NeurIPS* (2018).

Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. 2019. An abstract domain for certifying neural networks. *Proceedings of the ACM on Programming Languages* POPL (2019).

Gagandeep Singh, Markus Püschel, and Martin Vechev. 2017. Fast polyhedra abstract domain. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*.

Gagandeep Singh, Markus Püschel, and Martin Vechev. 2018b. A practical construction for decomposing numerical abstract domains. *Proceedings of the ACM on Programming Languages* POPL (2018).

Matthew Sotoudeh and Aditya V Thakur. 2020. Abstract Neural Networks. In *International Static Analysis Symposium*.

Ole Stauning. 1997. Automatic validation of numerical solutions.

Jorge Stolfi and Luiz Henrique de Figueiredo. 2003. An introduction to affine arithmetic. *Trends in Computational and Applied Mathematics* 4 (2003).

Vassilis Vassiliadis, Jan Riehme, Jens Deussen, Konstantinos Parasyris, Christos D Antonopoulos, Nikolaos Bellas, Spyros Lalis, and Uwe Naumann. 2016. Towards automatic significance analysis for approximate computing. In *2016 IEEE/ACM International Symposium on Code Generation and Optimization*.

Andrea Walther and Andreas Griewank. 2012. Getting Started with ADOL-C. *Combinatorial Scientific Computing* (2012).

Rem Yang, Jacob Laurel, Sasa Misailovic, and Gagandeep Singh. 2022. Provable Defense Against Geometric Transformations. *arXiv preprint arXiv:2207.11177* (2022).

Received 2022-04-15; accepted 2022-09-01