REASONING ABOUT PROGRAMS IN STATISTICALLY MODELED FIRST-ORDER ENVIRONMENTS

A Dissertation
Presented to
The Academic Faculty

By

Ravi Mangal

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science
College of Computing

Georgia Institute of Technology

December 2020

REASONING ABOUT PROGRAMS IN STATISTICALLY MODELED FIRST-ORDER ENVIRONMENTS

Thesis committee:

Dr. Alessandro Orso
School of Computer Science
Georgia Institute of Technology

Dr. William Harris
Principal Scientist
Galois, Inc.

Dr. Vivek Sarkar
School of Computer Science
Georgia Institute of Technology

Dr. Aditya V. Nori
Partner Research Manager
Microsoft Research

Dr. Qirun Zhang
School of Computer Science
Georgia Institute of Technology

Date approved: October 14, 2020

ACKNOWLEDGMENTS

I am extremely privileged to have had the opportunity to pursue a PhD in Computer Science and to have received the support to write this dissertation.

For this, first and foremost, I thank my PhD advisor, Alex Orso. In September 2016, at the start of my 5th year in the PhD program, I found myself in an increasingly toxic work environment and was faced with the difficult choice of continuing to subject myself to the toxicity at the cost of my integrity and mental health, quitting the PhD program, or seeking guidance and mentorship elsewhere. Alex stepped in to keep my PhD dream alive - agreeing to be my PhD advisor, while supporting me whole-heartedly and shielding me from the unpleasantness of my prior work environment (for this, thank you also to Prof. H. Venkateswaran). Ever since, Alex has given me the freedom and the space to learn, grow, and pursue an independent research agenda. For all this and more, I will be ever grateful to Alex. This dissertation owes its existence to him.

Thank you to Aditya Nori. Aditya's enthusiasm for computing is infectious and he has an impeccable eye for problems. He has been an incredible mentor, collaborator, and well-wisher throughout my PhD. He introduced me to the notion of using statistical techniques to aid program verification, and this marriage of program verification and statistics has been a constant theme in my research. The work on verifying neural networks was born during discussions with Aditya and if not for his encouragement, I may have never pursued it.

Thank you to Bill Harris for agreeing to serve on my PhD committee, for his pointed questions and suggestions that helped to improve this dissertation, and most importantly, for demonstrating idealism in practice. Thank you also to Vivek Sarkar and Qirun Zhang for serving on my PhD committee and providing thoughtful feedback.

Thank you to Kartik Sarangmath for trusting me enough to work with me on the neural network verification problem and for patiently hearing me ramble during our meetings.

Thank you to David Devecsery for discussions that led to the work on observational

abstract interpreters.

Thank you to Domagoj Babic and Jayanthkumar Kannan for hosting me at Google in the summer of 2014. Domagoj and Jayanth taught me to think about static analyses at an industrial scale and patiently mentored me in writing production-quality code.

Thank you to Patrice Godefroid for hosting me at Microsoft Research (MSR) in the summer of 2016. Experiencing the wonderful intellectual environment at MSR and the beauty of summer in the Pacific Northwest replenished my soul. Moreover, Patrice's mentoring style is to trust you as a peer while providing all the technical and administrative support that you need to succeed. This deceptively simple yet rare approach to mentoring helped me find self-confidence when I was lacking it the most. For all the lessons and experiences from the summer of 2016, I will be ever thankful to Patrice.

Thank you to Zena Ariola, Jim Allen, and all the organizers of Oregon Programming Languages Summer School (OPLSS). I attended OPLSS in the summer of 2017 with limited prior exposure to PL Theory, Logic, and their connections. The amazing lectures at OPLSS opened my eyes to the depth of the questions being pursued by the field. The two and half weeks spent in Eugene has been one of the most profound experiences of my life, and has irreversibly altered its course. OPLSS is a precious gift to the community and I am grateful for having experienced it.

Thank you to my friends in the CS PhD program: Caleb Voss, Girish Mururu, Prithayan Barua, Qianqian Wang, Rick Rutledge, Sulekha Kulkarni, and Swamit Tannu. Their encouragement and support helped me get through the uncertainties of graduate school. Their willingness to engage in discussions, technical and non-technical, created a stimulating environment, and helped me grow as a computer scientist and as a person.

Thank you to all the members of the Arktos and PLSE research groups for being ever-willing to provide feedback on research papers and talks.

Thank you to my friends in Atlanta: Jake Cobb, Qianqian, Rohan Mazarello, and Rick. They were my family away from family, providing me a home away from home, and I could

not have gotten through my PhD without weekends at Jake and Qianqian's, or without Friday evenings spent with Rohan.

Thank you to my parents, Shakuntala Mangal and Arvind Mangal. Everything I am is because of them. Thank you to my sister, Kavita Mangal, for always having my back, and for always speaking the truth.

Finally, thank you to my wife Laura King and my dog-friend-child Georgie for sharing their lives with me and for making mine infinitely richer in the process, for always supporting me, believing in me and helping me to be the best I can, and for making every day meaningful and worthwhile. None of this would be possible without them.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The objects of study in this dissertation are programs and algorithms that reason about programs using their syntactic structure. Such algorithms, referred to as *program verification algorithms* in the literature, are designed to find proofs of propositions about program behavior.

This dissertation adopts the perspective that programs operate in environments that can be modeled statistically. In other words, program inputs are samples drawn from a generative statistical model. This statistical perspective has two main advantages. First, it allows us to reason about programs that are not expected to exhibit the desired behavior on all program inputs, such as neural networks that are learnt from data, by formulating and proving probabilistic propositions about program behavior. Second, it enables us to simplify the search for proofs of non-probabilistic propositions about program behavior by designing program verification algorithms that are capable of inferring "likely" hypotheses about the program environment.

The first contribution of this dissertation is a pair of program verification algorithms for finding proofs of *probabilistic robustness* of neural networks. A trained neural network f is probabilistically robust if, for a pair of inputs that is randomly generated as per the environment statistical model, f is likely to demonstrate k-Lipschitzness, i.e., the distance between the outputs computed by f is upper-bounded by the k^{th} multiple of the distance between the pair of inputs. A proof of probabilistic robustness guarantees that the neural network is unlikely to exhibit divergent behaviors on similar inputs.

The second contribution of this dissertation is a generic algorithmic framework, referred to as *observational abstract interpreters*, for designing algorithms that compute hypothetical semantic program invariants. Semantic invariants are logical predicates about program behavior and are used in program proofs as lemmas. The well-studied algorithmic framework of abstract interpretation provides a standard recipe for constructing algorithms

that compute semantic program invariants. Observational abstract interpreters extend this framework to allow for computing hypothetical invariants that are valid only under specific hypotheses about program environments. These hypotheses are inferred from observations of program behavior and are embedded as dynamic/run-time checks in the program to ensure the validity of program proofs that use hypothetical invariants.

CHAPTER 1
INTRODUCTION

1.1 Overview

Computer software is an essential component of modern infrastructure. With the increasing complexity of software, there is a pressing need to develop techniques for understanding the behavior of deployed programs to ensure their reliability and correctness. An empirical approach for understanding and reasoning about a program is to run the program and draw conclusions based on the observed program behaviors. Though this approach, referred to as *program testing*, is widely used and successful, it is typically computationally infeasible to observe all program behaviors using this approach. A different approach is to construct a mathematical model of a program and algorithmically prove theorems about the mathematical model in order to establish correctness of the program. Construction of such mathematical models relies on the mathematical definition of the *semantics* of the programming language used to express the program. This approach for reasoning about programs, referred to as *program verification*, is the subject of our study.

In particular, we focus on the verification of *open* programs, as opposed to *closed* programs. An open program interacts with its environment and the program behavior depends on this interaction. Formally, an open program consists of free variables and the environment (or context) provides the values of these variables.[1] Reasoning about an open program, therefore, requires reasoning about program behavior with respect to arbitrary environments. In order to model the interaction with the environment, the standard practice in program testing and verification literature is to formally specify the structure of the values or data generated by the environment via specification of the datatypes of the free

[1] We do not consider reactive programs that repeatedly interact with the environment in this dissertation.

variables.

In this dissertation, we investigate the design of program verification algorithms under the assumption that the environment in which an open program operates can be modeled statistically. In other words, we assume that the values assigned by the environment to the free variables of an open program are generated in accordance with a known or unknown statistical model. We note that the statistical modeling of the environment in this dissertation is restricted to free variables with a first-order datatype. Our statistical perspective on program environments is driven by the observation that inputs to programs are generated by natural and social processes, and the use of statistical models is ubiquitous for modeling such processes. While existing program verification algorithms do not take the statistical nature of program environments into account, in this dissertation we show how the statistical perspective can be fruitfully employed for reasoning about program behavior. In particular, we study two program verification problems where the statistical perspective on program environments allows us to go beyond the standard program verification literature.

First, we study the problem of verifying programs that are not expected nor required to exhibit correct behavior in all environments. Traditionally, such programs would be deemed incorrect and discarded. However, there is a growing usage of algorithms that learn programs from data. Learning programs that are correct in all environments is often computationally infeasible. Consequently, one frequently encounters programs that are not expected to exhibit the correct behavior in all possible environments. Probabilistic notions of program correctness become essential for certifying these programs as correct. Statistical modeling of program environments allows us to formulate such probabilistic statements of program correctness. In this dissertation, we restrict ourselves to verifying probabilistic correctness of learned neural networks. Notice that this notion of probabilistic correctness maybe viewed as analogous, in spirit, to the notion of average-case complexity from complexity theory [1]. Instead of fixating on the worst case behaviors of a program, we instead try to establish correctness in the common case.

Second, we study the standard verification problem of checking if programs satisfy non-probabilistic notions of correctness using new verification algorithms that fruitfully exploit the statistical perspective on program environments. Typically, program verification algorithms make no assumptions about the program under analysis or about the program environment, except for the assumptions already provided by the programmer, for instance, type annotations or logical preconditions on free variables. However, decision problems addressed by program verification are well-known to be undecidable [2], so verification algorithms are not always guaranteed to find a proof of program correctness, even if such a proof exists. One approach to addressing this problem is to allow verification algorithms to make any required assumptions, about the environment and about the program under analysis, that can help construct a proof, and to embed these assumptions as run time checks in the program. Well-studied program verification approaches like gradual typing [3, 4] and hybrid typing [5, 6] already employ this strategy. However, these techniques can have excessive run time overheads [7]. Worse, the assumptions about the environment and the program can be too strong and lead to frequent run time violations. However, modeling the program environment statistically enables the design of verification algorithms that can estimate the probability of an assumption being valid with respect the environment statistical model, and therefore, enable the choice of assumptions that are unlikely to be violated at run time.

Employing the statistical perspective on program environments, we make two contributions in this dissertation. The first contribution is an application of the idea of probabilistic program correctness to neural network verification. As we have already described, probabilistic notions of program correctness are particularly suited in the context of certifying the correctness of neural networks, but such ideas have been under-explored in the literature. In this dissertation, we formulate a new notion of correctness for neural networks (referred to as *probabilistic robustness* or *probabilistic Lipschitzness*) and present verification algorithms for the same. The second contribution of this dissertation is a generic

algorithmic framework that extends the well-studied framework of abstract interpretation [8, 9] to enable construction of algorithms (i.e., abstract interpreters) that can exploit the statistical model of program environment for computing hypothetical semantic invariants of the program, such that the hypotheses are unlikely to be violated at run time. We refer to such abstract interpreters as *observational abstract interpreters*.

We note that the notion of program verification with statistically modeled program environments is not new [10, 11], but applications of this perspective have been lacking. However, with the advances in deep learning and computational statistics in recent years, algorithmically learning statistical models of program environments is more feasible than ever. In particular, the advances in neural network based generative modeling of data generation process [12, 13] have made it possible to learn very accurate and complicated statistical models of program environments/inputs with almost no human intervention.

1.2 Probabilistic Robustness of Neural Networks

A neural network f is probabilistically robust if, for a randomly generated pair of inputs, f is likely to demonstrate k-Lipschitzness, i.e., the distance between the outputs computed by f is upper-bounded by the k^{th} multiple of the distance between the pair of inputs. We name this property, *probabilistic Lipschitzness*. Proving neural networks robust has been an open and urgent problem since Szegedy et al. [14] first noticed that state-of-the-art neural networks learned to perform the task of image recognition were unstable - small changes to the inputs caused the learned neural networks to produce large, unexpected, and undesirable changes in the outputs. In other words, small changes to the images, imperceptible to humans, caused the neural networks to produce very different image labels. Though various notions of neural network robustness have been discussed in the literature, a majority of the existing literature has focused on local notions of robustness. Informally, a neural network is *locally robust* at a specific input, x_0, if it behaves robustly in a bounded, local region of the input Euclidean space centered at x_0. We are however interested in the global

notion of probabilistic Lipschitzness that, while providing a stronger correctness guarantee, also allows the flexibility of neural networks behaving non-robustly at unlikely inputs. Our formulation of probabilistic Lipschitzness assumes it feasible to construct a statistical model of the process generating the inputs of a neural network. We find this a reasonable assumption given the rapid advances in algorithms for learning generative statistical models represented via neural networks

We present two verification algorithms for proving probabilistic robustness of neural networks. While our first algorithm is too expensive to be useful in practice, our second algorithm is practically feasible. This algorithm requires that we model the statistical program environment and the neural network under analysis, together, as a program in a simple, first-order, imperative, probabilistic programming language, $pcat$. Inspired by a large body of existing literature, we define a denotational semantics for this language. Then we develop a sound *local Lipschitzness* analysis for cat, a non-probabilistic sublanguage of $pcat$. This analysis can compute an upper bound of the Lipschitzness of a neural network in a bounded region of the input set. We next present a provably correct algorithm, `PROLIP`, that analyzes the behavior of a neural network in a user-specified box-shaped input region and computes, (i) lower bounds on the probabilistic mass of such a region with respect to the generative model, and (ii) upper bounds on the Lipschitz constant of the neural network in this region, with the help of the local Lipschitzness analysis. Finally, we present a sketch of a proof-search algorithm that uses `PROLIP` as a primitive for finding proofs of probabilistic Lipschitzness. Verification of probabilistic robustness of neural networks is described in detail in chapter 2.

1.3 Observational Abstract Interpreters

A common strategy used by program verification algorithms in the search for proofs of program correctness is to compute semantic program invariants that serve as lemmas in the proofs. A semantic invariant is a simplified representation of the meaning of a program.

Practically, semantic invariants should be efficiently computable even when the program under analysis is non-terminating. The well-studied algorithmic framework of abstract interpretation provides a standard recipe for constructing algorithms that compute semantic program invariants. In this dissertation, we present *observational abstract interpreters*, a new algorithmic framework for designing algorithms that compute semantic program invariants in statistically modeled program environments. The invariants computed by observational abstract interpreters are permitted to be hypothetical, i.e. valid only under specific hypotheses about the program environment. These hypotheses are inferred from observed behaviors of the program. Observational abstract interpreters do not require the statistical model of the environment to be known, but do assume that the observations of program behavior are generated by drawing independent samples from an environment statistical model and running the program where the free variables are substituted with the sampled values. This assumption allows us one to infer hypotheses from the observations that are likely to be valid with respect to the environment statistical model. In order to ensure that the proofs of program correctness computed using hypothetical semantic invariants are valid, we embed the hypotheses as run time/dynamic checks in the program.

We formalize our ideas in the context of a simple higher-order language (λ_S) with integers. We develop a generic observational abstract interpreter for λ_S, drawing inspiration from the abstracting abstract machines (AAM) recipe of Van Horn and Might [15] for abstract interpreter construction. Observational abstract interpreters are structured as monadic abstract interpreters [16, 17, 18], have a monadic structure, and are capable of making hypotheses based on program observations during the computation of semantic program invariants. We present an instantiation of the generic observational abstract interpreter for λ_S, yielding an observational variant of interval analysis for λ_S. Observational abstract interpreters are presented in detail in chapter 3.

CHAPTER 2

PROBABILISTIC LIPSCHITZ ANALYSIS OF NEURAL NETWORKS

2.1 Introduction

Neural networks (NNs) are useful for modeling a variety of computational tasks that are beyond the reach of manually written programs. We like to think of NNs as programs in a first-order programming language specialized to operate over vectors from high-dimensional Euclidean spaces. However, NNs are algorithmically learned from observational data about the task being modeled. These tasks typically represent natural processes for which we have large amounts of data but limited mathematical understanding. For example, NNs have been successful at image recognition [19] - assigning descriptive labels to images. In this case, the underlying natural process that we want to mimic computationally is image recognition as it happens in the human brain. However, insufficient mathematical theory about this task makes it hard to develop a hand-crafted algorithm.

Given that NNs are discovered algorithmically, it is important to ensure that a learned NN actually models the computational task of interest. With the perspective of NNs as programs, this reduces to proving that the NN behaves in accordance with the formal specification of the task at hand. Unfortunately, limited mathematical understanding of the tasks implies that, in general, we are unable to even state the formal specification. In fact, it is precisely in situations where we are neither able to manually design an algorithm nor able to provide formal specifications in which NNs tend to be deployed. This inability to verify or make sense of the computation represented by a NN is one of the primary challenges to the widespread adoption of NNs, particularly for safety critical applications. In practice, NNs are tested on a limited number of manually provided tests (referred to as test data) before deploying. However, a natural question is, what formal correctness guarantees, if

any, can we provide about NNs?

A hint towards a useful notion of correctness comes from an important observation about the behavior of NNs, first made by [14]. They noticed that state-of-the-art NNs that had been learned to perform the image recognition task were unstable - small changes in the inputs caused the learned NNs to produce large, unexpected, and undesirable changes in the outputs. In the context of the image recognition task, this meant that small changes to the images, imperceptible to humans, caused the NN to produce very different labels. The same phenomenon has been observed by others, and in the context of very different tasks, like natural language processing [20, 21] and speech recognition [22, 23, 24]. This phenomenon, commonly referred to as lack of *robustness*, is widespread and undesirable. This has motivated a large body of work (see [25, 26, 27] for broad but non-exhaustive surveys) on algorithmically proving NNs robust. These approaches differ not only in the algorithms employed but also in the formal notions of robustness that they prove.

An majority of the existing literature has focused on local notions of robustness. Informally, a NN is *locally robust* at a specific input, x_0, if it behaves robustly in a bounded, local region of the input Euclidean space centered at x_0. There are multiple ways of formalizing this seemingly intuitive property. A common approach is to formalize this property as, $\forall x.(\|x - x_0\| \leqslant r) \rightarrow \phi((fx), (fx_0))$, where f is the NN to be proven locally robust at x_0, (fx) represents the result of applying the NN f on input x, $\phi((fx), (fx_0))$ represents a set of linear constraints imposed on (fx), and $\|\cdot\|$ represents the norm or distance metric used for measuring distances in the input and output Euclidean spaces (typically, an l_p norm is used with $p \in \{1, 2, \infty\}$). An alternate, less popular, formulation of local robustness, referred to as *local Lipschitzness* at a point, requires that $\forall x, x'.(\|x - x_0\| \leqslant r) \wedge (\|x' - x_0\| \leqslant r) \rightarrow (\|fx - fx'\| \leqslant k * \|x - x'\|)$. Local Lipschitzness ensures that in a ball of radius r centered at x_0, changes in the input only lead to bounded changes in the output. One can derive other forms of local robustness from local Lipschitzness. (see Theorem 3.2 in [28]). We also find local Lipschitzness to be an aesthetically more pleasing and natural property

of a function. But, local Lipschitzness is a relational property [29, 30]/hyperproperty [31] unlike the first formulation, which is a safety property [32]. Algorithms for proving safety properties of programs have been more widely studied and there are a number of mature approaches to build upon, which may explain the prevalence of techniques for proving the former notion of local robustness. For instance, [33, 34] are based on variants of polyhedral abstract interpretation [35], [36, 37, 38] encode the local robustness verification problem as an SMT constraint.

Local robustness (including local Lipschitzness) is a useful but limited guarantee. For inputs where the NN has not been proven to be locally robust, no guarantees can be given. Consequently, a global notion of robustness is desirable. Local Lipschitzness can be extended to a global property - a NN f is *globally Lipschitz* or k-*Lipschitz* if, $\forall x, x'.(\|fx - fx'\| \leqslant k * \|x - x'\|)$. Algorithms have been proposed in programming languages and machine learning literature for computing Lipschitz constant upper bounds. Global robustness is guaranteed if the computed upper bound is $\leqslant k$.

Given the desirability of global robustness over local robustness, the focus on local robustness in the existing literature may seem surprising. There are two orthogonal reasons that, we believe, explain this state of affairs - (i) proving global Lipschitzness, particularly with a tight upper bound on the Lipschitz constant, is more technically and computationally challenging than proving local Lipschitzness, which is itself hard to prove due its relational nature; (ii) requiring NNs to be globally Lipschitz with some low constant k can be an excessively stringent specification, unlikely to be met by most NNs in practice. NNs, unlike typical programs, are algorithmically learnt from data. Unless the learning algorithm enforces the global robustness constraint, it is unlikely for a learned NN to exhibit this "strong" property. Unfortunately, learning algorithms are ill-suited for imposing such logical constraints. These algorithms search over a set of NNs (referred to as the hypothesis class) for the NN minimizing a cost function (referred to as loss function) that measures the "goodness" of a NN for modeling the computational task at hand. These algorithms are

greedy and iterative, following the gradient of the loss function. Modifying the loss function in order to impose the desired logical constraints significantly complicates the function structure and makes the gradient-based, greedy learning algorithms ineffective.[1]

Consequently, in this work, we focus on a probabilistic notion of global robustness. This formulation, adopted from [40], introduces a new mathematical object to the NN verification story, namely, a probability measure over the inputs to the NN under analysis. One assumes it feasible to construct a statistical model of the process generating the inputs of a NN. We find this a reasonable assumption given the rapid advances in algorithms for learning generative models of data [13, 12]. Such a statistical model yields a distribution D over the inputs of the NN. Given distribution D and a NN f, this notion of robustness, that we refer to as *probabilistic Lipschitzness*, is formally stated as,

$$\Pr_{x,x' \sim D} \left(\|fx - fx'\| \leqslant k * \|x - x'\| \;\middle|\; \|x - x'\| \leqslant r \right) \geqslant 1 - \epsilon$$

This says that if we randomly draw two samples, x and x' from the distribution D, then, under the condition that x and x' are r-close, there is a high probability ($\geqslant (1 - \epsilon)$) that NN f behaves stably for these inputs. If the parameter $\epsilon = 0$ and $r = \infty$, then we recover the standard notion of k-Lipschitzness. Conditioning on the event of x and x' being r-close reflects the fact that we are primarily concerned with the behavior of the NN on pairs of inputs that are close.

To algorithmically search for proofs of probabilistic Lipschitzness, we model generative models and NNs together as programs in a simple, first-order, imperative, probabilistic programming language, $pcat$. First-order probabilistic programming languages with a `sample` construct, like $pcat$, have been well-studied.[2] Programs in $pcat$ denote transformers from Euclidean spaces to probability measures over Euclidean spaces. $pcat$, inspired by the non-probabilistic language cat [34], is explicitly designed to model NNs, with vectors in $\mathbb{R}^n$ as

[1] Recent work has tried to combine loss functions with logical constraints [39].

[2] $pcat$ has no `observe` or `score` construct and cannot be used for Bayesian reasoning.

the basic datatype. The suitability of $pcat$ for representing generative models stems from the fact that popular classes of generative models (for instance, the generative network of generative adversarial networks [12] and the decoder network of variational autoencoders [13]) are represented by `NN`s. Samples from the input distribution D are obtained by drawing a sample from a standard distribution (typically a normal distribution) and running this sample through generative or decoder networks. In $pcat$, this can be represented as the program, $z \leftsquigarrow N(0,1); g$, where the first statement represents the sampling operation (referred to as sampling from the latent space, with z as the latent variable) and g is the generative or decoder `NN`. If the `NN` to be analyzed is f, then we can construct the program, $z \leftsquigarrow N(0,1); g; f$, in $pcat$, and subject it to our analysis.

Adapting a language-theoretic perspective allows us to study the problem in a principled, general manner and utilize existing program analysis and verification literature. In particular, we are interested in sound algorithms that can verify properties of probabilistic programs without needing manual intervention. Thus approaches based on interactive proofs [41, 42], requiring manually-provided annotations and complex side-conditions [43, 44, 45] or only providing statistical guarantees [46, 47] are precluded. Frameworks based on abstract interpretation [48, 49] are helpful for thinking about analysis of probabilistic programs but we focus on a class of completely automated proof-search algorithms [10, 50, 51] that only consider probabilistic programs where all randomness introducing statements (i.e., `sample` statements) are independent of program inputs, i.e. samples are drawn from fixed, standard probability distributions, similar to our setting. These algorithms analyze the program to generate symbolic constraints (i.e., sentences in first-order logic with theories supported by SMT solvers) and then compute the probability mass or "volume", with respect to a fixed probability measure, of the set of values satisfying these constraints. These algorithms are unsuitable for parametric probability measures but suffice for our problem. Both generating symbolic constraints and computing volumes can be computationally expensive (and even intractable for large programs), so a typical strategy is to break

down the task into simpler sub-goals. This is usually achieved by defining the notion of "program path" and analyzing each path separately. This "per path" strategy is unsuitable for NNs, with their highly-branched program structure. We propose partitioning the program input space (i.e., the latent space in our case) into box-shaped regions, and analyzing the program behavior separately on each box. The box partitioning strategy offers two important advantages - (i) by not relying explicitly on program structure to guide partitioning strategy, we have more flexibility to balance analysis efficiency and precision; (ii) computing the volume of boxes is easier than computing the same for sets with arbitrary or even convex structure.

For the class of probabilistic programs we are interested in (with structure, $z \leftsquigarrow N(0,1); g; f$), the box-partitioning strategy implies repeatedly analyzing the program $g; f$ while restricting z to from box shaped regions. In every run, the analysis of $g; f$ involves computing a box-shaped overapproximation, x_B, of the outputs computed by g when z is restricted to some specific box z_B and computing an upper bound on the local Lipschitz constant of f in the box-shaped region x_B. We package these computations, performed in each iteration of the proof-search algorithm, in an algorithmic primitive, PROLIP.

For computing upper bounds on local Lipschitz constants, we draw inspiration from existing literature on Lipschitz analysis of programs [52] and NNs [14, 53, 54, 55, 56, 57, 58, 59, 60]. In particular, we build on the algorithms presented in [57, 58]. We translate these algorithms in to our language-theoretic setting and present the local Lipschitzness analysis in the form of an abstract semantics for the cat language, which is a non-probabilistic sub-language of $pcat$. In the process, we also simplify and generalize the original algorithms.

To summarize, our primary contributions in this work are - (i) we present a provably sound algorithmic primitive PROLIP and a sketch of a proof-search algorithm for probabilistic Lipschitzness of NNs, (ii) we develop a simplified and generalized version of the local Lipschitzness analysis in [57], capable of computing an upper bound on the local Lipschitz constant of box-shaped input regions for any program in the cat language, (iii) we

$$
\begin{array}{rl}
\text{(variables)} & x, y \in V \\
\text{(naturals)} & m, n \in \mathbb{N} \\
\text{(weights)} & w \in \bigcup_{m,n \in \mathbb{N}} \mathbb{R}^{m \times n} \\
\text{(biases)} & \beta \in \bigcup_{n \in \mathbb{N}} \mathbb{R}^{n}
\end{array}
$$

$$
\begin{array}{rcl}
s & ::= & \mathbf{skip} \mid y \leftarrow w \cdot x + \beta \mid y \leftsquigarrow N(0,1) \mid s; s \mid \mathbf{if}\ b\ \mathbf{then}\ s\ \mathbf{else}\ s \\
s^{-} & ::= & \mathbf{skip} \mid y \leftarrow w \cdot x + \beta \mid s^{-}; s^{-} \mid \mathbf{if}\ b\ \mathbf{then}\ s^{-}\ \mathbf{else}\ s^{-} \\
b & ::= & \boldsymbol{\pi}(x,m) \geqslant \boldsymbol{\pi}(y,n) \mid \boldsymbol{\pi}(x,n) \geqslant 0 \mid \boldsymbol{\pi}(x,n) < 0 \mid b \wedge b \mid \neg b \\
e & ::= & \boldsymbol{\pi}(x,n) \mid w \cdot x + \beta
\end{array}
$$

Figure 2.1: $pcat$ syntax

develop a strategy for computing proofs of probabilistic programs that limits probabilistic reasoning to volume computation of regularly shaped sets with respect to standard distributions, (iv) we implement the PROLIP algorithm, and evaluate its computational complexity.

2.2 Language Definition

2.2.1 Language Syntax

$pcat$ (probabilistic conditional affine transformations) is a first-order, imperative probabilistic programming language, inspired by the cat language [34]. $pcat$ describes always terminating computations on data with a base type of vectors over the field of reals (i.e., of type $\bigcup_{n \in \mathbb{N}} \mathbb{R}^n$). $pcat$ is not meant to be a practical language for programming, but serves as a simple, analyzable, toy language that captures the essence of programs structured like NNs. We emphasize that $pcat$ does not capture the learning component of NNs. We think of $pcat$ programs as objects learnt by a learning algorithm (commonly stochastic gradient descent with symbolic gradient computation). We want to analyze these learned programs and prove that they satisfy the probabilistic Lipschitzness property.

$pcat$ can express a variety of popular NN architectures and generative models. For instance, $pcat$ can express ReLU, convolution, maxpool, batchnorm, transposed convolution, and other structures that form the building blocks of popular NN architectures. We describe the encodings of these structures in subsection 2.2.3. The probabilistic nature of $pcat$ fur-

ther allows us to express a variety of generative models, including different generative adversarial networks (GANs) [12] and variational autoencoders (VAEs) [13].

$pcat$ syntax is defined in Figure 2.1. $pcat$ variable names are drawn from a set V and refer to vector of reals. Constant matrices and vectors appear frequently in $pcat$ programs, playing the role of learned weights and biases of `NNs`, and are typically represented by w and β, respectively. Programs in $pcat$ are composed of basic statements for performing linear transformations of vectors ($y \leftarrow w \cdot x + \beta$) and sampling vectors from normal distributions ($y \leftsquigarrow N(0,1)$). Sampling from parametric distributions is not allowed. Programs can be composed sequentially ($s; s$) or conditionally ($\mathbf{if}\ b\ \mathbf{then}\ s\ \mathbf{else}\ s$). $pcat$ does not have a loop construct, acceptable as many `NN` architectures do not contain loops. $pcat$ provides a projection operator $\boldsymbol{\pi}(x, n)$ that reads the n^{th} element of the vector referred by x. For $pcat$ programs to be well-formed, all the matrix and vector dimensions need to fit together. Static analyses [61, 62] can ensure correct dimensions. In the rest of the paper, we assume that the programs are well-formed.

2.2.2 Language Semantics

$$
\begin{aligned}
\Sigma &\triangleq V \to \textstyle\bigcup_{n\in\mathbb{N}} \mathbb{R}^n \\
[\![e]\!] &: \Sigma \to \textstyle\bigcup_{n\in\mathbb{N}} \mathbb{R}^n \\
[\![\boldsymbol{\pi}(x,n)]\!](\sigma) &= \sigma(x)_n \\
[\![w \cdot x + \beta]\!](\sigma) &= w \cdot \sigma(x) + \beta
\end{aligned}
$$

$$
\begin{aligned}
[\![b]\!] &: \Sigma \to \{\mathbf{tt}, \mathbf{ff}\} \\
[\![\boldsymbol{\pi}(x,m) \geqslant \boldsymbol{\pi}(y,n)]\!](\sigma) &= \mathbf{if}\ ([\![\boldsymbol{\pi}(x,m)]\!](\sigma) \geqslant [\![\boldsymbol{\pi}(y,n)]\!](\sigma))\ \mathbf{then\ tt\ else\ ff} \\
[\![\boldsymbol{\pi}(x,m) \geqslant 0]\!](\sigma) &= \mathbf{if}\ ([\![\boldsymbol{\pi}(x,m) \geqslant 0]\!](\sigma))\ \mathbf{then\ tt\ else\ ff} \\
[\![\boldsymbol{\pi}(x,m) < 0]\!](\sigma) &= \mathbf{if}\ ([\![\boldsymbol{\pi}(x,m)]\!](\sigma) < 0)\ \mathbf{then\ ttelse\ ff} \\
[\![b_1 \wedge b_2]\!](\sigma) &= [\![b_1]\!](\sigma) \wedge [\![b_2]\!](\sigma) \\
[\![\neg b]\!](\sigma) &= \mathbf{if}\ ([\![b]\!] = \mathbf{tt})\ \mathbf{then\ ff\ else\ tt}
\end{aligned}
$$

$$
\begin{aligned}
[\![s]\!] &: \Sigma \to P(\Sigma) \\
[\![\mathbf{skip}]\!](\sigma) &= \delta_\sigma \\
[\![y \leftarrow w \cdot x + \beta]\!](\sigma) &= \delta_{\sigma[y \mapsto [\![w\cdot x+\beta]\!](\sigma)]} \\
[\![y \leftsquigarrow N(0,1)]\!](\sigma) &= \mathbb{E}_{a\sim N(0,1)}[\lambda\nu.\delta_{\sigma[y\mapsto\nu]}] \\
[\![s_1; s_2]\!](\sigma) &= \mathbb{E}_{\tilde{\sigma}\sim[\![s_1]\!](\sigma)}[[\![s_2]\!]] \\
[\![\mathbf{if}\ b\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2]\!](\sigma) &= \mathbf{if}\ ([\![b]\!](\sigma))\ \mathbf{then}\ [\![s_1]\!](\sigma)\ \mathbf{else}\ [\![s_2]\!](\sigma)
\end{aligned}
$$

$$
\begin{aligned}
\widehat{[\![s]\!]} &: P(\Sigma) \to P(\Sigma) \\
\widehat{[\![s]\!]}(\mu) &= \mathbb{E}_{\sigma\sim\mu}[[\![s]\!]]
\end{aligned}
$$

$$
\begin{aligned}
\widetilde{[\![s^-]\!]} &: \Sigma \to \Sigma \\
\widetilde{[\![\mathbf{skip}]\!]}(\sigma) &= \sigma \\
[\![y \leftarrow \widetilde{w \cdot x} + \beta]\!](\sigma) &= \sigma[y \mapsto [\![w \cdot x + \beta]\!](\sigma)] \\
\widetilde{[\![s_1; s_2]\!]}(\sigma) &= \widetilde{[\![s_2]\!]}(\widetilde{[\![s_1]\!]}(\sigma)) \\
[\![\mathbf{if}\ b\ \widetilde{\mathbf{then}\ s_1}\ \mathbf{else}\ s_2]\!](\sigma) &= \mathbf{if}\ ([\![b]\!](\sigma))\ \mathbf{then}\ \widetilde{[\![s_1]\!]}(\sigma)\ \mathbf{else}\ \widetilde{[\![s_2]\!]}(\sigma)
\end{aligned}
$$

Figure 2.2: *pcat* denotational semantics

We define the denotational semantics of $pcat$ in Figure 3.2, closely following those presented in [41]. We present definitions required to understand these semantics.

Definition 1. *A $\sigma-$algebra on a set X is a set Σ of subsets of X such that it contains X, is closed under complements and countable unions. A set with a $\sigma-$algebra is a measurable space and the subsets in Σ are measurable.*

A measure on a measurable space (X, Σ) is a function $\mu : \Sigma \to [0, \infty]$ such that $\mu(\varnothing) = 0$ and $\mu(\bigcup_{i\in\mathbb{N}} B_i) = \sum_{i\in\mathbb{N}} \mu(B_i)$ such that B_i is a countable family of disjoint measurable sets. A probability measure or probability distribution is a measure μ with $\mu(X) = 1$.

Given set X, we use $P(X)$ to denote the set of all probability measures over X. A *Dirac distribution* centered on x, written δ_x, maps x to 1 and all other elements of the underlying set to 0. Note that when giving semantics to probabilistic programming languages, it is typical to consider sub-distributions (measures such that $\mu(X) \leqslant 1$ for a measurable space (X, Σ)), as all programs in $pcat$ terminate, we do not describe the semantics in terms of sub-distributions. Next, following [41], we give a monadic structure to probability distributions.

Definition 2. *Let $\mu \in P(A)$ and $f : A \to P(B)$. Then, $\mathbb{E}_{a\sim\mu}[f] \in P(B)$ is defined as,* $\mathbb{E}_{a\sim\mu}[f] \triangleq \lambda\nu. \int_A f(a)(\nu)\ d\mu(a)$

Note that in the rest of the paper, we write expressions of the form $\int_A f(a)\ d\mu(a)$ as $\int_{a\in A} \mu(a) \cdot f(a)$ for notational convenience. The metalanguage used in Figure 3.2 and the rest of the paper is standard first-order logic with ZFC set theory, but we borrow notation from a variety of sources including languages like C and ML as well as standard set-theoretic notation. As needed, we provide notational clarification.

We define the semantics of $pcat$ with respect to the set Σ of states. A state σ is a map from variables V to vectors of reals of any finite dimension. The choice of real vectors as the basic type of values is motivated by the goal of $pcat$ to model `NN` computations.

The set $P(\Sigma)$ is the set of probability measures over Σ. A $pcat$ statement transforms a distribution over Σ to a new distribution over the same set. $[\![e]\!]$ and $[\![b]\!]$ denote the semantics of expressions and conditional checks, respectively. Expressions map states to vectors of reals while conditional checks map states to boolean values.

The semantics of statements are defined in two steps. We first define the standard semantics $[\![s]\!]$ where statements map incoming states to probability distributions. Next, the lifted semantics, $\widehat{[\![s]\!]}$, transform a probability distribution over the states, say μ, to a new probability distribution. The lifted semantics ($\widehat{[\![s]\!]}$) are obtained from the standard semantics ($[\![s]\!]$) using the monadic construction of Definition 2. Finally, we also defined a lowered semantics ($\widetilde{[\![s^-]\!]}$) for the cat sublanguage of $pcat$. As per these lowered semantics, statements are maps from states to states. Moreover, the lowered semantics of cat programs is tightly related to their standard semantics, as described by the following lemma.

Lemma 3. *(Equivalence of semantics)*
$\forall p \in s^-, \sigma \in \Sigma.\ [\![p]\!](\sigma) = \delta_{\widetilde{[\![p]\!]}(\sigma)}$

Proof. We prove this by induction on the structure of statements in s^-.
We first consider the base cases:

(i) $\underline{\mathbf{skip}}$

By definition, for any state σ,

$[\![\mathbf{skip}]\!](\sigma) = \delta_\sigma = \delta_{\widetilde{[\![\mathbf{skip}]\!]}(\sigma)}$

(ii) $\underline{y \leftarrow w \cdot x + \beta}$

Again, by definition, for any state σ,

$[\![y \leftarrow w \cdot x + \beta]\!](\sigma) = \delta_{\sigma[y \mapsto [\![w \cdot x + \beta]\!](\sigma)]} = \delta_{\widetilde{[\![y \leftarrow w \cdot x + \beta]\!]}(\sigma)}$

Next, we consider the inductive cases:

(iii) $\underline{s_1^-; s_2^-}$

$$\begin{aligned}
[\![s_1^-; s_2^-]\!](\sigma) &= \mathbb{E}_{\tilde{\sigma}\sim[\![s_1^-]\!](\sigma)}[[\![s_2^-]\!]] \\
&= \lambda\nu.\textstyle\int_{\tilde{\sigma}\in\Sigma} [\![s_1^-]\!](\sigma)(\tilde{\sigma}) \cdot [\![s_2^-]\!](\tilde{\sigma})(\nu) \\
&= \lambda\nu.\textstyle\int_{\tilde{\sigma}\in\Sigma} \delta_{\widetilde{[\![s_1^-]\!](\sigma)}}(\tilde{\sigma}) \cdot \delta_{\widetilde{[\![s_2^-]\!](\tilde{\sigma})}}(\nu) \quad \text{(using inductive hypothesis)} \\
&= \lambda\nu.\delta_{\widetilde{[\![s_2^-]\!]}(\widetilde{[\![s_1^-]\!]}(\sigma))}(\nu) \\
&= \delta_{\widetilde{[\![s_2^-]\!]}(\widetilde{[\![s_1^-]\!]}(\sigma))} \\
&= \delta_{\widetilde{[\![s_1^-; s_2^-]\!]}(\sigma)}
\end{aligned}$$

(iv) $\underline{\textbf{if } b \textbf{ then } s_1^- \textbf{ else } s_2^-}$

$$\begin{aligned}
[\![\textbf{if } b \textbf{ then } s_1^- \textbf{ else } s_2^-]\!](\sigma) &= \textbf{if } ([\![b]\!](\sigma)) \textbf{ then } [\![s_1^-]\!](\sigma) \textbf{ else } [\![s_2^-]\!](\sigma) \\
&= \textbf{if } ([\![b]\!](\sigma)) \textbf{ then } \delta_{\widetilde{[\![s_1^-]\!](\sigma)}} \textbf{ else } \delta_{\widetilde{[\![s_2^-]\!](\sigma)}} \\
&\qquad\qquad \text{(using inductive hypothesis)} \\
&= \delta_{\textbf{if } ([\![b]\!](\sigma)) \textbf{ then } \widetilde{[\![s_1^-]\!]}(\sigma) \textbf{ else } \widetilde{[\![s_2^-]\!]}(\sigma)} \\
&= \delta_{\widetilde{[\![\textbf{if } ([\![b]\!](\sigma)) \textbf{ then } s_1^- \textbf{ else } s_2^-]\!]}(\sigma)}
\end{aligned}$$

■

The lemma states that one can obtain the standard probabilistic semantics for a program p in cat, given an initial state σ, by a Dirac delta distribution centered at $\widetilde{[\![p]\!]}(\sigma)$. Using this lemma, one can prove the following useful corollary.

Corollary 4. $\forall p \in s^-, \sigma \in \Sigma, \mu \in P(\Sigma).\ \widehat{[\![p]\!]}(\mu)(\widetilde{[\![p]\!]}(\sigma)) \geqslant \mu(\sigma)$

Proof. By definition,

$$\begin{aligned}
\widehat{[\![p]\!]}(\mu) &= \mathbb{E}_{\sigma\sim\mu}[[\![p]\!]] \\
&= \lambda\nu.\textstyle\int_{\sigma\in\Sigma} \mu(\sigma) \cdot [\![p]\!](\sigma)(\nu) \\
&= \lambda\nu.\textstyle\int_{\sigma\in\Sigma} \mu(\sigma) \cdot \delta_{\widetilde{[\![p]\!](\sigma)}}(\nu) \quad \text{(using previous lemma)}
\end{aligned}$$

Now suppose, $\nu = \widetilde{[\![p]\!]}(\tilde{\sigma})$. Then, continuing from above,

$$\begin{aligned}\widehat{[\![p]\!]}(\mu)(\widetilde{[\![p]\!]}(\tilde{\sigma})) &= \textstyle\int_{\sigma\in\Sigma} \mu(\sigma) \cdot \delta_{\widetilde{[\![p]\!]}(\sigma)}(\widetilde{[\![p]\!]}(\tilde{\sigma})) \\ &\geqslant \mu(\tilde{\sigma})\end{aligned}$$

■

2.2.3 Translating Neural Networks Into $pcat$

NNs are often described as a sequential composition of "layers", with each layer describing the computation to be performed on an incoming vector. Many commonly used layers can be expressed in the $pcat$ language. For instance, [34] describes the translation of maxpool, convolution, ReLU, and fully connected layers into the cat language. Here, we describe the translation of two other common layers, namely, the batchnorm layer [63] and the transposed convolution layer (also referred to as the deconvolution layer) [64].

Batchnorm layer. A batchnorm layer typically typically expects an input $x \in \mathbb{R}^{C\times H\times W}$ which we flatten, using a row-major form in to $x' \in \mathbb{R}^{C\cdot H\cdot W}$ where, historically, C denotes the number of channels in the input, H denotes the height, and W denotes the width. For instance, given an RGB image of dimensions 28 × 28 pixels, $H = 28$, $W = 28$, and $C = 3$.

A batchnorm layer is associated with vectors m and v such that $\mathbf{dim}(m) = \mathbf{dim}(v) = C$ where $\mathbf{dim}(\cdot)$ returns the dimension of a vector. m and v represent the running-mean and running-variance of the values in each channel observed during the training time of the NN. A batchnorm layer is also associated with a scaling vector s^1 and a shift vector s^2, both also of dimension c. For a particular element $x_{i,j,k}$ in the input, the corresponding output element is $s^1_i \cdot (\frac{x_{i,j,k}-m_i}{\sqrt{v_i+\epsilon}}) + s^2_i$ where ϵ is a constant that is added for numerical stability (commonly set to $1e^{-5}$). Note that the batchnorm operation produces an output of the same dimensions as the input. We can represent the batchnorm operation by the statement, $y \leftarrow w \cdot x' + \beta$, where x' is the flattened input, w is a weight matrix of dimension $C \cdot H \cdot W \times C \cdot H \cdot W$ and β is a bias vector of dimension $C \cdot H \cdot W$, such that,

$$w = I \cdot [\frac{s^1_{\lfloor i/H \cdot W \rfloor}}{\sqrt{v_{\lfloor i/H \cdot W \rfloor} + \epsilon}} \mid i \in \{1, ..., C \cdot H \cdot W\}]$$

$$\beta = [-\frac{s^1_{\lfloor i/H \cdot W \rfloor} \cdot m_{\lfloor i/H \cdot W \rfloor}}{\sqrt{v_{\lfloor i/H \cdot W \rfloor} + \epsilon}} + s^2_{\lfloor i/H \cdot W \rfloor} \mid i \in \{1, ..., C \cdot H \cdot W\}]$$

where I is the identity matrix with dimension $(C \cdot H \cdot W, C \cdot H \cdot W)$, $\lfloor \cdot \rfloor$ is the floor operation that rounds down to an integer, and [|] is the list builder/comprehension notation.

Transposed convolution layer. A convolution layer applies a kernel or a filter on the input vector and typically, compresses this vector so that the output vector is of a smaller dimension. A deconvolution or transposed convolution layer does the opposite - it applies the kernel in a manner that produces a larger output vector. A transposed convolution layer expects an input $x \in \mathbb{R}^{C_{in} \times H_{in} \times W_{in}}$ and applies a kernel $k \in \mathbb{R}^{C_{out} \times C_{in} \times K_h \times K_w}$ using a stride S. For simplicity of presentation, we assume that $K_h = K_w = K$ and $W_{in} = H_{in}$. In $pcat$, the transposed convolution layer can be expressed by the statement, $y \leftarrow w \cdot x'$, where x' is the flattened version of input x, w is a weight matrix that we derive from the parameters associated with the transposed convolution layer, and the bias vector, β, is a zero vector in this case. To compute the dimensions of the weight matrix, we first calculate the height (H_{out}) and width (W_{out}) of each channel in the output using formulae, $H_{out} = H_{in} \cdot S + K$, and $W_{out} = W_{in} \cdot S + K$. Since we assume $W_{in} = H_{in}$, we have $W_{out} = H_{out}$ here. Then, the dimension of w is $C_{out} \cdot H_{out} \cdot W_{out} \times C_{in} \cdot H_{in} \cdot W_{in}$, and the definition of w is as follows,

$$w = \left[\begin{array}{lll|l} \textbf{let} & \textbf{in} x = & \lceil i/C_{out} \rceil & \textbf{in} & \\ \textbf{let} & \textbf{in} y = & \lceil j/C_{in} \rceil & \textbf{in} & \\ \textbf{let} & \textbf{in} h = & 1 + \lfloor ((i \bmod C_{out}) - (\lfloor ((j \bmod C_{in}) - 1)/H_{in} \rfloor \cdot & & \\ & & H_{out} \cdot S + 1 + (((j \bmod C_{in}) - 1) \bmod H_{in}) \cdot S))/H_{out} \rfloor & \textbf{in} & i \in I, \\ \textbf{let} & \textbf{in} w = & 1 + ((i \bmod C_{out}) - (\lfloor ((j \bmod C_{in}) - 1)/H_{in} \rfloor \cdot & & j \in J \\ & & H_{out} \cdot S + 1 + (((j \bmod C_{in}) - 1) \bmod H_{in}) \cdot S)) \bmod H_{out} & \textbf{in} & \\ \multicolumn{4}{l|}{\textbf{if } h, w \in \{1...K\} \textbf{ then } k_{x,y,h,w} \textbf{ else } 0} & \end{array} \right]$$

where $I = \{1, ..., C_{out} \cdot H_{out} \cdot W_{out}\}$ and $J = \{1, ..., C_{in} \cdot H_{in} \cdot W_{in}\}$

2.3 Lipschitz Analysis

A function f is locally Lipschitz in a bounded set S if, $\forall x, x' \in S. \|fx - fx'\| \leqslant k \cdot \|x - x'\|$, where $\|\cdot\|$ can be any l_p norm. Quickly computing tight upper bounds on the local Lipschitzness constant (k) is an important requirement of our proof-search algorithm for probabilistic Lipschitzness of $pcat$ programs. However, as mentioned previously, local Lipschitzness is a relational property (hyperproperty) and computing upper bounds on k can get expensive.

The problem can be made tractable by exploiting a known relationship between Lipschitz constants and directional directives of a function. Let f be a function of type $\mathbb{R}^m \to \mathbb{R}^n$, and let $S \subset \mathbb{R}^m$ be a convex bounded set. From [28] we know that the local Lipschitz constant of f in the region S can be upper bounded by the maximum value of the norm of the directional directives of f in S, where the directional directive, informally, is the derivative of f in the direction of some vector v. Since f is a vector-valued function (i.e., mapping vectors to vectors), the derivative (including directional derivative) of f appears as a matrix of the form, $\mathbf{J} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_m} \\ \cdots & & \cdots \\ \frac{\partial y_n}{\partial x_1} & \cdots & \frac{\partial y_n}{\partial x_m} \end{bmatrix}$, referred to as the Jacobian matrix of f (with x and y referring to the input and output of f). Moreover, to compute the norm of $\mathbf{J}$, i.e. $\|\mathbf{J}\|$, we use the operator norm, $\|\mathbf{J}\| = \inf\{c \geqslant 0 \mid \|\mathbf{J}v\| \leqslant c\|v\| \text{ for all } v \in \mathbb{R}^m\}$. Intuitively, thinking of a matrix M as a linear operator mapping between two vector spaces, the operator norm of M measures the maximum amount by which a vector gets "stretched" when mapped using M.

For piecewise linear functions with a finite number of "pieces"(i.e., the type of functions that can be computed by cat), using lemma 3.3 from [28], we can compute an upper bound on the Lipschitz constant by computing the operator norm of the Jacobian of each linear piece, and picking the maximum value. Since each piece of the function is linear, computing the Jacobian for a piece is straightforward. But the number of pieces in piecewise linear

functions represented by NNs (or cat programs) can be exponential in the number of layers in the NN, even in a bounded region S. Instead of computing the Jacobian for each piece, we instead define a static analysis inspired by the Fast-Lip algorithm presented in [57] that computes lower and upper bounds of each element (i.e., each partial derivative) appearing in the Jacobian. Since our analysis is sound, such an interval includes all the possible values of the partial derivative in a given convex region S. We describe this Jacobian analysis in the rest of the section.

2.3.1 Instrumented *cat* Semantics

$$
\begin{aligned}
\Sigma^D &\triangleq \Sigma \times (V \to ((\textstyle\bigcup_{m,n\in\mathbb{N}} (\mathbb{R})^{m\times n}) \times V) \\
[\![e]\!]_\delta &: \Sigma^D \to \textstyle\bigcup_{n\in\mathbb{N}} \mathbb{R}^n \times (V \to ((\bigcup_{m,n\in\mathbb{N}} (\mathbb{R})^{m\times n}) \\
[\![w \cdot x + \beta]\!]_\delta(\sigma^D) &= \textbf{let } \ l = \textbf{dim}(w)_1 \textbf{ in} \\
&\quad\ \textbf{let } \ m = \textbf{dim}(w)_2 \textbf{ in} \\
&\quad\ \textbf{let } \ n = \textbf{dim}(\sigma_2^D(x)_1)_2 \textbf{ in} \\
&\quad\ \textbf{let } \ a = [\![w \cdot x + \beta]\!](\sigma_1^D) \textbf{ in} \\
&\quad\ \textbf{let } \ b = \\
&\quad\ \left[\sum_{i=1}^{m} w_{j,i} \cdot (((\sigma_2^D(x))_1)_{i,k}) \ \middle| \ j \in \{1,..,l\},\ k \in \{1,...,n\} \right] \textbf{ in} \\
&\quad\ (a, b)
\end{aligned}
$$

$$
\begin{aligned}
[\![b]\!]_\delta &: \Sigma^D \to \{\mathbf{tt}, \mathbf{ff}\} \\
[\![b]\!]_\delta(\sigma^D) &= [\![b]\!](\sigma_1^D)
\end{aligned}
$$

$$
\begin{aligned}
[\![s^-]\!]_\delta &: \Sigma^D \to \Sigma^D \\
[\![\textbf{skip}]\!]_\delta(\sigma^D) &= \sigma^D \\
[\![y \leftarrow w \cdot x + \beta]\!]_\delta(\sigma^D) &= (\sigma_1^D[y \mapsto ([\![w \cdot x + \beta]\!]_\delta(\sigma^D))_1], \sigma_2^D[y \mapsto (([\![w \cdot x + \beta]\!]_\delta(\sigma^D))_2, \sigma_2^D(x)_2)]) \\
[\![s_1; s_2]\!]_\delta(\sigma^D) &= [\![s_2]\!]_\delta([\![s_1]\!]_\delta(\sigma^D)) \\
[\![\textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2]\!]_\delta(\sigma^D) &= \textbf{if } ([\![b]\!]_\delta(\sigma^D) = \mathbf{tt}) \textbf{ then } [\![s_1]\!]_\delta(\sigma^D) \textbf{ else } [\![s_2]\!]_\delta(\sigma^D)
\end{aligned}
$$

Figure 2.3: *cat* denotational semantics instrumented with Jacobians

We define an instrumented denotational semantics for *cat* (the non-probabilistic sublanguage of *pcat*) in Figure 2.3 that computes Jacobians for a particular program path, in addition to the standard meaning of the program (as defined in Figure 3.2). The semantics are notated by $[\![\cdot]\!]_\delta$ (notice the subscript D). Program states, Σ^D, are pairs of maps such that the first element of each pair belongs to the previously defined set Σ of states, while the second element of each pair is a map that records the Jacobians. The second map is

of type $V \rightarrow ((\bigcup_{m,n\in\mathbb{N}}(\mathbb{R})^{m\times n}) \times V)$, mapping each variable in V to a pair of values, namely, a Jacobian which is matrix of reals, and a variable in V. A cat program can map multiple input vectors to multiple output vectors, so one can compute a Jacobian of the cat program for each output vector with respect to each input vector. This explains the type of the second map in Σ^D - for each variable, the map records the corresponding Jacobian of the cat program computed with respect to the input variable that forms the second element of the pair.

Before explaining the semantics in Figure 2.3, we clarify the notation used in the figure. We use subscript indices, starting from 1, to refer to elements in a pair or a tuple. For instance, we can read $((\sigma_2^D(x))_1)_{i,k}$ in the definition of $[\![w \cdot x + \beta]\!]_\delta$ as follows - σ_2^D refers to the second map of the σ^D pair, $\sigma_2^D(x)_1$ extracts the first element (i.e., the Jacobian matrix) of the pair mapped to variable x, and then finally, we extract the element at location (i, k) in the Jacobian matrix. Also, we use let expressions in a manner similar to ML, and list comprehensions similar to Haskell (though we extend the notation to handle matrices). $\mathbf{dim}$ is polymorphic and returns the dimensions of vectors and matrices.

The only interesting semantic definitions are the ones associated with the expression $w \cdot x + \beta$ and the statement $y \leftarrow w \cdot x + \beta$. The value associated with any variable in a cat program is always of the form, $w_n(w_{n-1}(...(w_2(w_1 \cdot x + \beta_1) + \beta_2)...) + \beta_{n-1}) + \beta_n = w_n \cdot w_{n-1} \cdot ... \cdot w_2 \cdot w_1 \cdot x + w_n \cdot w_{n-1} \cdot ... \cdot w_2 \cdot \beta_1 + w_n \cdot w_{n-1} \cdot ... \cdot w_3 \cdot \beta_2 + ... + \beta_n$. The derivative (the Jacobian) of this term with respect to x is $w_n \cdot w_{n-1} \cdot ... \cdot w_2 \cdot w_1$. Thus, calculating the Jacobian of a cat program for a particular output variable with respect to a particular input variable only requires multiplying the relevant weight matrices together and the bias terms can be ignored. This is exactly how we define the semantics of $w \cdot x + \beta$.

2.3.2 Jacobian Analysis

The abstract version of the instrumented denotational semantics of cat is defined in Figure 2.4. The semantics are notated by $[\![\cdot]\!]_L$ (notice the subscript L). The analysis computes

$$
\begin{array}{rl}
\Sigma^B & \triangleq V \to \bigcup_{n\in\mathbb{N}} (\mathbb{R}\times\mathbb{R})^n \\
\Sigma^L & \triangleq \Sigma^B \times (V \to ((\bigcup_{m,n\in\mathbb{N}} (\mathbb{R}\times\mathbb{R})^{m\times n}) \times (V \cup \{\bot,\top\}))) \\
[\![e]\!]_{_L} & : \Sigma^L \to ((\bigcup_{n\in\mathbb{N}} (\mathbb{R}\times\mathbb{R})^n) \times (\bigcup_{m,n\in\mathbb{N}} (\mathbb{R}\times\mathbb{R})^{m\times n})) \\
[\![w\cdot x+\beta]\!]_{_L}(\sigma^L) = & \textbf{let } \ l = \textbf{dim}(w)_1 \textbf{ in} \\
& \textbf{let } \ m = \textbf{dim}(w)_2 \textbf{ in} \\
& \textbf{let } \ n = \textbf{dim}(\sigma_2^L(x))_2 \textbf{ in} \\
& \textbf{let } \ a = [\![w\cdot x+\beta]\!]_{_B}(\sigma_1^L) \textbf{ in} \\
& \textbf{let } \ b = \\
& \left[\begin{array}{l|l}
\begin{array}{l}
((\sum\limits_{i=1\wedge w_{j,i}\geqslant 0}^{m} w_{j,i}\cdot(((\sigma_2^L(x))_1)_{i,k})_1 + \\
\quad \sum\limits_{i=1\wedge w_{j,i}<0}^{m} w_{j,i}\cdot(((\sigma_2^L(x))_1)_{i,k})_2) \\
(\sum\limits_{i=1\wedge w_{j,i}\geqslant 0}^{m} w_{j,i}\cdot(((\sigma_2^L(x))_1)_{i,k})_2 + \\
\quad \sum\limits_{i=1\wedge w_{j,i}<0}^{m} w_{j,i}\cdot(((\sigma_2^L(x))_1)_{i,k})_1)
\end{array} & j\in\{1,..,l\},\ k\in\{1,...,n\}
\end{array}\right] \textbf{ in} \\
& (a,b)
\end{array}
$$

$$
\begin{array}{rl}
\bigsqcup_L & : \Sigma^L\times\Sigma^L\to\Sigma^L \\
\sigma^L \bigsqcup_L \tilde{\sigma}^L = & ((\sigma_1^L \bigsqcup_B \sigma_2^L), \\
& (\lambda v.\ \textbf{let } \ (m,n) = \textbf{dim}(\sigma_2^L(v)) \textbf{ in} \\
& \quad \textbf{if } (\sigma_2^L(v)_2 = \tilde{\sigma}_2^L(v)_2) \textbf{ then} \\
& \quad ([[(\textbf{min}\{(\sigma_2^L(v)_1)_{i,j}, (\tilde{\sigma}_2^L(v)_1)_{i,j}\}, \textbf{max}\{(\sigma_2^L(v)_1)_{i,j}, (\tilde{\sigma}_2^L(v)_1)_{i,j}\}) \mid \\
& \quad\quad i\in\{1,...,m\},\ j\in\{1,...,n\}]], \sigma_2^L(v)_2) \\
& \quad \textbf{else } ([[(-\infty,\infty) \mid i\in\{1,...,m\},\ j\in\{1,...,n\}]], \top)
\end{array}
$$

$$
\begin{array}{rl}
[\![b]\!]_{_L} & : \Sigma^L \to \{\textbf{tt},\textbf{ff},\top\} \\
[\![b]\!]_{_L}(\sigma^L) = & [\![b]\!]_{_B}(\sigma_1^L)
\end{array}
$$

$$
\begin{array}{rl}
[\![s^-]\!]_{_L} & : \Sigma^L \to \Sigma^L \\
[\![\textbf{skip}]\!]_{_L}(\sigma^L) = & \sigma^L \\
[\![\textbf{assert } b]\!]_{_L}(\sigma^L) = & ((([\![\textbf{assert } b]\!]_{_B}(\sigma_1^L)), \sigma_2^L) \\
[\![y \leftarrow w\cdot x+\beta]\!]_{_L}(\sigma^L) = & (\sigma_1^L[y\mapsto([\![w\cdot x+\beta]\!]_{_L}(\sigma^L))_1], \sigma_2^L[y\mapsto(([\![w\cdot x+\beta]\!]_{_L}(\sigma^L))_2, \sigma_2^L(x)_2)]) \\
[\![s_1; s_2]\!]_{_L}(\sigma^L) = & [\![s_2]\!]_{_L}([\![s_1]\!]_{_L}(\sigma^L)) \\
[\![\textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2]\!]_{_L}(\sigma^L) = & \textbf{if} \quad ([\![b]\!]_{_L}(\sigma^L) = \textbf{tt}) \textbf{ then } [\![s_1]\!]_{_L}(\sigma^L) \\
& \textbf{else if } ([\![b]\!]_{_L}(\sigma^L) = \textbf{ff}) \textbf{ then } [\![s_2]\!]_{_L}(\sigma^L) \\
& \textbf{else} \quad [\![s_1]\!]_{_L}([\![\textbf{assert } b]\!]_{_L}(\sigma^L)) \bigsqcup_L [\![s_2]\!]_{_L}([\![\textbf{assert } \neg b]\!]_{_L}(\sigma^L))
\end{array}
$$

Figure 2.4: *cat* abstract semantics for Jacobian analysis

box-shaped overapproximations of all the possible outcomes of a cat program when executed on inputs from a box-shaped bounded set. This is similar to standard interval analysis except that cat operates on data of base type of real vectors. The analysis maintains bounds on real vectors by computing intervals for every element of a vector. In addition, this analysis also computes an overapproximation of all the possible Jacobian matrices. Note that the Jacobian matrices computed by the instrumented semantics of cat only depend on the path through the program, i.e. the entries in the computed Jacobian are control-dependent on the program inputs but not data-dependent. Consequently, for precision, it is essential that our analysis exhibit some notion of path-sensitivity. We achieve this by evaluating the branch conditions using the computed intervals and abstractly interpreting both the branches of an **if then else** statement only if the branch direction cannot be resolved.

An abstract program state, $\sigma^L \in \Sigma^L$, is a pair of maps. The first map in an abstract state maps variables in V to abstract vectors representing a box-shaped set of vectors. Each element of an abstract vector is pair of reals representing a lower bound and an upper bound on the possible values (first element of the pair is the lower bound and second element is the upper bound). The second map in an abstract state maps variables in V to pairs of abstract Jacobian matrices and elements in V extended with a top and a bottom element. Like abstract vectors, each element of an abstract Jacobian matrix is a pair of reals representing lower and upper bounds of the corresponding partial derivative.

The definition of the abstract semantics is straightforward but we describe the abstract semantics for affine expressions and for conditional statements. First, we discuss affine expressions. As a quick reminder of the notation, a term of the form $(((\sigma_2^L(x))_1)_{i,k})_1$ represents the lower bound of the element at location (i, k) in the abstract Jacobian associated with variable x. Now, recall that the instrumented semantics computes Jacobians simply by multiplying the weight matrices. In the abstract semantics, we multiply abstract Jacobians such that the bounds on each abstract element in the output abstract Jacobian reflect the minimum and maximum possible values that the element could take given the input

abstract Jacobians. The abstract vectors for the first map are computed using the abstract box semantics (notated by $[\![\cdot]\!]_B$), defined in subsection 2.3.3.For conditional statements, as mentioned previously, we first evaluate the branch condition using the abstract state. If this evaluation returns $\top$, meaning that the analysis was unable to discern the branch to be taken, we abstractly interpret both the branches and then join the computed abstract states. Note that before abstractly interpreting both branches, we update the abstract state to reflect that the branch condition should hold before executing s_1 and should not hold before executing s_2. However, the $\mathbf{assert}\ b$ statement is not a part of the cat language, and only used for defining the abstract semantics. The join operation ($\bigcup_L$) is as expected, except for one detail that we want to highlight - in case the Jacobians along different branches are computed with respect to different input variables we make the most conservative choice when joining the abstract Jacobians, bounding each element with $(-\infty, \infty)$ as well as recording $\top$ for the input variable.

Next, we define the concretization function (γ_L) for the abstract program states that maps elements in Σ^L to sets of elements in Σ^D and then state the soundness theorem for our analysis.

Definition 5. *(Concretization function for Jacobian analysis)*

$\gamma_L(\sigma^L) = \{\sigma^D \mid (\bigwedge_{v \in V} .\sigma_1^L(v)_1 \leqslant \sigma_1^D(v) \leqslant \sigma_1^L(v)_2) \wedge (\bigwedge_{v \in V} .(\sigma_2^L(v)_1)_1 \leqslant \sigma_2^D(v)_1 \leqslant (\sigma_2^L(v)_1)_2) \wedge \sigma_2^D(v)_2 \in \gamma_V(\sigma_2^L(v)_2)\}$ *where* $\gamma_V(v) = v$ *and* $\gamma_V(\top) = V$

Theorem 6. *(Soundness of Jacobian analysis)*

$\forall p \in s^-, \sigma^L \in \Sigma^L.\ \{[\![p]\!]_\delta(\sigma^D) \mid \sigma^D \in \gamma_L(\sigma^L)\} \subseteq \gamma_L([\![p]\!]_L(\sigma^L))$

We first prove a lemma needed for the proof.

Lemma 7. *(Soundness of abstract conditional checks)*

$\forall c \in b, \sigma^L \in \Sigma^L.\ \{[\![c]\!]_\delta(\sigma^D) \mid \sigma^D \in \gamma_L(\sigma^L)\} \subseteq \gamma_C([\![c]\!]_L(\sigma^L))$ *where*

$\gamma_C(\mathbf{tt}) = \{\mathbf{tt}\}$, $\gamma_C(\mathbf{ff}) = \{\mathbf{ff}\}$, $\gamma_C(\top) = \{\mathbf{tt}, \mathbf{ff}\}$

Proof. We prove this by induction on the structure of the boolean expressions in b.

We first consider the base cases:

(i) $\underline{\boldsymbol{\pi}(x,m) \geqslant \boldsymbol{\pi}(y,n)}$

By definition, $[\![\boldsymbol{\pi}(x,m) \geqslant \boldsymbol{\pi}(y,n)]\!]_{L}(\sigma^{L}) = [\![\boldsymbol{\pi}(x,m) \geqslant \boldsymbol{\pi}(y,n)]\!]_{B}(\sigma_1^{L})$

Consider the case where, $[\![\boldsymbol{\pi}(x,m) \geqslant \boldsymbol{\pi}(y,n)]\!]_{B}(\sigma_1^{L}) = \mathbf{tt}$, then, by the semantics described in Figure 2.5, we know that,

$$\sigma_1^{L}(x)_m)_1 \geqslant (\sigma_1^{L}(y)_n)_2) \tag{2.1}$$

By the definition of γ_L (Definition 5), we also know that,

$$\forall \sigma^{D} \in \gamma_L.\ (\sigma_1^{L}(x)_1 \leqslant \sigma_1^{D}(x) \leqslant \sigma_1^{L}(x)_2) \wedge (\sigma_1^{L}(y)_1 \leqslant \sigma_1^{D}(y) \leqslant \sigma_1^{L}(y)_2) \tag{2.2}$$

where the comparisons are performed pointwise for every element in the vector.

From Equation 2.1 and Equation 2.2, we can conclude that,

$$\forall \sigma^{D} \in \gamma_L(\sigma^{L}).\ \sigma_1^{D}(y)_n \leqslant (\sigma_1^{L}(y)_n)_2 \leqslant (\sigma_1^{L}(x)_m)_1 \leqslant \sigma_1^{D}(x)_m \tag{2.3}$$

Now,

$$[\![\boldsymbol{\pi}(x,m) \geqslant \boldsymbol{\pi}(y,n)]\!]_{\delta}(\sigma^{D}) = [\![\boldsymbol{\pi}(x,m) \geqslant \boldsymbol{\pi}(y,n)]\!](\sigma_1^{D}) = \\ \mathbf{if}\ \sigma_1^{D}(x)_m \geqslant \sigma_1^{D}(y)_n\ \mathbf{then\ tt\ else\ ff} \tag{2.4}$$

From Equation 2.3 and Equation 2.4, we can conclude that,

$\forall \sigma^{D} \in \gamma_L(\sigma^{L}).\ [\![\boldsymbol{\pi}(x,m) \geqslant \boldsymbol{\pi}(y,n)]\!]_{\delta}(\sigma^{D}) = \mathbf{tt}$, or in other words,

$\{[\![\boldsymbol{\pi}(x,m) \geqslant \boldsymbol{\pi}(y,n)]\!]_{\delta}(\sigma^{D}) \mid \sigma^{D} \in \gamma_L(\sigma^{L})\} \subseteq \gamma_C([\![\boldsymbol{\pi}(x,m) \geqslant \boldsymbol{\pi}(y,n)]\!]_{L}(\sigma^{L}))$ when the analysis returns $\mathbf{tt}$.

We can similarly prove the case when the analysis returns $\mathbf{ff}$. In case, the analysis returns $\top$, the required subset containment is trivially true since $\gamma_C(\top) = \{\mathbf{tt}, \mathbf{ff}\}$.

(ii) $\underline{\boldsymbol{\pi}(x, m) \geqslant 0}$

The proof is very similar to the first case, and we skip the details.

(iii) $\underline{\boldsymbol{\pi}(x, m) < 0}$

The proof is very similar to the first case, and we skip the details.

We next consider the inductive cases:

(iv) $\underline{b_1 \wedge b_2}$

By the inductive hypothesis, we know that,

$\{[\![b_1]\!]_\delta(\sigma^D) \mid \sigma^D \in \gamma_L(\sigma^L)\} \subseteq \gamma_C([\![b_1]\!]_L(\sigma^L))$

$\{[\![b_2]\!]_\delta(\sigma^D) \mid \sigma^D \in \gamma_L(\sigma^L)\} \subseteq \gamma_C([\![b_2]\!]_L(\sigma^L))$

If $[\![b_1]\!]_L(\sigma^L) = \top \vee [\![b_2]\!]_L(\sigma^L) = \top$, then, as per the semantics in Figure 2.5, $[\![b_1 \wedge b_2]\!]_L(\sigma^L) = \top$, and the desired property trivially holds.

However, if $[\![b_1]\!]_L(\sigma^L) \neq \top \wedge [\![b_2]\!]_L(\sigma^L) \neq \top$, then using the inductive hypotheses, we know that for all $\sigma^D \in \gamma_L(\sigma^L)$, $[\![b_1]\!]_\delta(\sigma^D)$ evaluates to the same boolean value as $[\![b_1]\!]_L(\sigma^L)$. We can make the same deduction for b_2. So, evaluating $[\![b_1 \wedge b_2]\!]_\delta$ also yields the same boolean value for all $\sigma^D \in \gamma_L(\sigma^L)$, and this value is equal to $[\![b_1 \wedge b_2]\!]_L(\sigma^L)$.

(v) $\underline{\neg b}$

By the inductive hypothesis, we know that,

$\{[\![b]\!]_\delta(\sigma^D) \mid \sigma^D \in \gamma_L(\sigma^L)\} \subseteq \gamma_C([\![b]\!]_L(\sigma^L))$

If $[\![b]\!]_L(\sigma^L) = \mathbf{tt}$, then $\forall \sigma^D \in \gamma_L(\sigma^L).\ [\![b]\!]_\delta(\sigma^D) = \mathbf{tt}$.

So, $\forall \sigma^D \in \gamma_L(\sigma^L).\ [\![\neg b]\!]_\delta(\sigma^D) = \mathbf{ff}$, and we can conclude that,

$\{[\![\neg b]\!]_\delta(\sigma^D) \mid \sigma^D \in \gamma_L(\sigma^L)\} \subseteq \gamma_C([\![\neg b]\!]_L(\sigma^L)) = \{\mathbf{ff}\}$.

We can similarly argue about the case when $[\![b]\!]_{L}(\sigma^{L}) = \mathbf{ff}$, and as stated previously, the case with, $[\![b]\!]_{L}(\sigma^{L}) = \top$ trivially holds.

■

Theorem 6. *(Soundness of Jacobian analysis)*

$\forall p \in s^{-}, \sigma^{L} \in \Sigma^{L}.\ \{[\![p]\!]_{\delta}(\sigma^{D}) \mid \sigma^{D} \in \gamma_{L}(\sigma^{L})\} \subseteq \gamma_{L}([\![p]\!]_{L}(\sigma^{L}))$

Proof. We prove this by induction on the structure of statements in s^{-}.

We first consider the base cases:

(i) $\underline{\mathbf{skip}}$

By definition, for any state σ^{L},

$$[\![\mathbf{skip}]\!]_{L}(\sigma^{L}) = \sigma^{L} \tag{2.5}$$

$$\{[\![\mathbf{skip}]\!]_{\delta}(\sigma^{D}) \mid \sigma^{D} \in \gamma_{L}(\sigma^{L})\} = \{\sigma^{D} \mid \sigma^{D} \in \gamma_{L}(\sigma^{L})\} = \gamma_{L}(\sigma^{L}) \tag{2.6}$$

From Equations Equation 2.5 and Equation 2.6,

$$\{[\![\mathbf{skip}]\!]_{\delta}(\sigma^{D}) \mid \sigma^{D} \in \gamma_{L}(\sigma^{L})\} \subseteq \gamma_{L}([\![\mathbf{skip}]\!]_{L}(\sigma^{L})) \tag{2.7}$$

(ii) $\underline{y \leftarrow w \cdot x + \beta}$

We first observe that when multiplying an interval (l, u) with a constant c, if $c \geqslant 0$, then the result is simply given by the interval $(c \cdot l, c \cdot u)$. But if $c < 0$, then the result is in the interval $(c \cdot u, c \cdot l)$, i.e., the use of the lower bounds and upper bounds gets flipped. Similarly, when computing the dot product of an abstract vector v with a

constant vector w, for each multiplication operation $v_i \cdot w_i$, we use the same reasoning as above. Then, the lower bound and upper bound of the dot product result are given by,

$(\sum_{i=1 \wedge w_i \geqslant 0}^{n} w_i \cdot (v_i)_1 + \sum_{i=1 \wedge w_i < 0}^{n} w_i \cdot (v_i)_2, \sum_{i=1 \wedge w_i \geqslant 0}^{n} w_i \cdot (v_i)_2 + \sum_{i=1 \wedge w_i < 0}^{n} w_i \cdot (v_i)_1)$

where $(v_i)_1$ represents the lower bound of the i^{th} element of v and $(v_i)_2$ represents the lower bound of the i^{th} element of v, and we assume $\mathbf{dim}(w) = \mathbf{dim}(v) = n$.

We do not provide the rest of the formal proof for this case since it just involves using the definitions.

Next, we consider the inductive cases:

(iii) $\underline{s_1^-; s_2^-}$

From the inductive hypothesis, we know,

$$L_1 = \{[\![s_1^-]\!]_\delta(\sigma^D) \mid \sigma^D \in \gamma_L(\sigma^L)\} \subseteq \gamma_L([\![s_1^-]\!]_L(\sigma^L)) \tag{2.8}$$

$$L_2 = \{[\![s_2^-]\!]_\delta(\sigma^D) \mid \sigma^D \in \gamma_L([\![s_1^-]\!]_L(\sigma^L))\} \subseteq \gamma_L([\![s_2^-]\!]_L([\![s_1^-]\!]_L(\sigma^L))) \tag{2.9}$$

From Equations Equation 2.8 and Equation 2.9, we conclude,

$$\{[\![s_2^-]\!]_\delta(\sigma^D) \mid \sigma^D \in L_1\} \subseteq L_2 \subseteq \gamma_L([\![s_2^-]\!]_L([\![s_1^-]\!]_L(\sigma^L))) \tag{2.10}$$

Rewriting, we get,

$$\{[\![s_2^-]\!]_\delta([\![s_1^-]\!]_\delta(\sigma^D)) \mid \sigma^D \in \gamma_L(\sigma^L)\} \subseteq \gamma_L([\![s_2^-]\!]_L([\![s_1^-]\!]_L(\sigma^L))) \tag{2.11}$$

and this can be simplified further as,

$$\{[\![s_1^-; s_2^-]\!]_\delta(\sigma^D) \mid \sigma^D \in \gamma_L(\sigma^L)\} \subseteq \gamma_L([\![s_1^-; s_2^-]\!]_L(\sigma^L)) \tag{2.12}$$

(iv) $\underline{\textbf{if } b \textbf{ then } s_1^- \textbf{ else } s_2^-}$

From the inductive hypothesis, we know,

$$\{[\![s_1^-]\!]_\delta(\sigma^D) \mid \sigma^D \in \gamma_L(\sigma^L)\} \subseteq \gamma_L([\![s_1^-]\!]_L(\sigma^L)) \tag{2.13}$$

$$\{[\![s_2^-]\!]_\delta(\sigma^D) \mid \sigma^D \in \gamma_L(\sigma^L)\} \subseteq \gamma_L([\![s_2^-]\!]_L(\sigma^L)) \tag{2.14}$$

The conditional check can result in three different outcomes while performing the analysis - $\mathbf{tt}$, $\mathbf{ff}$, or $\top$. From Lemma 7, we know that the abstract boolean checks are sound. We analyze each of the three cases separately.

(a) $\underline{\mathbf{tt}}$

Since we only consider the true case, we can write,

$$[\![\textbf{if } b \textbf{ then } s_1^- \textbf{ else } s_2^-]\!]_L(\sigma^L) = [\![s_1^-]\!]_L(\sigma^L) \tag{2.15}$$

Also, from Lemma 7,

$$\{[\![\textbf{if } b \textbf{ then } s_1^- \textbf{ else } s_2^-]\!]_\delta(\sigma^D) \mid \sigma^D \in \gamma_L(\sigma^L)\} = \{[\![s_1^-]\!]_\delta(\sigma^D) \mid \sigma^D \in \gamma_L(\sigma^L)\} \tag{2.16}$$

From Equation 2.13, Equation 2.15, and Equation 2.16,

$$\{[\![\textbf{if } b \textbf{ then } s_1^- \textbf{ else } s_2^-]\!]_\delta(\sigma^D) \mid \sigma^D \in \gamma_L(\sigma^L)\} \subseteq \gamma_L([\![\textbf{if } b \textbf{ then } s_1^- \textbf{ else } s_2^-]\!]_L(\sigma^L)) \tag{2.17}$$

(b) $\underline{\mathbf{ff}}$

Similar to the $\mathbf{tt}$ case, for the $\mathbf{ff}$ case, we can show,

$$\{[\![\textbf{if } b \textbf{ then } s_1^- \textbf{ else } s_2^-]\!]_\delta(\sigma^D) \mid \sigma^D \in \gamma_L(\sigma^L)\} \subseteq \gamma_L([\![\textbf{if } b \textbf{ then } s_1^- \textbf{ else } s_2^-]\!]_L(\sigma^L)) \tag{2.18}$$

(c) $\top$

We first prove the following about the join ($\bigsqcup_L$) operation,

$$\gamma_L(\sigma^L) \cup \gamma_L(\tilde{\sigma}^L) \subseteq \gamma_L(\sigma^L \sqcup_L \tilde{\sigma}^L) \tag{2.19}$$

By definition of γ_L,

$$\begin{aligned}\gamma_L(\sigma^L) = \{\sigma^D \mid (\bigwedge_{v \in V}.\sigma_1^L(v)_1 \leqslant \sigma_1^D(v) \leqslant \sigma_1^L(v)_2) \wedge \\ (\bigwedge_{v \in V}.(\sigma_2^L(v)_1)_1 \leqslant \sigma_2^D(v)_1 \leqslant (\sigma_2^L(v)_1)_2) \wedge \\ \sigma_2^D(v)_2 \in \gamma_V(\sigma_2^L(v)_2)\}\end{aligned} \tag{2.20}$$

$\gamma_L(\tilde{\sigma}^L)$ can be defined similarly.

The join operation combines corresponding intervals in the abstract states by taking the smaller of the two lower bounds and larger of the two upper bounds. We do not prove the following formally, but from the definition of γ_L and $\bigsqcup_L$, one can see that the intended property holds.

Next, we consider the **assert** statements that appear in the abstract denotational semantics for the $\top$ case.

Let us call, $\sigma_1^L = [\![\mathbf{assert}\ b]\!]_L(\sigma^L)$ and $\sigma_2^L = [\![\mathbf{assert}\ \neg b]\!]_L(\sigma^L)$.

From inductive hypothesis (Equation 2.13 and Equation 2.14) we know,

$$L_1 = \{[\![s_1^-]\!]_\delta(\sigma^D) \mid \sigma^D \in \gamma_L(\sigma_1^L)\} \subseteq \gamma_L([\![s_1^-]\!]_L(\sigma_1^L)) \tag{2.21}$$

$$L_2 = \{[\![s_2^-]\!]_\delta(\sigma^D) \mid \sigma^D \in \gamma_L(\sigma_2^L)\} \subseteq \gamma_L([\![s_2^-]\!]_L(\sigma_2^L)) \tag{2.22}$$

From Equation 2.19,Equation 2.21, and Equation 2.22,

$$L_1 \cup L_2 \subseteq \gamma_L([\![s_1^-]\!]_L(\sigma_1^L)) \cup \gamma_L([\![s_2^-]\!]_L(\sigma_2^L)) \subseteq \gamma_L([\![s_1^-]\!]_L(\sigma_1^L) \sqcup [\![s_2^-]\!]_L(\sigma_2^L)) \quad (2.23)$$

Then, if we can show that,

$$\{\sigma^D \mid \sigma^D \in \gamma_L(\sigma^L) \wedge [\![b]\!](\sigma^D) = \mathbf{tt}\} \subseteq \gamma_L(\sigma_1^L) \quad (2.24)$$

$$\{\sigma^D \mid \sigma^D \in \gamma_L(\sigma^L) \wedge [\![b]\!](\sigma^D) = \mathbf{ff}\} \subseteq \gamma_L(\sigma_2^L) \quad (2.25)$$

then, from Equation 2.21, Equation 2.22,Equation 2.23,Equation 2.24,Equation 2.25, and the semantics of $\mathbf{if}\ b\ \mathbf{then}\ s_1^-\ \mathbf{else}\ s_2^-$, we can say,

$$\{[\![\mathbf{if}\ b\ \mathbf{then}\ s_1^-\ \mathbf{else}\ s_2^-]\!]_\delta(\sigma^D) \mid \sigma^D \in \gamma_L(\sigma^L)\} \subseteq \gamma_L([\![\mathbf{if}\ b\ \mathbf{then}\ s_1^-\ \mathbf{else}\ s_2^-]\!]_L(\sigma^L)) \quad (2.26)$$

Now, we need to show that Equation 2.24 and Equation 2.25 are true. The $\mathbf{assert}$ statements either behave as identity or produce a modified abstract state (see Figure 2.5). When $\mathbf{assert}$ behaves as identity, Equation 2.24 and Equation 2.25 are obviously true. We skip the proof of the case when $\mathbf{assert}$ produces a modified abstract state.

■

We next define the notion of operator norm of an abstract Jacobian. This definition is useful for stating Corollary 9. Given an abstract Jacobian, we construct a matrix J such every element of J is the maximum of the absolute values of the corresponding lower and upper bound in the abstract Jacobian.

Definition 8. *(Operator norm of abstract Jacobian)*

If $J = \sigma_2^L(v)_1$ *for some* σ^L *and* v*, and* $(m, n) = \mathbf{dim}(J)$ *then* $\|J\|_{_L}$ *is defined as,* $\|J\|_{_L} = \|[\mathbf{max}\{|(J_{k,l})_1|, |(J_{k,l})_2|\} \mid k \in \{1, ..., m\}, l \in \{1, ..., n\}]\|$

Corollary 9 shows that the operator norm of the abstract Jacobian computed by the analysis for some variable v is an upper bound of the operator norms of the all the Jacobians possible for v when a program p is executed on the set of inputs represented by $\gamma_L(\sigma^L)$, for any program p and any abstract state σ^L.

Corollary 9. *(Upper bound of Jacobian operator norm)*

$\forall p \in s^-, \sigma^L \in \Sigma^L, v \in V.$

$\mathbf{max}\{\left\|(([\![p]\!]_\delta(\sigma^D))_2)(v)_1\right\| \mid \sigma^D \in \gamma_L(\sigma^L)\} \leqslant \|(([\![p]\!]_{_L}(\sigma^L))_2(v))_1\|_{_L}$

Proof. From Theorem 6, we know that for any $p \in s^-, \sigma^L \in \Sigma^L$,

$$\{[\![p]\!]_\delta(\sigma^D) \mid \sigma^D \in \gamma_L(\sigma^L)\} \subseteq \gamma_L([\![p]\!]_{_L}(\sigma^L)) \tag{2.27}$$

Let us define, $D_V = \{(([\![p]\!]_\delta(\sigma^D))_2(v))_1 \mid \sigma^D \in \gamma_L(\sigma^L)\}$. This is the set of all Jacobian matrices associated with the variable v after executing p on the set of input states, $\gamma_L(\sigma^L)$. Note that the set D_V does not distinguish the Jacobians on the basis of the input that we are differentiating with respect to.

Let $D_V^L = \{(\tilde{\sigma}_2^D(v))_1 \mid \tilde{\sigma}^D \in \gamma_L([\![p]\!]_{_L}(\sigma^L))\}$, and $J = (([\![p]\!]_{_L}(\sigma^L))_2(v))_1$.

Using Definition 5 of γ_L, we can show,

$$\forall d \in D_V^L.\ J_1 \leqslant d \leqslant J_2 \tag{2.28}$$

where $\leqslant$ is defined pointwise on the matrices, and $J_1(J_2)$ refers to the matrix of lower(upper) bounds.

Then, from Equation 2.27 and definitions of D_V and D_V^L, we can deduce that,

$$D_V \subseteq D_V^L \tag{2.29}$$

From Equation 2.28 and Equation 2.29,

$$\forall d \in D_V.\ J_1 \leqslant d \leqslant J_2 \tag{2.30}$$

Let $J' = [\mathbf{max}\{|(J_{k,l})_1|, |(J_{k,l})_2|\} \mid k \in \{1, ..., m\}, l \in \{1, ..., n\}]$. Then,

$$\forall d \in D_V.\ |d| \leqslant J' \tag{2.31}$$

where $|\cdot|$ applies pointwise on matrices d.

Using definition of operator norm, one can show that,

$$M_1 \leqslant M_2 \implies \|M_1\| \leqslant \|M_2\| \tag{2.32}$$

where M_1 and M_2 are matrices with $\leqslant$ applied pointwise.

Finally, from Equation 2.31 and Equation 2.32, we conclude,

$$\forall d \in D_V.\ \|d\| \leqslant \|J'\| = \|J\| \tag{2.33}$$

Unrolling the definitions,

$$\mathbf{max}\{\left|\left|((\llbracket p \rrbracket_\delta(\sigma^D))_2)(v)_1\right|\right| \mid \sigma^D \in \gamma(\sigma^L)\} \leqslant \|((\llbracket p \rrbracket_L(\sigma^L))_2(v))_1\|_L \tag{2.34}$$

■

2.3.3 Box Analysis

$$
\begin{aligned}
\Sigma^B &\triangleq V \to \textstyle\bigcup_{n\in\mathbb{N}} (\mathbb{R}\times\mathbb{R})^n \\
[\![e]\!]_B &: \Sigma^B \to \textstyle\bigcup_{n\in\mathbb{N}} (\mathbb{R}\times\mathbb{R})^n \\
[\![\boldsymbol{\pi}(x,n)]\!]_B(\sigma^B) &= \sigma^B(x)_n \\
[\![w\cdot x+\beta]\!]_B(\sigma^B) &= \textbf{let } \ m=\mathbf{dim}(w)_1 \textbf{ in} \\
&\quad \textbf{let } \ n=\mathbf{dim}(\sigma^B(x)) \textbf{ in} \\
&\quad [((\sum_{j=1\wedge w_{i,j}\geqslant 0}^{n} w_{i,j}\cdot(\sigma^B(x)_i)_1 + \\
&\quad \sum_{j=1\wedge w_{i,j}<0}^{n} w_{i,j}\cdot(\sigma^B(x)_i)_2+\beta_i), \\
&\quad (\sum_{j=1\wedge w_{i,j}\geqslant 0}^{n} w_{i,j}\cdot(\sigma^B(x)_i)_2 + \\
&\quad \sum_{j=1\wedge w_{i,j}<0}^{n} w_{i,j}\cdot(\sigma^B(x)_i)_1+\beta_i)) \mid i\in\{1,...,m\}]
\end{aligned}
$$

$$
\begin{aligned}
[\![b]\!]_B &: \Sigma^B \to \{\mathbf{tt},\mathbf{ff},\top\} \\
[\![\boldsymbol{\pi}(x,m)\geqslant\boldsymbol{\pi}(y,n)]\!]_B(\sigma^B) &= \textbf{if} \quad ((\sigma^B(x)_m)_1 \geqslant (\sigma^B(y)_n)_2) \textbf{ then } \mathbf{tt} \\
&\quad \textbf{else if } ((\sigma^B(x)_m)_2 < (\sigma^B(y)_n)_1) \textbf{ then } \mathbf{ff} \\
&\quad \textbf{else} \quad \top \\
[\![\boldsymbol{\pi}(x,m)\geqslant 0]\!]_B(\sigma^B) &= \textbf{if} \quad ((\sigma^B(x)_m)_1 \geqslant 0) \textbf{ then } \mathbf{tt} \\
&\quad \textbf{else if } ((\sigma^B(x)_m)_2 < 0) \textbf{ then } \mathbf{ff} \\
&\quad \textbf{else} \quad \top \\
[\![\boldsymbol{\pi}(x,m)< 0]\!]_B(\sigma^B) &= \textbf{if} \quad ((\sigma^B(x)_m)_2 < 0) \textbf{ then } \mathbf{tt} \\
&\quad \textbf{else if } ((\sigma^B(x)_m)_1 \geqslant 0) \textbf{ then } \mathbf{ff} \\
&\quad \textbf{else} \quad \top \\
[\![b_1\wedge b_2]\!]_B(\sigma^B) &= \textbf{if} \quad ([\![b_1]\!]_B(\sigma^B)=\top \vee [\![b_2]\!]_B(\sigma^B)=\top) \textbf{ then } \top \\
&\quad \textbf{else } [\![b_1]\!]_B(\sigma^B) \wedge [\![b_2]\!]_B(\sigma^B) \\
[\![\neg b]\!]_B(\sigma^B) &= \textbf{if} \quad ([\![b]\!]_B(\sigma^B)=\mathbf{tt}) \textbf{ then } \mathbf{ff} \\
&\quad \textbf{else if } ([\![b]\!]_B(\sigma^B)=\mathbf{ff}) \textbf{ then } \mathbf{tt} \\
&\quad \textbf{else } \top
\end{aligned}
$$

$$
\begin{aligned}
\textstyle\bigsqcup_B &: \Sigma^B\times\Sigma^B \to \Sigma^B \\
\sigma^B \textstyle\bigsqcup_B \tilde{\sigma}^B &= \lambda v.\ [(\mathbf{min}\{(\sigma^B(v)_i)_1,(\tilde{\sigma}^B(v)_i)_1\},\mathbf{max}\{(\sigma^B(v)_i)_2,(\tilde{\sigma}^B(v)_i)_2\}) \mid \\
&\qquad i\in\{1,...,\mathbf{dim}(\sigma^B(v))\}]
\end{aligned}
$$

$$
\begin{aligned}
[\![s^-]\!]_B &: \Sigma^B \to \Sigma^B \\
[\![\mathbf{skip}]\!]_B(\sigma^B) &= \sigma^B \\
[\![\mathbf{assert}\ \boldsymbol{\pi}(x,m)\geqslant 0]\!]_B(\sigma^B) &= \sigma^B[x_m \mapsto (0,\mathbf{max}\{(\sigma^B(x)_m)_2,0\})] \\
[\![\mathbf{assert}\ \boldsymbol{\pi}(x,m)< 0]\!]_B(\sigma^B) &= \sigma^B[x_m \mapsto (\mathbf{min}\{(\sigma^B(x)_m)_1,0\},0)] \\
[\![\mathbf{assert}\ \neg(\boldsymbol{\pi}(x,m)\geqslant 0)]\!]_B(\sigma^B) &= [\![\mathbf{assert}\ \boldsymbol{\pi}(x,m)< 0]\!]_B(\sigma^B) \\
[\![\mathbf{assert}\ \neg(\boldsymbol{\pi}(x,m)< 0)]\!]_B(\sigma^B) &= [\![\mathbf{assert}\ \boldsymbol{\pi}(x,m)\geqslant 0]\!]_B(\sigma^B) \\
[\![\mathbf{assert}\ \hat{b}]\!]_B(\sigma^B) &= \sigma^B(\text{where } \hat{b} \text{ refers to all other boolean expressions}) \\
[\![y\leftarrow w\cdot x+\beta]\!]_B(\sigma^B) &= \sigma^B[y\mapsto [\![w\cdot x+\beta]\!]_B(\sigma^B)] \\
[\![s_1;s_2]\!]_B(\sigma^B) &= [\![s_2]\!]_B([\![s_1]\!]_B(\sigma^B)) \\
[\![\mathbf{if}\ b\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2]\!]_B(\sigma^B) &= \textbf{if} \quad ([\![b]\!]_B(\sigma^B)=\mathbf{tt}) \textbf{ then } [\![s_1]\!]_B(\sigma^B) \\
&\quad \textbf{else if } ([\![b]\!]_B(\sigma^B)=\mathbf{ff}) \textbf{ then } [\![s_2]\!]_B(\sigma^B) \\
&\quad \textbf{else} \quad [\![s_1]\!]_B([\![\mathbf{assert}\ b]\!]_B(\sigma^B)) \textstyle\bigsqcup_B [\![s_2]\!]_B([\![\mathbf{assert}\ \neg b]\!]_B(\sigma^B))
\end{aligned}
$$

Figure 2.5: *cat* abstract semantics for box analysis

The box analysis abstracts the lowered cat semantics instead of the instrumented semantics. Given a box-shaped set of input states, it computes box-shaped overapproximations of the program output in a manner similar to the Jacobian analysis. In fact, the box analysis only differs from the Jacobian analysis in not computing abstract Jacobians. We define a separate box analysis to avoid computing abstract Jacobians when not needed. The concretization function (γ_B) for the box analysis and the soundness theorem are stated below. However, we do not provide a separate proof of soundness for the box analysis since such a proof is straightforward given the soundness proof for the Jacobian analysis.

Definition 10. *(Concretization function for box analysis)*
$\gamma_B(\sigma^B) = \{\sigma \mid \bigwedge_{v \in V} . \sigma^B(v)_1 \leqslant \sigma(v) \leqslant \sigma^B(v)_2\}$

Theorem 11. *(Soundness of box analysis)*
$\forall p \in s^-, \sigma^B \in \Sigma^B. \{\widetilde{[\![p]\!]}(\sigma) | \sigma \in \gamma_B(\sigma^B)\} \subseteq \gamma_B([\![p]\!]_B(\sigma^B))$

2.4 Algorithms

We now describe our proof-search algorithms for probabilistic Lipschitzness of `NNs`. First, in subsection 2.4.1, we present the sketch of a randomized proof-search algorithm that is prohibitively expensive for practical use and can only provide statistical guarantees of probabilistic robustness. Next, we describe the `PROLIP` algorithm (subsection 2.4.2), an algorithmic primitive that can be used by a proof-search algorithm for probabilistic Lipschitzness. Finally, we provide the sketch of a proof-search algorithm that uses `PROLIP` in subsection 2.4.3.

2.4.1 A Randomized Algorithm

Using algorithm 1, we sketch a procedure for checking the probabilistic robustness of a neural network `NN`. `NN` is input to the algorithm and is expressed in the form of a cat function. The other inputs to the algorithm are the probabilistic bound ϵ, the Lipschitz

Algorithm 1: Randomized verification algorithm.

Input: NN: Neural network as a cat function.
D: Input distribution.
ϵ: Probabilistic error bound.
k: Lipschitz constant.
Output: $\{\mathbf{T}, \mathbf{F}\}$

```
pf := ConstructProduct(NN);
φ := ¬(‖fx' − fx‖ ⩽ k ∗ ‖x' − x‖));
poly := AbstractInterpret(pf, φ);
err := 0;
foreach p ∈ poly do
    e := sample(p, pf, φ, D);
    err := err + e;
end foreach
if err > ϵ then
    return F;
else
    return T;
```

constant k, and the input distribution D. D can either be represented as a closed form function or as a $pcat$ program but we leave this unspecified here. The algorithm outputs $\mathbf{T}$ (true) if NN satisfies probabilistic robustness, and $\mathbf{F}$ (false) otherwise.

Our algorithm frames the problem of checking the probabilistic robustness of a neural network as a relational program verification problem [65]. Relational verification is defined as checking program properties or specifications that are expressed over pairs of program traces. For instance, probabilistic robustness requires comparing the outputs ($\|fx' - fx\|$) generated by a neural network for pairs of inputs ($\|x' - x\|$). Such two-trace properties are also called *hyperproperties* [66].

A majority of program verification and analysis techniques are only applicable to single-trace properties. To be able to use such techniques for checking hyperproperties, a standard trick used in program verification is to construct a product program [67]. For a program P, a product program is constructed by creating a copy P' of P, where all the variables are renamed, and composing P and P' together to get program $P; P'$. A hyperproperty of the original program then corresponds to a single-trace property of the product program.

The first step of our algorithm is to construct a "product" neural network pf (line 1) by encoding two copies of the original network `NN` side by side. Assume that the input and the output of the original neural network `NN` are notated as $\bar{x}$ and $\bar{y}$, respectively. Then, intuitively, the product neural network (1) accepts the input $(\bar{x}, \bar{x}')$, (2) independently processes $\bar{x}$ and $\bar{x}'$, and (3) produces the output $(\bar{y}, \bar{y}')$, such that $\bar{y}$=`NN`$(\bar{x})$ and $\bar{y}'$=`NN`$(\bar{x}')$. This product construction enables us to use standard abstract interpretation techniques for checking a hyperproperty such as robustness. Note that, as we just discussed, any input for the product neural network represents a pair of inputs for the original neural network. In the rest of this subsection, we therefore use the term input to refer to a product neural network input.

In line 2, the algorithm assigns the temporary name ϕ to the property to be checked, i.e., the negation of the Lipschitz property. The backwards abstract interpreter `AbstractInterpret` produces the set $poly$ (line 3) as an overapproximation of the set of inputs that satisfy ϕ. Since ϕ is the negation of the Lipschitz property, all the inputs NOT in $poly$ satisfy the Lipschitz property. We assume that the set $poly$ denotes a set of disjoint polyhedra in the high-dimensional input space. Accordingly, `AbstractInterpret` is based on the powerset polyhedra abstract domain [68, 69], using sets of disjoint polyhedra to approximate sets of real-valued vectors. Computing $poly$ requires encoding ϕ as an element of the powerset polyhedra domain. The encoded representation ϕ needs an exponential (in the size of the input/output dimensions) number of polyhedra to denote the same set of real-valued vectors as ϕ. This exponential blow-up causes the algorithm to be too expensive for practical use.

Next, for each input polyhedron p in $poly$, the algorithm computes the probability e that a randomly sampled input is within p and satisfies ϕ (membership in p does not imply ϕ since p is an over-approximation of the region of inputs satisfying ϕ). This probability can be upper-bounded by computing the volume of p weighted by the probability distribution D. However, approximately computing the volume of a polyhedron weighted by a simple Gaussian distribution is already expensive [70]. Consequently, in the randomized algo-

rithm presented here, we instead use a sampling procedure to estimate this probability (line 6). Note that, estimating the probability instead of computing it exactly implies that algorithm 1 can only provide a statistical guarantee about the probabilistic robustness of a neural network. In other words, the algorithm is only capable of proving statements of the form, "with a high probability, the neural network is probabilistically robust" or "with a high probability, the neural network is not probabilistically robust". The sampling procedure is based on the importance sampling technique [71]. First, samples are drawn uniformly from the region p. For each sample, the sampling procedure checks if the distance between the two elements comprising the sample input is more than r. If so, the sample is rejected. Otherwise, the sample is accepted. For each accepted sample, the sampling procedure next checks if the sample satisfies ϕ. The probability estimate e is the sum of the likelihood ratios (or weights) of the samples satisfying ϕ divided by the number of samples drawn. The likelihood ratio depends on the Euclidean volume of p and on the density function of the input distribution D.

Finally, after processing all polyhedra, the algorithm checks the value of err, which is the total probability of satisfying ϕ. If err is greater than ϵ, the probability of violating the Lipschitz property is greater than ϵ, neural network `NN` is not probabilistically robust, and the algorithm returns $\mathbf{F}$ (lines 9–10). Otherwise, `NN` satisfies the property, and the algorithm returns $\mathbf{T}$ (lines 11–12).

This algorithm is impractical and undesirable for the following reasons: (1) exponential (in the number of dimensions) blow-up in the abstract representation of ϕ, causing the backwards abstract interpreter to be exponentially expensive; (2) the complexity of the sampling procedure; (3) the inability to provide non-statistical guarantees about probabilistic robustness of a neural network.

2.4.2 PROLIP Algorithmic Primitive

The PROLIP algorithm expects a $pcat$ program p of the form $z \leftsquigarrow N(0,1); g; f$ as input, where g and f are cat programs. $z \leftsquigarrow N(0,1); g$ represents the generative model and f represents the NN under analysis. Other inputs expected by PROLIP are a box-shaped region z_B in z and the input variable as well as the output variable of f ($\mathbf{in}$ and $\mathbf{out}$ respectively). Typically, NNs consume a single input and produce a single output. The outputs produced by PROLIP are (i) k_U, an upper bound on the local Lipschitzness constant of f in a box-shaped region of $\mathbf{in}$ (say $\mathbf{in}_B$) that overapproximates the set of $\mathbf{in}$ values in the image of z_B under g, (ii) d, the maximum distance between $\mathbf{in}$ values in $\mathbf{in}_B$, (iii) vol, the probabilistic volume of the region $z_B \times z_B$ with respect to the distribution $N(0,1) \times N(0,1)$.

PROLIP starts by constructing an initial abstract program state (σ^B) suitable for the box analysis (line 1). σ^B maps every variable in V to abstract vectors with elements in the interval $(-\infty, \infty)$. We assume that for the variables accessed in p, the length of the abstract vectors is known, and for the remaining variables we just assume vectors of length one in this initial state. Next, the initial entry in σ^B for z is replaced by z_B, and this updated abstract state is used to perform box analysis of g, producing $\tilde{\sigma}^B$ as the result (line 2). Next, $\tilde{\sigma}^B$ is used to create the initial abstract state σ^L for the Jacobian analysis of f (line 3).

Algorithm 2: `PROLIP` algorithmic primitive

Input:

p: $pcat$ program.

z_B: Box in z.

$\mathbf{in}$: Input variable of f.

$\mathbf{out}$: Output variable of f.

Output:

k_U: Lipschitz constant.

d: Max $\mathbf{in}$ distance.

vol: Mass of $z_B \times z_B$.

1 $\sigma^B := \lambda v.(-\infty, \infty)$;

2 $\tilde{\sigma}^B := [\![g]\!]_B(\sigma^B[z \mapsto z_B])$;

3 $\sigma^L := (\tilde{\sigma}^B, \lambda v.(I, v))$;

4 $\tilde{\sigma}^L := [\![f]\!]_L(\sigma^L)$;

5 **if** $(\tilde{\sigma}_2^L(\mathbf{out})_2 = \mathbf{in})$ **then**

6 $\quad J := \tilde{\sigma}_2^L(\mathbf{out})_1$;

7 $\quad k_U := \|J\|_L$;

8 **else**

9 $\quad k_U := \infty$;

10 $d :=$ `DIAG_LEN`$(\tilde{\sigma}^B(\mathbf{in}))$;

11 $vol :=$ `VOL`$(N \times N, z_B \times z_B)$;

12 **return** (k_U, d, vol);

Initially, every variable is mapped to an identity matrix as the Jacobian and itself as the variable with respect to which the Jacobian is computed. The initial Jacobian is a square matrix with side length same as that of the abstract vector associated with the variable being mapped. Next, we use σ^L to perform Jacobian analysis of f producing $\tilde{\sigma}^L$ as the result (line 4). If the abstract Jacobian mapped to $\mathbf{out}$ in $\tilde{\sigma}^L$ is computed with respect to $\mathbf{in}$ (line 5),

we proceed down the true branch else we assume that nothing is known about the required Jacobian and set k_U to ∞ (line 9). In the true branch, we first extract the abstract Jacobian and store it in J (line 6). Next, we compute the operator norm of the abstract Jacobian J using Definition 8, giving us the required upper bound on the Lipschitz constant (line 7). We then compute the maximum distance between $\mathbf{in}$ values in the box described by $\tilde{\sigma}^B(\mathbf{in})$ using the procedure `DIAG_LEN` that just computes the length of the diagonal of the hyperrectangle represented by $\tilde{\sigma}^B(\mathbf{in})$ (line 10). We also compute the probabilistic mass of region $z_B \times z_B$ with respect to the distribution $N(0,1) \times N(0,1)$ (line 11). This is an easy computation since we can form an analytical expression and just plug in the boundaries of z_B. Finally, we return the tuple (k_U, d, vol) (line 12). This `PROLIP` algorithm is correct as stated by the following theorem.

Theorem 12. *(Soundness of* `PROLIP`*)*

Let $p = z \leftsquigarrow N(0,1); g; f$ *where* $g, f \in s^-$, $(k_U, d, vol) = \texttt{PROLIP}(p, z_B)$, $z \notin \mathbf{outv}(g), z \notin \mathbf{outv}(f), x \in \mathbf{inv}(f)$, *and* $y \in \mathbf{outv}(f)$ *then,* $\forall \sigma_0 \in \Sigma$.

$$\Pr_{\sigma,\sigma' \sim [\![p]\!](\sigma_0)} ((\|\sigma(y) - \sigma'(y)\| \leqslant k_U \cdot \|\sigma(x) - \sigma'(x)\|) \wedge (\sigma(z), \sigma'(z) \in \gamma(z_B))) \geqslant vol$$

Proof. We prove this theorem in two parts.

First, let us define set Σ_P as, $\Sigma_P = \{\sigma \mid \sigma \in \gamma_B([\![f]\!]_L([\![g]\!]_B(\sigma^B[z \mapsto z_B])))_1)\}$

In words, Σ_P is the concretization of the abstract box produced by abstractly "interpreting" $g; f$ on the input box z_B. Assuming that z is not written to by g or f, it is easy to see from the definitions of the abstract semantics in Figure 2.5 and Figure 2.4 that, $([\![f]\!]_L([\![g]\!]_B(\sigma^B[z \mapsto z_B])))_1(z) = z_B$, i.e., the final abstract value of z is the same as the initial value z_B. Moreover, from Corollary 9, we know that the operator norm of the abstract Jacobian matrix, $\|J\|_L$ upper bounds the operator norm of every Jacobian of f for variable y with respect to x (since $x \in \mathbf{inv}(f), y \in \mathbf{outv}(f)$) for every input in $\gamma_B([\![g]\!]_B(\sigma^B[z \mapsto z_B]))$, which itself is an upper bound on the local Lipschitz constant in the same region.

In other words, we can say that,

$\forall \sigma, \sigma' \in \Sigma_P.\ \sigma(z), \sigma'(z) \in \gamma(z_B) \wedge \|\sigma(y) - \sigma'(y)\| \leqslant k_U \cdot \|\sigma(x) - \sigma'(x)\|$.

To complete the proof, we need to show that, $\Pr_{\sigma,\sigma' \sim [\![p]\!](\sigma_0)} (\sigma, \sigma' \in \Sigma_P) \geqslant vol$. We show this in the second part of this proof.

Using the semantic definition of $pcat$ (Figure 3.2), we know that,

$[\![p]\!](\sigma_0) = \widehat{[\![f]\!]}(\widehat{[\![g]\!]}([\![z \leftsquigarrow N(0,1)]\!](\sigma_0)))$

We first analyze $[\![z \leftsquigarrow N(0,1)]\!](\sigma_0)$. Again using the semantic definition of $pcat$, we write,

$$\begin{aligned} [\![z \leftsquigarrow N(0,1)]\!](\sigma_0) &= \mathbb{E}_{z \sim N(0,1)}[\lambda \nu. \delta_{\sigma_0[z \mapsto \nu]}] \\ &= \lambda \nu'. \textstyle\int_a N(a) \cdot \delta_{\sigma_0[z \mapsto a]}(\nu') \\ &= \lambda \nu'. 1_{\nu' = \sigma_0[z \mapsto a]} \cdot N(a) \end{aligned} \tag{2.35}$$

We are interested in the volume of the set Σ_z, defined as, $\Sigma_z = \{\sigma \mid \sigma(z) \in z_B\}$. Using the expression for $[\![z \leftsquigarrow N(0,1)]\!](\sigma_0)$ from above, we can now compute the required probability as follows,

$$\begin{aligned} \Pr_{\sigma \sim [\![z \leftsquigarrow N(0,1)]\!](\sigma_0)} (\sigma \in \Sigma_z) &= \textstyle\int_{\sigma \in \Sigma} ([\![z \leftsquigarrow N(0,1)]\!](\sigma_0))(\sigma) \cdot 1_{\sigma \in \Sigma_z} \\ &= \textstyle\int_{\sigma \in \Sigma} (1_{\sigma = \sigma_0[z \mapsto a]} \cdot N(a)) \cdot 1_{\sigma \in \Sigma_z} \\ &= \textstyle\int_{\sigma \in \Sigma_z} (1_{\sigma = \sigma_0[z \mapsto a]} \cdot N(a)) \\ &= \textstyle\int_{a \in z_B} N(a) \quad \text{(by uniqueness of } \sigma_0[z \mapsto a]) \\ &= vol' \end{aligned} \tag{2.36}$$

This shows that starting from any $\sigma_0 \in \Sigma$, after executing the first statement of p, the probability that the value stored at z lies in the box z_B is vol'.

Next, we analyze $[\![z \leftsquigarrow N(0,1)]\!](\sigma_0)$. In particular, we are interested in the volume of the

set, $\widetilde{[\![g]\!]}(\Sigma_z)$ (which is notational abuse for the set $\{\widetilde{[\![g]\!]}(\sigma) \mid \sigma \in \Sigma_z\}$). We can lower bound this volume as follows,

$$\begin{aligned}
\Pr_{\sigma \sim \widehat{[\![g]\!]}([\![z \leftsquigarrow N(0,1)]\!](\sigma_0))} (\sigma \in \widetilde{[\![g]\!]}(\Sigma_z)) &= \textstyle\int_{\sigma \in \Sigma} (\widehat{[\![g]\!]}([\![z \leftsquigarrow N(0,1)]\!](\sigma_0))(\sigma) \cdot 1_{\sigma \in \widetilde{[\![g]\!]}(\Sigma_z)} \\
&= \textstyle\int_{\sigma \in \widetilde{[\![g]\!]}(\Sigma_z)} \widehat{[\![g]\!]}([\![z \leftsquigarrow N(0,1)]\!](\sigma_0))(\sigma) \\
&\geqslant \textstyle\int_{\sigma \in \Sigma_z} [\![z \leftsquigarrow N(0,1)]\!](\sigma_0))(\sigma) \text{ (from Corollary 4)} \\
&= vol' \quad \text{(from Equation 2.36)}
\end{aligned} \tag{2.37}$$

We can similarly show that,

$$\Pr_{\sigma \sim \widehat{[\![f]\!]}(\widehat{[\![g]\!]}([\![z \leftsquigarrow N(0,1)]\!](\sigma_0)))} (\sigma \in \widetilde{[\![f]\!]}(\widetilde{[\![g]\!]}(\Sigma_z))) \geqslant vol' \tag{2.38}$$

Now, $\sigma^B[z \mapsto z_B]$ defined on line 2 of algorithm 2 is such that $\gamma(\sigma^B[z \mapsto z_B]) = \Sigma_z$. From Theorem 11, we can conclude that,

$$\widetilde{[\![g]\!]}(\Sigma_z) \subseteq \gamma([\![g]\!]_B(\sigma^B[z \mapsto z_B])) \tag{2.39}$$

Similarly, from Theorem 6, we can conclude that,

$$\widetilde{[\![f]\!]}(\widetilde{[\![g]\!]}(\Sigma_z)) \subseteq \gamma([\![f]\!]_L([\![g]\!]_B(\sigma^B[z \mapsto z_B]))_1) \tag{2.40}$$

From Equation 2.38 and Equation 2.40, we conclude that,

$$\Pr_{\sigma \sim [\![p]\!](\sigma_0)} (\sigma \in \gamma([\![f]\!]_L([\![g]\!]_B(\sigma^B[z \mapsto z_B])))_1) \geqslant vol' \tag{2.41}$$

Consequently,

$$\Pr_{\sigma, \sigma' \sim [\![p]\!](\sigma_0)} (\sigma, \sigma' \in \gamma([\![f]\!]_L([\![g]\!]_B(\sigma^B[z \mapsto z_B])))_1) \geqslant vol' \times vol' = vol \tag{2.42}$$

since each act of sampling is independent. ■

This theorem is applicable for any program p in the required form, such that g and f are cat programs, variable z is not written to by g and f ($\mathbf{outv}(\cdot)$ gives the set of variables that a program writes to, $\mathbf{inv}(\cdot)$ gives the set of live variables at the start of a program). It states that the result (k_U, d, vol) of invoking `PROLIP` on p with box z_B is safe, i.e., with probability at least vol, any pair of program states (σ, σ'), randomly sampled from the distribution denoted by $[\![p]\!](\sigma_0)$, where σ_0 is any initial state, satisfies the Lipschitzness property (with constant k_U) and has z variables mapped to vectors in the box z_B.

2.4.3 Sketch of Proof-Search Algorithm

Algorithm 3: Checking Probabilistic Robustness.

Input:

p: $pcat$ program.

r: Input closeness bound.

ϵ: Probabilistic bound.

k: Lipschitz constant.

$\texttt{gas}$: Iteration bound.

Output: $\{\mathtt{tt}, ?\}$

```
1  pr_l := 0; pr_r := 0; pr_f := 0;
2  α := INIT_AGENT(dim(z), r, ϵ, k);
3  while (pr_l < (1 − ϵ)) ∧ (gas ≠ 0) do
4      gas := gas − 1;
5      z_B := CHOOSE(α);
6      (k_U, d, vol) := PROLIP(p, z_B, x, y);
7      UPDATE_AGENT(α, k_U, d, vol);
8      if d ⩽ r then
9          pr_r := pr_r + vol;
10         if k_U ⩽ k then
11             pr_l := pr_l + vol;
12             pr_f := pr_f/pr_r;
13 end while
14 if gas = 0 then
15     return ? ;
16 else
17     return tt ;
```

We give a sketch of a proof-search algorithm that uses the PROLIP algorithm as a prim-

itive. The inputs to such an algorithm are a $pcat$ program p in the appropriate form, the constants r, ϵ, and k that appear in the formulation of probabilistic Lipschitzness, and a resource bound `gas` that limits the number of times `PROLIP` is invoked. This algorithm either finds a proof or runs out of `gas`. Before describing the algorithm, we recall the property we are trying to prove, stated as follows, $\Pr_{\sigma,\sigma' \sim [\![p]\!](\sigma_0)} (\|\sigma(y) - \sigma'(y)\| \leqslant k * \|\sigma(x) - \sigma'(x)\| \mid \|\sigma(x) - \sigma'(x)\| \leqslant r) \geqslant 1 - \epsilon$ The conditional nature of this probabilistic property complicates the design of the proof-search algorithm, and we use the fact that $Pr(A \mid B) = Pr(A \wedge B)/Pr(B)$ for computing conditional probabilities. Accordingly, the algorithm maintains three different probability counters, namely, pr_l, pr_r, and pr_f, which are all initialized to zero as the first step (line 1). pr_l records the probability that a randomly sampled pair of program states (σ, σ') satisfies the Lipschitzness and closeness property (i.e., $(\|\sigma(y) - \sigma'(y)\| \leqslant k * \|\sigma(x) - \sigma'(x)\|) \wedge (\|\sigma(x) - \sigma'(x)\| \leqslant r)$). pr_r records the probability that a randomly sampled pair of program states satisfies the closeness property (i.e., $\|\sigma(x) - \sigma'(x)\| \leqslant r$). pr_f tracks the conditional probability which is equal to pr_l/pr_r. After initializing the probability counters, the algorithm initializes an "agent" (line 2), which we think of as black-box capable of deciding which box-shaped regions in z should be explored. Ideally, we want to pick a box such that - (i) it has a high probability mass, (ii) it satisfies, both, Lipschitzness and closeness. Of course, we do not know a priori if Lipschitzness and closeness will hold for a particular box in z, the crux of the challenge in designing a proof-search algorithm. Here, we leave the algorithm driving the agent's decisions unspecified (and hence, refer to the proof-search algorithm as a sketch). After initializing the agent, the algorithm enters a loop (lines 3 - 13) that continues till we have no `gas` left or we have found a proof. Notice that if $(pr_l \geqslant (1 - \epsilon))$, the probabilistic Lipschitzness property is certainly true, but this is an overly strong condition that maybe false even when probabilistic Lipschitzness holds. For instance, if ϵ was 0.1 and the ground-truth value of pr_r for the program p was 0.2, then pr_l could never be $\geqslant 0.9$, even if probabilistic Lipschitzness holds. However, continuing with our algorithm description,

after decrementing gas (line 4), the algorithm queries the agent for a box in z (line 5), and runs PROLIP with this box, assuming x as the input variable of f and y as the output (line 6). Next, the agent is updated with the result of calling PROLIP, allowing the agent to update it's internal state (line 7). Next, we check if for the currently considered box (z_B), the maximum distance between the inputs to f is less than r (line 8), and if so, we update the closeness probability counter pr_r (line 9). We also check if the upper bound of the local Lipschitzness constant returned by PROLIP is less than k (line 10), and if so, update pr_l (line 11) and pr_f (line 12). Finally, if we have exhausted the gas, we were unable to prove the property, otherwise we have a proof of probabilistic Lipschitzness.

2.4.4 Discussion

Informally, we can think of the Jacobian analysis as computing two different kinds of "information" about a neural network: (i) an overapproximation of the outputs, given a set of inputs σ^B, using the box analysis; (ii) an upper bound on the local Lipschitz constant of the neural network for inputs in σ^B. The results of the box analysis are used to overapproximate the set of "program paths" in the neural network exercised by inputs in σ^B, safely allowing the Jacobian computation to be restricted to this set of paths. Consequently, it is possible to replace the use of box domain in (i) with other abstract domains like zonotopes [72] or DeepPoly [33] for greater precision in overapproximating the set of paths. In contrast, one needs to be very careful with the abstract domain used for the analysis of the generative model g in algorithm 2, since the choice of the abstract domain has a dramatic effect on the complexity of the volume computation algorithm VOL invoked by the PROLIP algorithm. While Gaussian volume computation of boxes is easy, it is hard for general convex bodies [73, 74, 75] unless one uses randomized algorithms for volume computation [76, 70]. Finally, note that the design of a suitable agent for iteratively selecting the input regions to analyze in algorithm 3 remains an open problem.

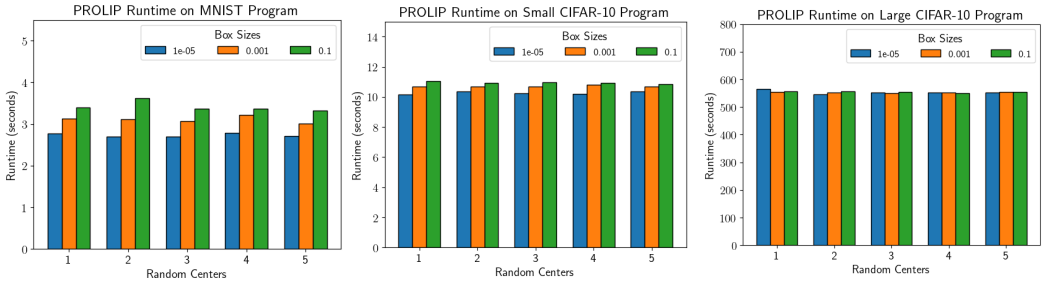
2.5 Empirical Evaluation

We aim to empirically evaluate the computational complexity of PROLIP. We ask the following questions: **(RQ1)** Given a program, is the run time of PROLIP affected by the size and location of the box in z? **(RQ2)** What is the run time of PROLIP on popular generative models and NNs?

2.5.1 Experimental Setup

We implement PROLIP in Python, using Pytorch, Numpy, and SciPy for the core functionalities, and Numba for program optimization and parallelization. We run PROLIP on three $pcat$ programs corresponding to two datasets: the MNIST dataset and the CIFAR-10 dataset. Each program has a generator network g and a classifier network f. The g networks in each program consist of five convolution transpose layers, four batch norm layers, four ReLU layers, and a tanh layer. The full generator architectures and parameter weights can be seen in [77]. The f network for the MNIST program consists of three fully connected layers and two ReLU layers. For the CIFAR-10 dataset, we create two different $pcat$ programs: one with a large classifier architecture and one with a small classifier architecture. The f network for the large CIFAR-10 program consists of seven convolution layers, seven batch norm layers, seven ReLU layers, four maxpool layers, and one fully connected layer. The f network for the small CIFAR-10 program consists of two convolution layers, two maxpool layers, two ReLU layers, and three fully connected layers. The full classifier architectures and parameter weights for the MNIST and large CIFAR-10 program can be seen in [78].

In our experiments, each generative model has a latent space dimension of 100, meaning that the model samples a vector of length 100 from a multi-dimensional normal distribution, which is then used by the generator network. We create five random vectors of length 100 by randomly sampling each element of the vectors from a normal distribution. For each



(a) (b) (c)

Figure 2.6: PROLIP run times

vector, we create three different sized square boxes by adding and subtracting a constant from each element in the vector. This forms an upper and lower bound for the randomly-centered box. The constants we chose to form these boxes are 0.00001, 0.001, and 0.1. In total, 15 different data points are collected for each program. We ran these experiments on a Linux machine with 32 vCPU's, 204 GB of RAM, and no GPU.

2.5.2 Results

RQ1. As seen in Figure 2.6a and Figure 2.6b, there is a positive correlation between box size and run time of PROLIP on the MNIST and small CIFAR-10 programs. This is likely because as the z input box size increases, more branches in the program stay unresolved, forcing the analysis to reason about more of the program. However, z box size does not seem to impact PROLIP run time on the large CIFAR-10 program (Figure 2.6c) as the time spent in analyzing convolution layers completely dominates any effect on run time of the increase in z box size.

RQ2. There is a significant increase in the run time of PROLIP for the large CIFAR-10 program compared to the MNIST and small CIFAR-10 programs, and this is due to the architectures of their classifiers. When calculating the abstract Jacobian matrix for an affine assignment statement ($y \leftarrow w \cdot x + \beta$), we multiply the weight matrix with the incoming abstract Jacobian matrix. The dimensions of a weight matrix for a fully connected layer is $N_{in} \times N_{out}$ where N_{in} is the number of input neurons and N_{out} is the number of output neurons. The dimensions of a weight matrix for a convolution layer are $C_{out} \cdot H_{out} \cdot W_{out} \times C_{in} \cdot H_{in} \cdot W_{in}$ where C_{in}, H_{in}, and W_{in} are the input's channel,

height, and width dimensions and C_{out}, H_{out}, and W_{out} are the output's channel, height, and width dimensions. For our MNIST and small CIFAR-10 classifiers, the largest weight matrices formed had dimensions of 784×256 and 4704×3072 respectively. In comparison, the largest weight matrix calculated in the large CIFAR-10 classifier had a dimension of 131072×131072. Propagating the Jacobian matrix for the large CIFAR-10 program requires first creating a weight matrix of that size, which is memory intensive, and second, multiplying the matrix with the incoming abstract Jacobian matrix, which is computationally expensive. The increase in run time of the PROLIP algorithm can be attributed to the massive size blow-up in the weight matrices computed for convolution layers.

Other Results. Table Table 2.1 shows the upper bounds on local Lipschitz constant computed by the PROLIP algorithm for every combination of box size and $pcat$ program considered in our experiments. The computed upper bounds are comparable to those computed by the Fast-Lip algorithm from [57] as well as other state-of-the-art approaches for computing Lipschitz constants of neural networks. A phenomenon observed in our experiments is the convergence of local Lipschitz constants to an upper bound, as the z box size increases. This occurs because beyond a certain z box size, for every box in z, the output bounds of g represent the entire input space for f. Therefore any increase in the z box size, past the tipping point, results in computing an upper bound on the global Lipschitz constant of f.

The run time of the PROLIP algorithm can be improved by utilizing a GPU for matrix multiplication. The multiplication of massive matrices computed in the Jacobian propagation of convolution layers or large fully connected layers accounts for a significant portion of the run time of PROLIP, and the run time can benefit from GPU-based parallelization of matrix multiplication. Another factor that slows down our current implementation of PROLIP algorithm is the creation of the weight matrix for a convolution layer. These weight matrices are quite sparse, and constructing sparse matrices that hold '0' values implicitly can be much faster than explicitly constructing the entire matrix in memory, which is what

Table 2.1: Local Lipschitz constants discovered by PROLIP

Box Size	MNIST Lip Constant	Large CIFAR Lip Constant	Small CIFAR Lip Constant
1e-05	1.683e1	5.885e14	3.252e5
0.001	1.154e2	8.070e14	4.218e5
0.1	1.154e2	8.070e14	4.218e5
1e-05	1.072e1	5.331e14	1.814e5
0.001	1.154e2	8.070e14	4.218e5
0.1	1.154e2	8.070e14	4.218e5
1e-05	1.460e1	6.740e14	2.719e5
0.001	1.154e2	8.070e14	4.218e5
0.1	1.154e2	8.070e14	4.218e5
1e-05	1.754e1	6.571e14	2.868e5
0.001	1.154e2	8.070e14	4.218e5
0.1	1.154e2	8.070e14	4.218e5
1e-05	1.312e1	5.647e14	2.884e5
0.001	1.154e2	8.070e14	4.218e5
0.1	1.154e2	8.070e14	4.218e5

our current implementation does.

2.6 Related Work

Our work draws from different bodies of literature, particularly literature on *verification of* NNs, *Lipschitz analysis of programs and* NNs, and *semantics and verification of probabilistic programs*. These connections and influences have been described in detail in section 3.1. Here, we focus on describing connections with existing work on proving probabilistic/statistical properties of NNs.

[40] is the source of the probabilistic Lipschitzness property that we consider. They propose a proof-search algorithm that (i) constructs a product program [79], (ii) uses an abstract interpreter with a powerset polyhedral domain to compute input pre-conditions that guarantee the satisfaction of the Lipschitzness property, (iii) computes approximate volumes of these input regions via sampling. They do not implement this algorithm. If one

encodes the Lipschitzness property as disjunction of polyhedra, the number of disjuncts is exponential in the number of dimensions of the output vector. There is a further blow-up in the number of disjuncts as we propagate the abstract state backwards.

Other works on probabilistic properties of NNs [80, 81] focus on local robustness. Given an input x_0, and an input distribution, they compute the probability that a random sample x' drawn from a ball centered at x_0 causes non-robust behavior of the NN at x' compared with x_0. [80] computes these probabilities via sampling while [81] constructs analytical expressions for computing upper and lower bounds of such probabilities. Finally, [82] presents a model-counting based approach for proving quantitative properties of NNs. They translate the NN as well as the property of interest into SAT constraints, and then invoke an approximate model-counting algorithm to estimate the number of satisfying solutions. We believe that their framework may be general enough to encode our problem but the scalability of such an approach remains to be explored. We also note that the guarantees produced by [82] are statistical, so one is unable to claim with certainty if probabilistic Lipschitzness is satisfied or violated.

2.7 Conclusion

We study the problem of algorithmically proving probabilistic Lipschitzness of NNs with respect to generative models representing input distributions. We employ a language-theoretic lens, thinking of the generative model and NN, together, as programs of the form $z \leftsquigarrow N(0,1); g; f$ in a first-order, imperative, probabilistic programming language $pcat$. We develop a sound local Lipschitzness analysis for cat, a non-probabilistic sublanguage of $pcat$ that performs a Jacobian analysis under the hood. We then present PROLIP, a provably correct algorithmic primitive that takes in a box-shaped region in the latent space of the generative model as an input, and returns a lower bound on the volume of this region as well as an upper bound on a local Lipschitz constant of f. Finally, we sketch a proof-search algorithm that uses PROLIP and avoids expensive volume computation operations in

the process of proving theorems about probabilistic programs. Empirical evaluation of the computational complexity of PROLIP suggests its feasibility as an algorithmic primitive, although convolution-style operations can be expensive and warrant further investigation.

CHAPTER 3

OBSERVATIONAL ABSTRACT INTERPRETERS

3.1 Introduction

Program verification, as used colloquially, refers to the practice of algorithmically finding program proofs, i.e., proofs of program judgments. These program judgments come in many forms, common forms are either type-theoretic judgments like $\Gamma \vdash e : t$ saying that in context Γ program e has type t, or program logic judgments of the form, $\{P\}e\{Q\}$, particularly when e is from an effectful language, where P is a pre-condition and Q is a post-condition of e.[1] Irrespective of the form of the judgment, a common step in the proof strategy employed by proof search algorithms is to compute semantic invariants of e which are then further used to construct the proofs of program judgments. The use of semantic invariants is particularly common when the programs or terms e are only partially annotated or are completely unannotated (à la Curry where programs are thought to be terms from an untyped language and the type system is extrinsic [83]). Informally, a semantic invariant is a simplified representation of the meaning of a program and practically, one wants these representations to be efficiently computed even when the program under analysis is non-terminating. A unifying perspective on algorithms for computing such invariants is provided by the theory of abstract interpretation [8, 9].

In general, the decision problems addressed by program verification are undecidable [2]. Even in the instances where the problems are decidable, the ability of an invariant-based proof search algorithm to find a proof (or a counterexample) crucially depends on the computed invariants. Invariants computed by abstract interpreters, in turn, depend on the abstract semantic domain and the abstract semantic function used to construct the ab-

[1] There are many connections between these two judgment forms that we do not elaborate here.

stract interpreter. The theory of abstract interpretation defines language semantics as a pair of a concrete semantic domain and a semantic function. The theory also defines the manner in which the concrete semantics should relate to an abstract semantics so that the invariants computed using the abstract semantics can be soundly used in the background type theory/program logic for constructing a program proof. However, defining an abstract semantics that leads to efficient computation of useful invariants requires creativity and theoretical expertise.

Many ideas have been presented in the literature for making the process of designing abstract semantics "easier" - [15] present a systematic approach for constructing an abstract interpreter starting from abstract machine semantics of higher-order languages and a number of follow-on works extend these ideas [16, 17, 18, 84, 85]; calculational abstract interpretation yields the abstract semantic function automatically given the concrete semantics and the abstract semantic domain [86, 87, 88, 89]; in the counterexample-guided abstraction refinement (CEGAR) style of abstract interpretation [90], the designer defines a set (finite or infinite) of "correct" abstract semantics and, given a specific program judgment, the CEGAR algorithm searches through this set for an abstract semantics that can efficiently yield a proof (or a counterexample) of the judgment. While all these ideas have helped make the design of effective abstract interpreters easier, the design process still involves much human ingenuity.

A different, increasingly common, proof strategy employed by proof search algorithms is to modify the program under study and embed it with run time or dynamic checks. This allows making hypotheses about program behavior such that a proof of the required program judgment can be constructed. This type of reasoning has been popularized by the gradual typing [3, 4] and hybrid typing [5, 6] philosophy as well as the work on using logical abduction for program reasoning [91, 92]. Ideally, we want to compute the weakest hypotheses that allow the construction of a program proof but inferring such hypotheses is not trivial.

We are interested in the design of proof search algorithms that combine the use of semantic invariants and dynamic checks. Apart from recent work on gradual liquid type inference [93] and gradual program verification [94], such a combination has been relatively under explored formally. In this work, we present the design of a new class of abstract interpreters that compute semantic invariants while making hypotheses about program behavior, embedded as dynamic checks in the program. These hypotheses help the abstract interpreter compute potentially stronger semantic invariants, at the cost of the overheads of dynamic run time checks. A key challenge in such hypothesis-based reasoning is automatically computing the appropriate hypothesis. Typically, the computation of these hypotheses is guided by the proof goal. In our abstract interpreter design, we instead rely on observations about the program behavior to infer the hypotheses. Intuitively, the idea is to make hypotheses that are consistent with the observed behavior of the program. This observational style of reasoning motivates our use of the term, *observational abstract interpreters*, to refer to the class of abstract interpreters that we propose.

The benefit of an observational reasoning style, particularly in combination with the hypotheses-based reasoning, is that we no longer need to derive custom proof goal guided algorithms, specific to the type theory or program logic we are working with, for computing the appropriate hypotheses. More interestingly, such observational, hypothetical proofs, can be used to make judgments about the program behavior in the "commonly" observed ways of using the program, even if the same judgment cannot be proven for the program in general. On the other hand, an obvious drawback of using program observations (instead of the proof goal) for computing hypotheses is that the computed hypotheses are not guaranteed to be strong enough to allow the construction of a program proof. In any case, we believe that this combination of invariant-based reasoning with hypotheses-based reasoning, where the hypotheses are inferred from program observations is an interesting point worth further exploration.

We formalize our ideas in the context of a simple higher-order language (λ_S). In partic-

$$
\begin{array}{rcl}
 & i \in \mathbb{Z} \quad x \in Var \quad l \in Lbl & \\
a \in Atom & ::= & i \mid x \mid \boldsymbol{\lambda}(x).e \mid \mathbf{abort} \\
\oplus \in IOp & ::= & + \mid - \\
\odot \in Op & ::= & \oplus \mid @ \\
e \in Exp & ::= & (a)^l \mid (e \odot e)^l \mid (\mathbf{if0}(e)\{e\}\{e\})^l
\end{array}
$$

Figure 3.1: $\lambda_S(\lambda_{SA})$ language syntax

ular, starting from an abstract machine semantics of λ_S, we demonstrate the construction of a generic observational abstract interpreter for λ_S, and in the process, we formally define the notion of program observations as well as the notion of correctness or soundness for an observational abstract interpreter. Our formal development is heavily inspired by the abstracting abstract machines (AAM) [15] style of abstract interpreter construction. Observational abstract interpreters are structured as monadic abstract interpreters [16, 17, 18] that reify the notion of an AAM-style interpreter. We believe that the recipe we present here for constructing observational abstract interpreters of λ_S can also be applied to other languages.

Our main contributions are as follows - (i) we propose observational abstract interpreters, a synthesis of invariant-based reasoning about programs with hypothesis-based reasoning and observational program reasoning, (ii) we formally construct a generic observational abstract interpreter for λ_S, a higher-order language, (iii) we present an instantiation of the generic observational abstract interpreter for λ_S, yielding an observational interval analysis for programs in λ_S.

3.2 Language Definition

We present our ideas with the help of λ_S, a higher-order language with built-in integers and conditionals. The language is fairly standard, and we adopt the syntax and semantics from [17]. λ_S syntax is defined in Figure 3.1. Note that function application is explicitly represented using the @ operator. Figure 3.1 also describes the syntax of λ_{SA}, which additionally allows programs with $\mathbf{abort}$ expressions (the gray background color is intended to high-

light that **abort** expressions are only allowed in λ_{SA} programs, but not in λ_S programs). These **abort** expressions enable dynamic checks to be embedded in the programs. We distinguish between λ_S and λ_{SA} for ease of formal presentation. We design observational abstract interpreters that are capable of analyzing λ_S programs to produce hypothetical semantics invariants. These hypotheses are then embedded in the original λ_S program with the help of **abort** expressions, producing a λ_{SA} program. Note that every expression in a λ_S (λ_{SA}) is associated with a unique label, drawn from an infinitely large set of labels (Lbl). To avoid notational clutter, we do not show the labels in the rest of the paper, but assume that such a label always exists. Moreover, we assume the existence of a function get-Label that accepts an expression and returns the label associated with the expression.

The semantics of λ_S (and λ_{SA}) are presented in Figure 3.2. We define the language semantics using the formalism of abstract machines. Before describing the semantics, we make a note on the metalanguage used in Figure 3.2 and the rest of the paper. Our metalanguage notation resembles Haskell syntax, though we freely use other syntactic constructs. Function application is notated as $f(e)$, where f is the function applied to e. Pairs and tuples are notated by $\langle\cdot\rangle$. We reserve $=$ to explicitly notate equality, with $:=$ used to notate definitions, and $::=$ notates datatype definitions. Wherever necessary, we explain the notation that we use.

The abstract machine semantics of λ_S (and λ_{SA}) is defined as a transition relation ($\rightsquigarrow$) on the set Σ of abstract machine states. An abstract machine state is a 6-tuple consisting of a program/expression, an environment (Env), a store for values ($Store$), a store for continuations ($KStore$) that are linked together (similar to a call stack), the address of the next continuation ($KAddr$), and a time component ($Time$). The abstract machine semantics presented here is similar to the CESK machine [95], except that the continuations are threaded through the store, and the time component is used to compute a new address for allocation in the value or continuation store. As mentioned earlier, the abstract machine design presented here follows the design by Darais, Might, and Van Horn, which is itself

$$
\begin{array}{rcl}
t \in Time & := & Exp^* \\
a \in Addr & := & Var \times Time \\
\rho \in Env & := & Var \rightharpoonup Addr \\
s \in Store & := & Addr \rightharpoonup Val \\
kf \in KFrame & := & Frame \times Env \\
ka \in KAddr & := & Time \\
ks \in KStore & := & KAddr \rightharpoonup KFrame \times KAddr \\
c \in Clo & ::= & \langle \boldsymbol{\lambda}(x).e, \rho \rangle \\
v \in Val & ::= & i \mid c \mid \mathbf{abort} \\
fr \in Frame & ::= & \square \odot e \mid v \odot \square \mid \mathbf{if0}(\square)\{e\}\{e\} \\
\sigma \in \Sigma & ::= & \langle e, \rho, s, ka, ks, t \rangle
\end{array}
$$

(a) Type definitions

$$
\begin{array}{rcl}
[\![\cdot]\!]_A & : & Atom \to (Env \times Store \rightharpoonup Val) \\
[\![i]\!]_A(\langle \rho, s \rangle) & := & i \\
[\![x]\!]_A(\langle \rho, s \rangle) & := & s(\rho(x)) \\
[\![\boldsymbol{\lambda}(x).e]\!]_A(\langle \rho, s \rangle) & := & \langle \boldsymbol{\lambda}(x).e, \rho \rangle \\
[\![\cdot]\!]_\delta & : & IOp \to (\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}) \\
[\![+]\!]_\delta(\langle i_1, i_2 \rangle) & := & i_1 + i_2 \\
[\![-]\!]_\delta(\langle i_1, i_2 \rangle) & := & i_1 - i_2
\end{array}
$$

(b) Denotational semantics of atomic expressions

$$
\begin{array}{rcll}
\cdot \rightsquigarrow \cdot & : & P(\Sigma \times \Sigma) & \\
\langle e_1 \odot e_2, \rho, s, ka, ks, t \rangle & \rightsquigarrow & \langle e_1, \rho, s, ka, ks', t' \rangle & \text{where} \\
t' & := & (e_1 \odot e_2) :: t & \\
ks' & := & ks[t' \mapsto \langle \langle \square \odot e_2, \rho \rangle, ka \rangle] & \\
\langle \mathbf{if0}(e_1)\{e_2\}\{e_3\}, \rho, s, ka, ks, t \rangle & \rightsquigarrow & \langle e_1, \rho, s, ka, ks', t' \rangle & \text{where} \\
t' & := & (\mathbf{if0}(e_1)\{e_2\}\{e_3\}) :: t & \\
ks' & := & ks[t' \mapsto \langle \langle \mathbf{if0}(\square)\{e_2\}\{e_3\}, \rho \rangle, ka \rangle] & \\
\langle \mathbf{abort}, \rho, s, ka, ks, t \rangle & \rightsquigarrow & \langle \mathbf{abort}, \rho, s, t, ks, t \rangle & \\
\langle a, \rho, s, ka, ks, t \rangle & \rightsquigarrow & \langle e, \rho', s, t, ks', t' \rangle & \text{where} \\
t' & := & a :: t & \\
\langle \langle \square \odot e, \rho' \rangle, ka' \rangle & := & ks(ka) & \\
ks' & := & ks[t' \mapsto \langle \langle [\![a]\!]_A(\langle \rho, s \rangle) \odot \square, \rho \rangle, ka' \rangle] & \\
\langle a, \rho, s, ka, ks, t \rangle & \rightsquigarrow & \langle e, \rho'', s', ka', ks, t' \rangle & \text{where} \\
t' & := & a :: t & \\
\langle \langle \langle \boldsymbol{\lambda}(x).e, \rho' \rangle @ \square, \rho' \rangle, ka' \rangle & := & ks(ka) & \\
\rho'' & := & \rho'[x \mapsto \langle x, t' \rangle] & \\
s' & := & s[\langle x, t' \rangle \mapsto [\![a]\!]_A(\langle \rho, s \rangle)] & \\
\langle i_2, \rho, s, ka, ks, t \rangle & \rightsquigarrow & \langle i, \rho, s, ka', ks, t' \rangle & \text{where} \\
t' & := & i_2 :: t & \\
\langle \langle i_1 \oplus \square, \rho' \rangle, ka' \rangle & := & ks(ka) & \\
i & := & [\![\oplus]\!]_\delta(\langle i_1, i_2 \rangle) & \\
\langle i, \rho, s, ka, ks, t \rangle & \rightsquigarrow & \langle e, \rho', s, ka', ks, t' \rangle & \text{where} \\
t' & := & i :: t & \\
\langle \langle \mathbf{if0}(\square)\{e_1\}\{e_2\}, \rho' \rangle, ka' \rangle & := & ks(ka) & \\
e & := & \mathbf{if}\ i = 0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 &
\end{array}
$$

(c) Abstract machine semantics

Figure 3.2: $\lambda_S(\lambda_{SA})$ concrete semantics in the form of an abstract machine

$$
\begin{array}{rcl}
\overline{\mathsf{init\text{-}States}} & : & Exp^{-} \to \mathcal{P}(\Sigma) \\
\overline{\mathsf{init\text{-}States}}(e) & := & \textbf{let } \rho := \{\langle x, \langle x, \epsilon\rangle\rangle \mid x \in FV(e)\} \textbf{ in} \\
 & & \textbf{let } \mathsf{init\text{-}Store} := \{\{\langle\langle x, \epsilon\rangle, v_x\rangle \mid x \in FV(e)\} \mid \textstyle\bigwedge_{x \in FV(e)} v_x \in \mathbb{Z}\} \textbf{ in} \\
 & & \{\langle e, \rho, s, \epsilon, \bot, \epsilon\rangle \mid s \in \mathsf{init\text{-}Store}\} \\
 & & \\
[\![\cdot]\!]_{Cl} & : & Exp^{-} \to \mathcal{P}(\Sigma) \\
[\![e]\!]_{Cl} & := & \textbf{lfp } \lambda(x).\ x \cup \overline{\mathsf{init\text{-}States}}(e) \cup \{\sigma_2 \mid \sigma_1 \in x \wedge \sigma_1 \rightsquigarrow \sigma_2\}
\end{array}
$$

Figure 3.3: $\lambda_S(\lambda_{SA})$ collecting semantics

based on work by Van Horn and Might [15]. Since values and continuations are both allocated in their respective stores, by restricting the number of distinct locations/addresses in the store, one can easily abstract the abstract machine, yielding an abstract interpreter for the language, an observation that first appeared in [15].

Figure 3.2a defines the different components of an abstract machine state. We would like to draw notice to the definition of $Time$ and $Addr$. $Time$ is defined as a sequence of expressions, while an address is a pair of a variable name and time. We assume that each of the type (or set) defined here has the structure of a lattice. The semantics of atomic expressions and primitive operations are defined denotationally (Figure 3.2b), and the abstract machine semantics for compound expressions are defined by a relation (Figure 3.2c). Note that if the abstract machine encounters an **abort** expression while executing a λ_{SA} program, it steps to an unmodified state.

Following all these definitions, we are finally ready to define the notion of "meaning" of a program, also referred to as *collecting semantics* in the abstract interpretation literature. Figure 3.3 defines the collecting semantics of $\lambda_S(\lambda_{SA})$. Note that the collecting semantics are not defined for all the expressions (Exp) in our language. Instead, we only consider programs where the free variables are of type $\mathbb{Z}$, and name this set of expressions, Exp^{-}. The meaning of a program/expression in Exp^{-} is described in terms of abstract machine states. Intuitively, the meaning of a program is the set of all abstract machines states that are "reachable" from a set of "initial" states. Let us unpack this definition. Given a program e, the definition of initial states ($\overline{\mathsf{init\text{-}States}}$) in Figure 3.3 states that, if a program e has no free variables, then the set of initial states is just the singleton set, $\{\langle e, \bot, \bot, \epsilon, \bot, \epsilon\rangle\}$.

For programs with free variables of type $\mathbb{Z}$ (set of free variables is represented by FV), the set of initial states is defined such that all possibles ways of "closing" the program, i.e., assigning values to the free variables, are represented in the set. In Figure 3.3, this is captured by the definition of init-Store, which uses the set-builder notation in a nested manner. Given a free variable x, we assume that the initial value assigned to x is stored at address $\langle x, \epsilon \rangle$ in the store s. Then, the collecting semantics, notated by $[\![\cdot]\!]_{Cl}$, is defined as the least fixed point of a function of type $\mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$. This function uses the definitions of $\overline{\text{init-States}}$ and the transition relation $\rightsquigarrow$ describing our abstract machine semantics. Defining a collecting semantics for expressions with free variables of function type is a problem of independent interest, and by only considering programs from Exp^{-}, we avoid dealing with that issue in this paper.

3.3 Monadic Interpreters: Concrete and Abstract

The Van Horn-Might [15] style of abstract machine semantics for higher-order languages makes it easy to refactor the abstract machine such that designing an abstract interpreter simply becomes a matter of redefining some interfaces (expressible as type classes in Haskell or modules in ML). The authors of [16] first noticed that the Van Horn-Might abstract machine can be refactored using monads. That interpreters for higher-order languages can be modularized and structured monadically has been known for a while [96, 97], but using the monadic structure to ease the design of abstract interpreters and simplify their proofs of correctness has only been recently investigated [16, 17, 18, 84, 85]. These recent advances play an important role in our design of observational abstract interpreters. In this section, we describe how the abstract machine semantics for λ_S can be modularized and expressed monadically, closely following [17]. We also show the manner in which the resulting monad can be instantiated to yield semantics equivalent to the collecting semantics defined in Figure 3.3, as well as an abstract interpreter for an interval analysis of λ_S programs.

$\underline{\mathring{Time}}$
tick : $Exp^- \times \mathring{Time} \to \mathring{Time}$

$\underline{\mathring{Val}}$
int-I : $\mathbb{Z} \to \mathring{Val}$
if0-E : $\mathring{Val} \to \mathcal{P}(Bool)$
clo-I : $\mathring{Clo} \to \mathring{Val}$
clo-E : $\mathring{Val} \to \mathcal{P}(\mathring{Clo})$
$[\![\oplus]\!]_{m\mathring{s}} : \mathring{Val} \times \mathring{Val} \to \mathring{Val}$

$\mathring{Addr} := Var \times \mathring{Time}$
$\mathring{Env} := Var \rightharpoonup \mathring{Addr}$
$\mathring{Clo} ::= \langle \boldsymbol{\lambda}(x).e, \rho \rangle$
$\mathring{Store} := \mathring{Addr} \rightharpoonup \mathring{Val}$
$\mathring{KFrame} := Frame \times \mathring{Env}$
$\mathring{KAddr} := \mathring{Time}$
$\mathring{KStore} := \mathring{KAddr} \rightharpoonup \mathcal{P}(\mathring{KFrame} \times \mathring{KAddr})$

$\underline{\mathring{\Sigma}}$
init-States : $Exp^- \to \mathring{\Sigma}$

$\underline{\mathring{m}}$
return : $\forall A.A \to \mathring{m}(A)$
bind : $\forall A, B.\mathring{m}(A) \to (A \to \mathring{m}(B)) \to \mathring{m}(B)$
get-Env : $\mathring{m}(\mathring{Env})$
put-Env : $\mathring{Env} \to \mathring{m}(1)$
get-Store : $\mathring{m}(\mathring{Store})$
put-Store : $\mathring{Store} \to \mathring{m}(1)$
get-KAddr : $\mathring{m}(\mathring{KAddr})$
put-KAddr : $\mathring{KAddr} \to \mathring{m}(1)$
get-KStore : $\mathring{m}(\mathring{KStore})$
put-KStore : $\mathring{KStore} \to \mathring{m}(1)$
get-Time : $\mathring{m}(\mathring{Time})$
put-Time : $\mathring{Time} \to \mathring{m}(1)$
mzero : $\forall A.\mathring{m}(A)$
$\cdot\langle + \rangle\cdot : \forall A.\mathring{m}(A) \times \mathring{m}(A) \to \mathring{m}(A)$

$\alpha^{\mathring{\Sigma} \leftrightarrow \mathring{m}} : (\mathring{\Sigma} \to \mathring{\Sigma}) \to (Exp^- \to \mathring{m}(Exp^-))$
$\gamma^{\mathring{\Sigma} \leftrightarrow \mathring{m}} : (Exp^- \to \mathring{m}(Exp^-)) \to (\mathring{\Sigma} \to \mathring{\Sigma})$

(a) Type definitions

Figure 3.4: λ_S monadic interpreter

```
step^m : Exp⁻ → m̊(Exp⁻)
step^m(e) := do
  ρ ← get-Env
  e'' ← case e of
    e₁ ⊙ e₂ → tick^m(e); push(⟨□ ⊙ e₂, ρ⟩); return(e₁)
    if0(e₁){e₂}{e₃} → do
      tick^m(e); push(⟨if0(□){e₂}{e₃}, ρ⟩); return(e₁)
    a → do
      v ← ⟦a⟧_mA; fr ← pop
      case fr of
        ⟨□ ⊙ e', ρ'⟩ → do
          tick^m(e); put-Env(ρ'); push(⟨v ⊙ □, ρ⟩); return(e')
        ⟨v'@□, ρ'⟩ → do
          tick^m(e); t ← get-Time; s ← get-Store
          ⟨λ(x).e', ρ''⟩ ← ↑_p(clo-E(v'))
          put-Env(ρ''[x ↦ ⟨x, t⟩])
          put-Store(s ⊔ [⟨x, t⟩ ↦ v]); return(e')
        ⟨v' ⊕ □, ρ'⟩ → tick^m(e); return(⟦⊕⟧_mδ(⟨v', v⟩))
        ⟨if0(□){e₁}{e₂}, ρ'⟩ → do
          tick^m(e); put-Env(ρ'); b ← ↑_p(if0-E(v)); refine(⟨a, b⟩)
          if(b) then return(e₁) else return(e₂)
        ⊥ → return(e)
  return(e'')
```

(b) Step function

```
⟦·⟧_mA : Atom → m̊(V̊al)
⟦i⟧_mA := return(int-I(i))
⟦x⟧_mA := do
  ρ ← get-Env; s ← get-Store
  if(x ∈ ρ) then return(s(ρ(x))) else return(⊥)
⟦λ(x).e⟧_mA := ρ ← get-Env; return(clo-I(⟨λ(x).e, ρ⟩))

push : KF̊rame → m̊(1)
push(fr) := do
  ka ← get-KAddr; ks ← get-KStore; ka' ← get-Time
  put-KStore(ks ⊔ [ka' ↦ ⟨fr, ka⟩]); put-KAddr(ka')

pop : m̊(KF̊rame)
pop := do
  ka ← get-KAddr; ks ← get-KStore;
  if(ka ∉ ks) then return(⊥)
  else⟨fr, ka'⟩ ← ↑_p(ks(ka)); put-KAddr(ka'); return(fr)

↑_p : ∀A.P(A) → m̊(A)
↑_p({a₁, ..., aₙ}) := return(a₁)⟨+⟩...⟨+⟩return(aₙ)

refine : Atom × Bool → m̊(1)
refine(⟨i, b⟩) := return(1)
refine(⟨x, b⟩) := do
  ρ ← get-Env; s ← get-Store
  if(b) then put-Store(s[ρ(x) ↦ int-I(0)]) else return(1)

tick^m : Exp⁻ → m̊(1)
tick^m(e) := do
  t ← get-Time
  put-Time(tick(⟨e, t⟩))
```

(c) Helper functions

$$\begin{array}{lcl} \llbracket\cdot\rrbracket_m & : & Exp^- \to \mathring{\Sigma} \\ \llbracket e\rrbracket_m & := & \mathbf{lfp}\ \lambda(x).\ x \sqcup \mathsf{init\text{-}States}(e) \sqcup (\gamma^{\mathring{\Sigma}\leftrightarrow\mathring{m}}(\mathsf{step}^{\mathsf{m}}))(x) \end{array}$$

(d) Collecting semantics

Figure 3.4: λ_S monadic interpreter

Figure 3.4 describes the design of a generic monadic interpreter for programs in λ_S with free variables of type $\mathbb{Z}$. The design of the monadic interpreter is based on the intuition that the computation performed by the interpreter (or the abstract machine) primarily depends on the structure of the expression being interpreted, and the interaction with the other components of the abstract machine state, like the environment and the store, can be hidden behind a monadic interface. This monadic interface in defined in Figure 3.4a. Our metalanguage supports Haskell-like typeclasses [98], and we define a typeclass $\mathring{m}$ that includes standard monadic operations like bind and return. In addition, the monad is required to support a number of get and put operations for interacting with the abstract machine state components. Additionally, the monad is also required to support non-deterministic choice operation $\langle + \rangle$. Besides the monad typeclass, the monadic interpreter design also requires abstracting other types that the interpreter interacts with via corresponding typeclasses. In our notation, we distinguish typeclass names from type names by using a small circle ($\mathring{name}$) over the typeclass names. The typeclass $\mathring{Time}$ has an associated operation, tick. The typeclass $\mathring{Val}$ has a number of operations associated with it that map from values of type $\mathbb{Z}$ and closures to elements of types instantiating $\mathring{Val}$, and vice versa. More details about these operations can be read in section 4.2 of [17]. Finally, we expect $\mathring{Time}, \mathring{Val}, \mathring{Addr}, \mathring{Env}, \mathring{Store}, \mathring{KFrame}, \mathring{KAddr}, \mathring{KStore}$, and $\mathring{\Sigma}$ to all have a lattice structure, i.e., they support the lattice operations $\sqcup$, $\sqcap$, and $\sqsubseteq$, as well as define lattice elements $\top$ and $\bot$.

Figure 3.4b defines the $\mathsf{step}^{\mathsf{m}}$ function describing a single step of the monadic interpreter. First, a comment on notation - we use the do notation from Haskell as well as ; for sequencing monadic operations. So $x \leftarrow s_1; s_2$ is syntactic sugar for $\mathsf{bind}(s_1)(\lambda(x).\ s_2)$, while

$$\begin{array}{l} \textbf{do} \\ x \leftarrow s_1 \\ s_2 \end{array}$$

is syntactic sugar for $\mathsf{bind}(s_1)(\lambda(x).\ s_2)$. Moreover, we allow combining these notations. The $\mathsf{step}^{\mathsf{m}}$ function uses a number of helper functions, defined in Figure 3.4c. A further comment on notation - in order to check if a partial may, say ρ, is defined for a certain key, say x, we use the notational shortcut $x \in \rho$. The structure of the $\mathsf{step}^{\mathsf{m}}$ function closely resembles the abstract machine transition relation defined in Figure 3.2c. The helper function $\uparrow_p$ helps hide the non-determinism behind the monadic interface. While the concrete interpreter for λ_S does not exhibit any non-determinism, we will soon see that the abstract interpreter is non-deterministic. Similarly, the function refine helps the abstract interpreter compute more precise results, particularly in cases where the branch taken by the conditional cannot be resolved.

Finally, Figure 3.4d defines the collecting semantics of a λ_S program in Exp^- using the monadic $\mathsf{step}^{\mathsf{m}}$ function. Note that the type signature of $\mathsf{step}^{\mathsf{m}}$ ($Exp^- \to \mathring{Exp}^-$) is incompatible with least-fixed point computation needed for computing the meaning of a program. As in [17], this problem is solved by defining a function $\gamma^{\mathring{\Sigma} \leftrightarrow \mathring{m}}$ that maps the monadic $\mathsf{step}^{\mathsf{m}}$ function, to a transition function of type $\mathring{\Sigma} \to \mathring{\Sigma}$, that can be iteratively invoked to compute the required least fixed point. The function $\alpha^{\mathring{\Sigma} \leftrightarrow \mathring{m}}$ does the opposite, with $\alpha^{\mathring{\Sigma} \leftrightarrow \mathring{m}}$ and $\gamma^{\mathring{\Sigma} \leftrightarrow \mathring{m}}$ representing a Galois connection between $Exp^- \to \mathring{m}(Exp^-)$ and $\mathring{\Sigma} \to \mathring{\Sigma}$.

3.3.1 Concrete Monadic Interpreter

The monadic concrete interpreter for λ_S is derived by instantiating the typeclasses defined in Figure 3.4. These typeclass instantiations are described in Figure 3.5. We make sure that the monadic interpreter is instantiated such that the resulting "concrete" monadic collecting semantics (notated by $\overline{[\![\cdot]\!]_m}$) is equivalent to the collecting semantics ($[\![\cdot]\!]_{Cl}$) defined in Figure 3.3. Notationally, concrete typeclass instantiations are indicated by a horizontal line over the typeclass names (for instance, $\overline{Time}$).

Note that $\overline{Time}, \overline{Val}, \overline{Clo}, \overline{Addr}, \overline{Env}, \overline{Store}$, and $\overline{KAddr}$ reuse the corresponding def-

initions from Figure 3.2a for the standard abstract machine semantics of λ_S. However, $\overline{KStore}$, i.e., the continuation store, is defined such that every address is mapped to a set of continuations. However, these sets are always singleton in the concrete semantics. The meanings of programs are elements of set $\overline{\Sigma}$, defined as the powerset of the set of abstract machine states. The lattice operations for $\overline{\Sigma}$, defined in Figure 3.5b, are straightforward. In the collecting semantics, we reuse the definition of $\overline{\mathsf{init\text{-}States}}$ from Figure 3.3 (ignoring the difference in the definitions of $\overline{KStore}$ and $KStore$ since it does not have any discernible effect on the definition of $\overline{\mathsf{init\text{-}States}}$).

The correctness of the monadic concrete collecting semantics with respect to the standard collecting semantics of λ_S is formally stated by the following proposition.

Proposition 13. *(Equivalence of $[\![\cdot]\!]_{Cl}$ and $\overline{[\![\cdot]\!]}_m$)*
$\forall e \in Exp^{-}.\ [\![e]\!]_{Cl} = \overline{[\![e]\!]}_m$

A proof of this equivalence can be found in prior works ([17]), and since our definitions of the standard collecting semantics and the monadic semantics presented here closely follows that of Darais, Might, and Van Horn, we do not present the proof here.

3.3.2 Abstract Monadic Interpreter

The flexibility and modularity afforded by the monadic design of the λ_S interpreter can be appreciated as one sets out to design an abstract interpreter for the language. We present a monadic abstract interpreter for λ_S that is capable of performing interval analysis of λ_S programs. As with the monadic concrete interpreter, we only need to instantiate the typeclasses in order to yield the abstract interpreter. We notate typeclass instances for the abstract interpreter with a hat over the typeclass name (for instance, $\widehat{Time}$).

Figure 3.6a includes all the typeclass definitions, except the monad definition. For Van Horn-Might abstract machines, the notion of time plays a key role in dictating the abstract machine behavior. In particular, the set of addresses available in the value and continuation stores depends on the definition of time. For the abstract interpreter, we want to finitize the

$t \in \overline{Time} := Time$
$\mathsf{tick}(\langle e, t\rangle) := e :: t$

$v \in \overline{Val} := Val$
$\mathsf{int\text{-}I}(i) := i$
$\mathsf{if0\text{-}E}(v) := \mathbf{if}(v = 0)\ \mathbf{then}\ \{\mathbf{tt}\}\ \mathbf{else}\ \{\mathbf{ff}\}$
$\mathsf{clo\text{-}I}(c) := c$
$\mathsf{clo\text{-}E}(v) := \{v\}$
$\overline{[\![+]\!]}_{m\delta}(\langle v, v'\rangle) := v + v'$
$\overline{[\![-]\!]}_{m\delta}(\langle v, v'\rangle) := v - v'$

$a \in \overline{Addr} := Addr$
$\rho \in \overline{Env} := Env$
$s \in \overline{Store} := Store$
$kf \in \overline{KFrame} := KFrame$
$ka \in \overline{KAddr} := KAddr$
$ks \in \overline{KStore} := \overline{KAddr} \rightharpoonup \mathcal{P}(\overline{KFrame} \times \overline{KAddr})$
$c \in \overline{Clo} := Clo$

$\psi \in \overline{\Psi} := \overline{Env} \times \overline{Store} \times \overline{KAddr} \times \overline{KStore} \times \overline{Time}$
$\overline{\sigma} \in \overline{\Sigma} = \mathcal{P}(Exp^- \times \overline{\Psi})$

$\alpha^{\overline{\Sigma} \leftrightarrow \overline{m}} : (\overline{\Sigma} \to \overline{\Sigma}) \to (Exp^- \to \overline{m}(Exp^-))$
$\alpha^{\overline{\Sigma} \leftrightarrow \overline{m}}(f)(e)(\psi) := f(\{\langle e, \psi\rangle\})$
$\gamma^{\overline{\Sigma} \leftrightarrow \overline{m}} : (Exp^- \to \overline{m}(Exp^-)) \to (\overline{\Sigma} \to \overline{\Sigma})$
$\gamma^{\overline{\Sigma} \leftrightarrow \overline{m}}(f)(\overline{\sigma}) := \bigcup_{\langle e,\psi\rangle \in \overline{\sigma}} f(e)(\psi)$

(a) Type definitions

$\sqsubseteq : \overline{\Sigma} \times \overline{\Sigma} \to Bool$
$\overline{\sigma} \sqsubseteq \overline{\sigma'} := \mathbf{if}(\overline{\sigma} \subseteq \overline{\sigma'})\ \mathbf{then\ tt\ else\ ff}$

$\sqcup : \overline{\Sigma} \times \overline{\Sigma} \to \overline{\Sigma}$
$\overline{\sigma} \sqcup \overline{\sigma'} := \overline{\sigma} \cup \overline{\sigma'}$

$\bot : \overline{\Sigma} := \varnothing$
$\top : \overline{\Sigma} := Exp^- \times \overline{\Psi}$

(b) Lattice operations for $\overline{\Sigma}$

Figure 3.5: λ_S monadic concrete interpreter

$\overline{m}(A) := \overline{\Psi} \to \mathcal{P}(A \times \overline{\Psi})$

$\mathsf{return}(x)(\psi) := \{\langle x, \psi\rangle\}$

$\mathsf{bind}(X)(f)(\psi) := \bigcup_{\langle x,\psi'\rangle \in X(\psi)} f(x)(\psi')$

$\mathsf{get\text{-}Env}(\langle \rho, s, ka, ks, t\rangle) := \{\langle \rho, \langle \rho, s, ka, ks, t\rangle\rangle\}$

$\mathsf{put\text{-}Env}(\rho')(\langle \rho, s, ka, ks, t\rangle) := \{(\langle 1, \langle \rho', s, ka, ks, t\rangle\rangle\}$

$\mathsf{get\text{-}Store}(\langle \rho, s, ka, ks, t\rangle) := \{\langle s, \langle \rho, s, ka, ks, t\rangle\rangle\}$

$\mathsf{put\text{-}Store}(s')(\langle \rho, s, ka, ks, t\rangle) := \{\langle 1, \langle \rho, s', ka, ks, t\rangle\rangle\}$

$\mathsf{get\text{-}KAddr}(\langle \rho, s, ka, ks, t\rangle) := \{\langle ka, \langle \rho, s, ka, ks, t\rangle\rangle\}$

$\mathsf{put\text{-}KAddr}(ka')(\langle \rho, s, ka, ks, t\rangle) := \{\langle 1, \langle \rho, s, ka', ks, t\rangle\rangle\}$

$\mathsf{get\text{-}KStore}(\langle \rho, s, ka, ks, t\rangle) := \{\langle ks, \langle \rho, s, ka, ks, t\rangle\rangle\}$

$\mathsf{put\text{-}KStore}(ks')(\langle \rho, s, ka, ks, t\rangle) := \{\langle 1, \langle \rho, s, ka, ks', t\rangle\rangle\}$

$\mathsf{get\text{-}Time}(\langle \rho, s, ka, ks, t\rangle) := \{\langle t, \langle \rho, s, ka, ks, t\rangle\rangle\}$

$\mathsf{put\text{-}Time}(t')(\langle \rho, s, ka, ks, t\rangle) := \{\langle 1, \langle \rho', s, ka, ks, t'\rangle\rangle\}$

$\mathsf{mzero}(\psi) := \{\}$

$(X_1 \langle + \rangle X_2)(\psi) := X_1(\psi) \sqcup X_2(\psi)$

(c) Monad definition

$$\begin{array}{lcl} \overline{[\![\cdot]\!]}_m & : & Exp^- \to \overline{\Sigma} \\ \overline{[\![e]\!]}_m & := & \mathbf{lfp}\ \lambda(x).\ x \sqcup \overline{\mathsf{init\text{-}States}}(e) \\ & & \qquad \sqcup\ (\gamma^{\overline{\Sigma} \leftrightarrow \overline{m}}(\overline{\mathsf{step}^{\mathsf{m}}}))(x) \end{array}$$

(d) Collecting semantics

Figure 3.5: λ_S monadic concrete interpreter

$t \in \widehat{Time} := Exp^{*k}$
$\mathsf{tick}(\langle e, t\rangle) := \lfloor e :: t \rfloor_k$

$\mathbb{Z}^\infty := \mathbb{Z} \cup \{-\infty, \infty\}$
$v \in \widehat{Val} := \mathcal{P}(\widehat{Clo}) \times ((\mathbb{Z}^\infty \times \mathbb{Z}^\infty) \cup \{\bot\})$
$\mathsf{int\text{-}I}(i) := \{\langle \varnothing, \langle i, i\rangle\rangle\}$
$\mathsf{if0\text{-}E}(v) := \{\mathbf{tt} \mid v.2 \neq \bot \wedge (v.2).1 \leqslant 0 \leqslant (v.2).2\}$
$\quad \cup \{\mathbf{ff} \mid v.2 = \bot \vee (v.2).1 \neq 0 \vee (v.2).2 \neq 0)\}$
$\mathsf{clo\text{-}I}(c) := \{\langle c, \bot\rangle\}$
$\mathsf{clo\text{-}E}(v) := \{c \mid c \in v.1\}$
$\widehat{[\![+]\!]}_{ms}(\langle v, v'\rangle) := \langle v.1 \cup v'.1, \langle v.2.1 + v'.2.1, v.2.2 + v'.2.2\rangle\rangle$
$\widehat{[\![-]\!]}_{ms}(\langle v, v'\rangle) := \langle v.1 \cup v'.1, \langle v.2.1 - v'.2.2, v.2.2 - v'.2.1\rangle\rangle$

$a \in \widehat{Addr} := Var \times \widehat{Time}$
$\rho \in \widehat{Env} := Var \rightharpoonup \widehat{Addr}$
$s \in \widehat{Store} := \widehat{Addr} \rightharpoonup \widehat{Val}$
$kf \in \widehat{KFrame} := Frame \times \widehat{Env}$
$ka \in \widehat{KAddr} := \widehat{Time}$
$ks \in \widehat{KStore} := \widehat{KAddr} \rightharpoonup \mathcal{P}(\widehat{KFrame} \times \widehat{KAddr})$
$c \in \widehat{Clo} ::= \langle \boldsymbol{\lambda}(x).e, \rho\rangle$

$\psi \in \hat{\Psi} := \widehat{Env} \times \widehat{Store} \times \widehat{KAddr} \times \widehat{KStore} \times \widehat{Time}$
$\sigma \in \hat{\Sigma} := \mathcal{P}(Exp^- \times \hat{\Psi})$
$\widehat{\mathsf{init\text{-}States}}(e) := \alpha(\overline{\mathsf{init\text{-}States}}(e))$

$\alpha^{\hat{\Sigma} \leftrightarrow \hat{m}} : (\hat{\Sigma} \to \hat{\Sigma}) \to (Exp^- \to \hat{m}(Exp^-))$
$\alpha^{\hat{\Sigma} \leftrightarrow \hat{m}}(f)(e)(\psi) := f(\{\langle e, \psi\rangle\})$
$\gamma^{\hat{\Sigma} \leftrightarrow \hat{m}} : (Exp^- \to \hat{m}(Exp^-)) \to (\hat{\Sigma} \to \hat{\Sigma})$
$\gamma^{\hat{\Sigma} \leftrightarrow \hat{m}}(f)(\hat{\sigma}) := \bigcup_{\langle e, \psi\rangle \in \hat{\sigma}} f(e)(\psi)$

(a) Type definitions

$\sqsubseteq : \widehat{\Sigma} \times \widehat{\Sigma} \to Bool$
$\widehat{\sigma} \sqsubseteq \widehat{\sigma'} :=$
$\mathbf{if}(\forall \sigma \in \widehat{\sigma}. \exists \sigma' \in \widehat{\sigma'}.\ \sigma \,\tilde{\sqsubseteq}\, \sigma')\ \mathbf{then\ tt\ else\ ff}$

$\tilde{\sqsubseteq} : (Exp^- \times \hat{\Psi}) \times (Exp^- \times \hat{\Psi}) \to Bool$
$\langle e, \rho, s, ka, ks, t\rangle \,\tilde{\sqsubseteq}\, \langle e', \rho', s', ka', ks', t'\rangle :=$
$\quad \mathbf{if} \begin{pmatrix} e = e' \wedge ka = ka' \wedge t = t' \wedge \rho = \rho' \\ \wedge\ (\forall a \in s.s(a) \sqsubseteq s'(a)) \\ \wedge\ (\forall ka \in ks.ks(ka) \subseteq ks'(ka)) \end{pmatrix}$
$\quad \mathbf{then\ tt\ else\ ff}$

$\sqcup : \widehat{\Sigma} \times \widehat{\Sigma} \to \widehat{\Sigma}$
$\widehat{\sigma} \sqcup \widehat{\sigma'} := \widehat{\sigma} \cup \widehat{\sigma'}$

$\bot : \widehat{\Sigma} := \varnothing$
$\top : \widehat{\Sigma} := \top$

(b) Lattice operations for $\widehat{\Sigma}$

Figure 3.6: λ_S monadic abstract interpreter for interval analysis

$\widehat{m}(A) := \widehat{\Psi} \to \mathcal{P}(A \times \widehat{\Psi})$

$\mathsf{return}(x)(\psi) := \{\langle x, \psi\rangle\}$

$\mathsf{bind}(X)(f)(\psi) := \bigcup_{\langle x,\psi'\rangle \in X(\psi)} f(x)(\psi')$

$\mathsf{get\text{-}Env}(\langle \rho, s, ka, ks, t\rangle) := \{\langle \rho, \langle \rho, s, ka, ks, t\rangle\rangle\}$

$\mathsf{put\text{-}Env}(\rho')(\langle \rho, s, ka, ks, t\rangle) := \{\langle 1, \langle \rho', s, ka, ks, t\rangle\rangle\}$

$\mathsf{get\text{-}Store}(\langle \rho, s, ka, ks, t\rangle) := \{\langle s, \langle \rho, s, ka, ks, t\rangle\rangle\}$

$\mathsf{put\text{-}Store}(s')(\langle \rho, s, ka, ks, t\rangle) := \{\langle 1, \langle \rho, s', ka, ks, t\rangle\rangle\}$

$\mathsf{get\text{-}KAddr}(\langle \rho, s, ka, ks, t\rangle) := \{\langle ka, \langle \rho, s, ka, ks, t\rangle\rangle\}$

$\mathsf{put\text{-}KAddr}(ka')(\langle \rho, s, ka, ks, t\rangle) := \{\langle 1, \langle \rho, s, ka', ks, t\rangle\rangle\}$

$\mathsf{get\text{-}KStore}(\langle \rho, s, ka, ks, t\rangle) := \{\langle ks, \langle \rho, s, ka, ks, t\rangle\rangle\}$

$\mathsf{put\text{-}KStore}(ks')(\langle \rho, s, ka, ks, t\rangle) := \{\langle 1, \langle \rho, s, ka, ks', t\rangle\rangle\}$

$\mathsf{get\text{-}Time}(\langle \rho, s, ka, ks, t\rangle) := \{\langle t, \langle \rho, s, ka, ks, t\rangle\rangle\}$

$\mathsf{put\text{-}Time}(t')(\langle \rho, s, ka, ks, t\rangle) := \{\langle 1, \langle \rho', s, ka, ks, t'\rangle\rangle\}$

$\mathsf{mzero}(\psi) := \{\}$

$(X_1 \langle + \rangle X_2)(\psi) := X_1(\psi) \sqcup X_2(\psi)$

(c) Monad definition

$\alpha : \overline{\Sigma} \to \widehat{\Sigma}$
$\alpha(\overline{\sigma}) := \{\alpha(\sigma) \mid \sigma \in \overline{\sigma}\}$

$\alpha : Exp^- \times \overline{\Psi} \to Exp^- \times \widehat{\Psi}$
$\alpha(\langle e, \rho, s, ka, ks, t\rangle) := \langle e, \alpha(\rho), \alpha(s), \alpha(ka), \alpha(ks), \alpha(t)\rangle$

$\alpha : \overline{Env} \to \widehat{Env}$
$\alpha(\rho) := \{\langle x, \langle x, \lfloor \rho(x).2 \rfloor_k \rangle\rangle \mid x \in \rho\}$

$\alpha : \overline{Store} \to \widehat{Store}$
$\alpha(s) := \lambda(\widehat{a}).\ \bigsqcup_{\alpha(a)=\widehat{a} \wedge a \in s} \alpha(s(a))$

$\alpha : \overline{KAddr} \to \widehat{KAddr}$
$\alpha(ka) := \lfloor ka \rfloor_k$

$\alpha : \overline{KStore} \to \widehat{KStore}$
$\alpha(ks) := \lambda(\widehat{ka}).\ \bigcup_{\alpha(ka)=\widehat{ka} \wedge ka \in ks} \alpha(ks(ka))$

$\alpha : \overline{Time} \to \widehat{Time}$
$\alpha(t) := \lfloor t \rfloor_k$

(d) Abstraction map α from $\overline{\Sigma}$ to $\widehat{\Sigma}$

$$\begin{array}{lcl} \widehat{[\![\cdot]\!]}_m & : & Exp^- \to \widehat{\Sigma} \\ \widehat{[\![e]\!]}_m & := & \mathbf{lfp}\ \lambda(x).\ x \sqcup \widehat{\mathsf{init\text{-}States}}(e) \\ & & \qquad \sqcup\ (\gamma^{\widehat{\Sigma} \leftrightarrow \widehat{m}}(\widehat{\mathsf{step^m}}))(x) \end{array}$$

(e) Abstract semantics

Figure 3.6: λ_S monadic abstract interpreter for interval analysis

set of available addresses, and this is achieved by restricting $\widehat{Time}$ to sequences of upto k expressions (with the set of syntactic expressions contained in a program being finite), as opposed to sequences of unbounded length for concrete interpreters. The notation $\lfloor l \rfloor_k$ refers to the first k elements of the list l. Values ($\widehat{Val}$) are defined as a pair of a set of closures and an integer interval. Note that we extend the set of integers $\mathbb{Z}$ to $\mathbb{Z}^\infty$ that includes $\{-\infty, \infty\}$. The top element of the set of intervals is defined as $\langle -\infty, \infty \rangle$ while bottom is defined by a special element $\perp$. The reason for defining values as pairs of closures and intervals is that, due to the finite number of addresses available in the store, it is possible for a particular location to be mapped to values of both these types. The operations defined for $\widehat{Val}$ are self-explanatory though we make a quick comment on notation - the projection of the i^{th} element of a tuple t is written as $t.i$, with indices starting from 1. All the other definitions in Figure 3.6a are straightforward. Note that the abstract version of init-States ($\widehat{\text{init-States}}$) applies the abstraction map α, defined in Figure 3.6d to set of initial states constructed by $\overline{\text{init-States}}$. In the abstract setting, this set of initial states only contains a single element, irrespective of whether the expression is closed, or if it has free variables of type $\mathbb{Z}$.

Figure 3.6b defines the lattice operations for $\widehat{\Sigma}$. An element $\widehat{\sigma}$ of $\widehat{\Sigma}$ is itself a set of abstract states. The lattice order operation ($\sqsubseteq$) is defined such that $\widehat{\sigma}$ is "less than" $\widehat{\sigma'}$ if for every element $\sigma \in \widehat{\sigma}$, there exists at least one element $\sigma' \in \widehat{\sigma'}$ such that $\sigma \;\tilde{\sqsubseteq}\; \sigma'$. $\tilde{\sqsubseteq}$ is itself defined such that σ is "less than" σ' if and only if σ and σ' the expression, the environment, the address of the next continuation, and the time components are the same, and for every address in σ's store s that is mapped to a value v, the same address in store s' in σ' is mapped to a value v' that is at least as large as v, and similarly, for every address in σ's continuation store ks that is mapped to a set X of continuations, the same address in store ks' in σ' is mapped to a set X' that is equal to X or a superset of X. A comment on the notation - we do not use distinct symbols to represent the lattice operations for different lattices, but the lattice being considered should be clear from the context. The bottom of $\widehat{\Sigma}$

lattice is just the empty set while the top is defined by the special element $\top$.

Figure 3.6c defines the monad for the abstract interpreter and Figure 3.6d defines the abstraction map from $\overline{\Sigma}$ to $\widehat{\Sigma}$. The abstract version of an element $\overline{\sigma} \in \overline{\Sigma}$ is obtained by abstracting each element $\sigma \in \overline{\sigma}$. In the same way that we do not notationally distinguish between lattice operations for different lattices, we do not notationally distinguish between the different abstraction operations, but the types involved should be clear from the context. As one would expect, abstracting a concrete abstract machine state involves abstracting every element of the state tuple. Environment abstraction requires abstracting the addresses that variables are mapped to. These addresses are pairs of variable names and times, and an abstract version of time t requires truncating the sequence of expressions to the latest k expressions. A value store is abstracted by first abstracting the addresses in the store, and then joining all the values that map to the same address. Similarly, continuation stores are abstracted by abstracting the addresses, and then taking a union of all the sets of continuations that map to the same address. Finally, the abstract semantics of a program in λ_S are defined as the least fixed point of a function that uses the abstract versions of the $\mathsf{init\text{-}States}$ and the $\mathsf{step}^{\mathsf{m}}$ functions, i.e., $\widehat{\mathsf{init\text{-}States}}$ and $\widehat{\mathsf{step}^{\mathsf{m}}}$.

We next state two propositions relating the λ_S monadic abstract interpreter to the monadic concrete interpreter. As is typical in the theory of abstract interpretation, we would like to state that the abstract interpreter is sound with respect to the concrete interpreter.

Proposition 14. *(Soundness of $\widehat{\mathsf{step}^{\mathsf{m}}}$ with respect to $\overline{\mathsf{step}^{\mathsf{m}}}$)*
$\forall \overline{\sigma} \in \overline{\Sigma}.\ \alpha((\gamma^{\overline{\Sigma} \leftrightarrow \overline{m}}(\overline{\mathsf{step}^{\mathsf{m}}}))(\overline{\sigma})) \sqsubseteq (\gamma^{\widehat{\Sigma} \leftrightarrow \hat{m}}(\widehat{\mathsf{step}^{\mathsf{m}}}))(\alpha(\overline{\sigma}))$

Proposition 14 relates the concrete $\overline{\mathsf{step}^{\mathsf{m}}}$ function to the abstract $\widehat{\mathsf{step}^{\mathsf{m}}}$ function. In particular, for every element $\overline{\sigma} \in \overline{\Sigma}$, we want the abstraction of the result of applying $\overline{\mathsf{step}^{\mathsf{m}}}$ to $\overline{\sigma}$ to be "less than" the result of applying $\widehat{\mathsf{step}^{\mathsf{m}}}$ to $\alpha(\overline{\sigma})$. This proposition does not directly relate the concrete and abstract semantics of λ_S (which involve computing least fixed points), but it can help us prove the soundness of $\widehat{[\![\cdot]\!]}_m$ with respect to $\overline{[\![\cdot]\!]}_m$. We can prove this result by performing a case analysis on the structure of λ_S expressions, where

the cases are the same as that considered by $\mathsf{step}^{\mathsf{m}}$. We do not present the proof here.

Proposition 15 states the soundness relationship between the abstract and the concrete semantics. In particular, for every λ_S program $e \in Exp^{-}$, it states that the result of abstracting the meaning of the program, as defined by the concrete semantics $\overline{[\![e]\!]}_m$, is less than the meaning defined by the abstract semantics $\widehat{[\![e]\!]}_m$. In other words, the program semantic invariant computed using $\widehat{[\![\cdot]\!]}_m$ can be safely used in proofs of program correctness.

Proposition 15. *(Soundness of $\widehat{[\![\cdot]\!]}_m$ with respect to $\overline{[\![\cdot]\!]}_m$)*
$\forall e \in Exp^{-}.\ \alpha(\overline{[\![e]\!]}_m) \sqsubseteq \widehat{[\![e]\!]}_m$

Proof. We only present an informal proof sketch. From a proof of proposition Proposition 14, a proof of this proposition can be constructed in standard manner using the fixed point transfer theorem from [99]. In particular, the monotonicity of the functions $\lambda(x).\ x \sqcup \widehat{\mathsf{init\text{-}States}}(e) \sqcup (\gamma^{\hat{\Sigma}\leftrightarrow\hat{m}}(\widehat{\mathsf{step}^{\mathsf{m}}}))(x)$ and $\lambda(x).\ x \sqcup \overline{\mathsf{init\text{-}States}}(e) \sqcup (\gamma^{\overline{\Sigma}\leftrightarrow\overline{m}}(\overline{\mathsf{step}^{\mathsf{m}}}))(x)$, combined with proposition Proposition 14 and the Knaster–Tarski theorem fixed point theorem yields the required result.

■

3.4 Observational Abstract Interpreters

In the previous sections, we have defined the language $\lambda_S(\lambda_{SA})$, and have discussed the construction of a monadically-structured interpreter for λ_S. This monadic interpreter is parameterized, i.e., the types of data accessed by the interpreter are defined using typeclass-like constructions. By suitably instantiating these typeclasses, one can recover the concrete semantics of the language. Additionally, one can also instantiate these typeclasses to yield an abstract interpreter (in our case, an abstract interpreter capable of interval analysis). From the perspective of proofs about program judgments, the monadic interpreter is a meta-theoretic construction for computing semantic program invariants. The abstract semantics or abstract meaning of a program, computed with a monadic abstract interpreter, is

$\mathring{Obs}$

$\mathring{m}_o$
return : $\forall A.A \to \mathring{m}_o(A)$
bind : $\forall A, B.\mathring{m}_o(A) \to (A \to \mathring{m}_o(B)) \to \mathring{m}_o(B)$
get-Env : $\mathring{m}_o(Env)$
put-Env : $Env \to \mathring{m}_o(1)$
get-Store : $\mathring{m}_o(\mathring{Store})$
put-Store : $\mathring{Store} \to \mathring{m}_o(1)$
get-KAddr : $\mathring{m}_o(\mathring{KAddr})$
put-KAddr : $\mathring{KAddr} \to \mathring{m}_o(1)$
get-KStore : $\mathring{m}_o(\mathring{KStore})$
put-KStore : $\mathring{KStore} \to \mathring{m}_o(1)$
get-Time : $\mathring{m}_o(\mathring{Time})$
put-Time : $\mathring{Time} \to \mathring{m}_o(1)$
mzero : $\forall A.\mathring{m}_o(A)$
$\cdot\langle+\rangle\cdot$: $\forall A.\mathring{m}_o(A) \times \mathring{m}_o(A) \to \mathring{m}_o(A)$
obs-Store : $Exp^- \times Var \times \mathring{Val} \times \mathring{Obs} \to \mathring{m}_o(\mathring{Val})$

$\alpha^{\mathring{\Sigma} \leftrightarrow \mathring{m}_o} : (\mathring{Obs} \to (\mathring{\Sigma} \to \mathring{\Sigma})) \to (\mathring{Obs} \to (Exp^- \to \mathring{m}(Exp^-)))$
$\gamma^{\mathring{\Sigma} \leftrightarrow \mathring{m}_o} : (\mathring{Obs} \to (Exp^- \to \mathring{m}(Exp^-))) \to (\mathring{Obs} \to (\mathring{\Sigma} \to \mathring{\Sigma}))$

(a) Type definitions

$\mathsf{step}^{\mathsf{m}}_o : Exp^- \times Obs \to \mathring{m}_o(Exp^-)$
$\mathsf{step}^{\mathsf{m}}_o(e) :=$ **do**
  $\rho \leftarrow$ get-Env
  $e'' \leftarrow$ **case** e **of**
    $e_1 \odot e_2 \to \mathsf{tick}^{\mathsf{m}}(e); \mathsf{push}(\langle \square \odot e_2, \rho \rangle); \mathsf{return}(e_1)$
    $\mathbf{if0}(e_1)\{e_2\}\{e_3\} \to$ **do**
      $\mathsf{tick}^{\mathsf{m}}(e); \mathsf{push}(\langle \mathbf{if0}(\square)\{e_2\}\{e_3\}, \rho \rangle); \mathsf{return}(e_1)$
    $a \to$ **do**
      $v \leftarrow [\![a]\!]_{mA}; fr \leftarrow$ pop
      case fr **of**
        $\langle \square \odot e', \rho' \rangle \to$ **do**
          $\mathsf{tick}^{\mathsf{m}}(e); \mathsf{put\text{-}Env}(\rho'); \mathsf{push}(\langle v \odot \square, \rho \rangle); \mathsf{return}(e')$
        $\langle v'@\square, \rho' \rangle \to$ **do**
          $\mathsf{tick}^{\mathsf{m}}(e); t \leftarrow$ get-Time; $s \leftarrow$ get-Store
          $\langle \boldsymbol{\lambda}(x).e', \rho'' \rangle \leftarrow \uparrow_p(\mathsf{clo\text{-}E}(v'))$
          $\mathsf{put\text{-}Env}(\rho''[x \mapsto \langle x, t \rangle])$
          $v' \leftarrow \mathsf{obs\text{-}Store}(\langle e', x, v, o \rangle)$
          $\mathsf{put\text{-}Store}(s \sqcup [\langle x, t \rangle \mapsto v']); \mathsf{return}(e')$
        $\langle v' \oplus \square, \rho' \rangle \to \mathsf{tick}^{\mathsf{m}}(e); \mathsf{return}([\![\oplus]\!]_{m\delta}(\langle v', v \rangle))$
        $\langle \mathbf{if0}(\square)\{e_1\}\{e_2\}, \rho' \rangle \to$ **do**
          $\mathsf{tick}^{\mathsf{m}}(e); \mathsf{put\text{-}Env}(\rho'); b \leftarrow \uparrow_p(\mathsf{if0\text{-}E}(v)); \mathsf{refine}(\langle a, b \rangle)$
          $\mathbf{if}(b)$ **then** return(e_1) **else** return(e_2)
        $\bot \to \mathsf{return}(e)$
  return(e'')

(b) Step function

$$
\begin{array}{rcl}
[\![\cdot]\!]_{mO} & : & Exp^- \times Obs \to \mathring{\Sigma}_o \\
[\![e]\!]_{mO}(o) & := & \mathbf{lfp}\ \lambda(x).\ x \sqcup \mathsf{init\text{-}States}(e) \sqcup (\gamma^{\mathring{\Sigma} \leftrightarrow \mathring{m}_o}(\mathsf{step}^{\mathsf{m}}_O))(x)(o)
\end{array}
$$

(c) Collecting semantics

Figure 3.7: λ_S observational interpreter

a semantic invariant of the program and a simplified representation (informally, containing lesser information) of the concrete program meaning.

We want to combine semantic invariant based reasoning, with reasoning hypothetically about program via run time/dynamic checks. Moreover, we want to use observations about program behavior for inferring the hypotheses. We achieve this by extending the monadic interpreter design, proposing a new meta-theoretic construction for reasoning about programs, that we refer to as *observational abstract interpreters*. There are two main reasons motivating our construction of observational abstract interpreters: (i) past work has investigated various combinations of invariant-based reasoning, hypothetical reasoning, and observational reasoning about programs, but there has been an absence of formal investigation of approaches combining these three reasoning styles. A precise formulation of a combined approach can bring greater clarity about the design space of algorithms for finding proofs of program judgments. (ii) hypothetical reasoning about programs using observations about their behavior can help us focus the program proof effort towards the observed or common program behaviors, potentially making the the search for program proofs cheaper, at the cost of dynamic/run time checks.

In Figure 3.7, we present an observational abstract interpreter for λ_S. This interpreter is monadically structured, and designed such that while the semantic invariant, i.e., the program semantics, is being computed, the interpreter can read data representing observations about program behavior, use these observations to make hypotheses about program behavior, and accordingly update the state of the interpreter. Moreover, the validity of these hypotheses is not checked statically, and instead, we embed dynamic checks in to the λ_S programs, producing λ_{SA} programs. In the process of designing an observational abstract interpreters, following are the main questions that we were forced to address:

- What is the form of the observational data about programs? What aspects of program behavior does it capture?

- How do we infer the hypotheses using the observational data? Moreover, how do we

avoid inferring too many hypotheses/dynamic checks, and how do we ensure that the inferred hypotheses are not overly restrictive, such that the program fails to satisfy the dynamic checks in most cases?

- How do we translate the hypotheses in to dynamic checks embedded in the program?

For the first question, the observational abstract interpreter design in Figure 3.7 assumes that the observational data is drawn from the collecting semantics, i.e., the set of reachable abstract machine states, of a λ_S program. However, the exact form of the observations is left unspecified (we give a specific definition for the observational interval analysis defined in Figure 3.8). Using observations about program inputs in order to infer program pre-conditions is not uncommon [100], but our design allows observations at any program point, and about any component of the abstract machine. In Figure 3.7, the typeclass $\mathring{Obs}$, with no constraints, represent the types of observations. We use the blue background to highlight parts of the observational interpreter design that are unique. We do not show any type definitions besides the monad typeclasses and elide the helper functions because these are similarly to the definitions in Figure 3.4.

To address the second question, we extend the monad typeclass with the operation obs-Store as shown in Figure 3.7a. This operation requires that a 4-tuple comprising of the current expression being evaluated, a variable name, the value associated with the variable, and the observational data is passed as an argument. We also modify $\mathsf{step}^{\mathsf{m}}$ such that whenever the term in the argument position of a function application is evaluated to a value and the next step of the evaluation is to actually apply the function to this value, the observational interpreter first invokes the obs-Store operation with the name of the argument (say x) and it's evaluated value (say v). Next, instead of substituting x with v in the function, we substitute it with the value (say v') returned by obs-Store (say v'). The hypothesis that the value of x is v' instead of v is the only form of hypothesis that the observational interpreter is allowed to make. The mechanism for computing v' is hidden behind the monadic interface. Notice that the observational data is passed as an argument

to the $\mathsf{step}^{\mathsf{m}}_{O}$ function. This is also forces us to change the type definitions of $\alpha^{\mathring{\Sigma}\leftrightarrow\mathring{m}_o}$ and $\gamma^{\mathring{\Sigma}\leftrightarrow\mathring{m}_o}$ (Figure 3.7a) that map between the monadic step function and transition function of type $\mathring{\Sigma} \to \mathring{\Sigma}$. Moreover, the observational collecting semantics ($[\![\cdot]\!]_{mO}$) are also modified to accept observations as an argument.

We address the third question in the specific context of an observational abstract interpreter for an interval analysis of λ_S programs in the next section.

3.4.1 Observational Interval Analysis for λ_S

We instantiate the generic observational interpreter for λ_S so as to yield an observational abstract interpreter for interval analysis of λ_S programs as described in Figure 3.8. The type definitions are presented in Figure 3.8a. Notice that observations (Obs) are defined as a partial map from labels to partial maps from variables names to sets of values. We assume that program observations are recorded at the granularity of syntactic program expressions, explicitly identified by their labels. Moreover, for each expression we can record a set of observed values for any variable in scope. We also assume that only the values of type $\mathbb{Z}$ are recorded. Extending this approach to with observations of higher-order values is an interesting direction for future work. The observational abstract interpreter computes an element $\widehat{\sigma}_o \in \widehat{\Sigma}_o$, where each element $\widehat{\sigma}_o$ is a pair of a set of abstract machine states and the hypotheses map. A hypotheses map h is a partial map from labels to partial maps from variables names to abstract values. Intuitively, the interpreter can make hypotheses at the granularity of syntactic program expressions. At each program expressions, hypotheses can made about the abstract values of the variables in scope. Though the type of hypotheses maps ($Lbl \rightharpoonup Var \rightharpoonup \widehat{Val}$) allows assumptions about higher-order values, the observational abstract interpreter defined here only makes assumptions on $\mathbb{Z}$ values. Initially, the hypotheses map is assumed to be $\bot$, as the definition of $\widehat{\mathsf{init\text{-}States}}_o$ shows.

Figure 3.8b defines the lattice operations for the lattice $\widehat{\Sigma}_o$. We draw notice to the definitions of the lattice operations for the hypotheses map. A hypotheses map h is "less

$o \in Obs := Lbl \rightharpoonup (Var \rightharpoonup \mathcal{P}(\mathbb{Z}))$
$h \in Hyp := Lbl \rightharpoonup (Var \rightharpoonup \widehat{Val})$
$\psi \in \widehat{\Psi} := \widehat{Env} \times \widehat{Store} \times \widehat{KAddr} \times \widehat{KStore} \times \widehat{Time}$
$\widehat{\sigma}_o \in \widehat{\Sigma}_o := \mathcal{P}(Exp^- \times \widehat{\Psi}) \times Hyp$
$\widehat{\mathsf{init\text{-}States}}_o(e) := \langle \alpha(\widehat{\mathsf{init\text{-}States}}(e)), \bot \rangle$

$\alpha^{\widehat{\Sigma} \leftrightarrow \widehat{m}_o} : (Obs \to (\widehat{\Sigma}_o \to \widehat{\Sigma}_o)) \to (Obs \to (Exp^- \to \widehat{m}_o(Exp^-)))$
$\alpha^{\widehat{\Sigma} \leftrightarrow \widehat{m}_o}(f)(o)(e)(\langle \psi, h \rangle) := f(o)(\langle \{\langle e, \psi \rangle\}, h \rangle)$
$\gamma^{\widehat{\Sigma} \leftrightarrow \widehat{m}_o} : (Obs \to (Exp^- \to \widehat{m}_o(Exp^-))) \to (Obs \to (\widehat{\Sigma}_o \to \widehat{\Sigma}_o))$
$\gamma^{\widehat{\Sigma} \leftrightarrow \widehat{m}_o}(f)(o)(\widehat{\sigma}_o) := \textbf{let } \langle X, h \rangle := \widehat{\sigma}_o \textbf{ in}$
$\quad \langle \bigcup_{\langle e,\psi \rangle \in X} f(o)(e)(\langle \psi, h \rangle).1, \bigsqcup_{\langle e,\psi \rangle \in X} f(o)(e)(\langle \psi, h \rangle).2 \rangle$

(a) Type definitions

$\sqsubseteq : \widehat{\Sigma}_o \times \widehat{\Sigma}_o \to Bool$
$\widehat{\sigma}_o \sqsubseteq \widehat{\sigma'}_o :=$
$\textbf{if}((\forall \sigma \in \widehat{\sigma}_o.1. \exists \sigma' \in \widehat{\sigma'}_o.1.\ \sigma \mathrel{\tilde{\sqsubseteq}}_o \sigma') \wedge (\widehat{\sigma}_o.2 \sqsubseteq \widehat{\sigma'}_o.2))$
$\textbf{then tt else ff}$

$\tilde{\sqsubseteq}_o : (Exp^- \times \widehat{\Psi}) \times (Exp^- \times \widehat{\Psi}) \to Bool$
$\tilde{\sqsubseteq}_o := \tilde{\sqsubseteq}$

$\sqsubseteq : Hyp \times Hyp \to Bool$
$h \sqsubseteq h' := \textbf{if}(\forall l \in h. \forall x \in h(l). h(l)(x) \sqsupseteq h'(l)(x))$
$\qquad \textbf{then tt else ff}$

$\sqcup : \widehat{\Sigma}_o \times \widehat{\Sigma}_o \to \widehat{\Sigma}_o$
$\widehat{\sigma}_o \sqcup \widehat{\sigma'}_o := \langle \widehat{\sigma}_o.1 \cup \widehat{\sigma'}_o.1, \widehat{\sigma}_o.2 \sqcup \widehat{\sigma'}_o.2 \rangle$

$\sqcup : Hyp \times Hyp \to Hyp$
$h \sqcup h' :=$
$\textbf{let } f := (\lambda(x).\ \textbf{if}(x \in h(l) \wedge x \in h'(l))$
$\quad \textbf{then if}(h(l)(x) = h'(l)(x)) \textbf{ then } h(l)(x) \textbf{ else } \bot$
$\quad \textbf{else if}(x \in h(l)) \textbf{ then } h(l)(x) \textbf{ else } h'(l)(x)) \textbf{ in}$
$\textbf{let } g = (\lambda(x).\ \textbf{if}(l \in h \wedge x \in h(l))$
$\quad \textbf{then } h(l)(x) \textbf{ else if}(l \in h' \wedge x \in h'(l)) \textbf{ then } h'(l)(x)) \textbf{ in}$
$\{f \mid l \in h \wedge l \in h'\} \cup \{g \mid l \in h \mathbin{\mathsf{xor}} l \in h'\}$

$\bot : \widehat{\Sigma}_o := \langle \varnothing, \bot \rangle$
$\top : \widehat{\Sigma}_o := \langle \top, \top \rangle$

(b) Lattice operations for $\widehat{\Sigma}_o$

Figure 3.8: λ_S observational abstract interpreter for interval analysis

$$\widehat{m}_o(A) := \widehat{\Psi} \times Hyp \to \mathcal{P}(A \times \widehat{\Psi}) \times Hyp$$

$$\begin{array}{l}
\mathsf{obs\text{-}Store} : Exp^- \times Var \times \widehat{Val} \times Obs \to \widehat{m}_o(\widehat{Val}) \\
\mathsf{obs\text{-}Store}(\langle e, x, v, o\rangle)(\langle \psi, h\rangle) := \\
\textbf{if}(\mathsf{get\text{-}Label}(e) \in h \wedge x \in h(\mathsf{get\text{-}Label}(e))) \textbf{ then } \{ \\
\quad \langle \{\langle\langle v.1, (h(\mathsf{get\text{-}Label}(e))(x)).2\rangle, \psi\rangle\}, h\rangle \\
\} \textbf{ else if}((\mathsf{get\text{-}Label}(e) \in o) \wedge (x \in o(\mathsf{get\text{-}Label}(e)))) \textbf{ then}\{ \\
\quad \textbf{let } v_O := \alpha(o(\mathsf{get\text{-}Label}(e))(x)) \textbf{ in} \\
\quad \textbf{let } \mathsf{distance} := d(v.2, v_O) \textbf{ in} \\
\quad \textbf{if}(\mathsf{distance} \geqslant \omega \wedge v_O \sqsubseteq v.2) \textbf{ then}\{ \\
\quad\quad \langle \{\langle\langle v.1, v_O\rangle, \psi\rangle\}, h \sqcup [\mathsf{get\text{-}Label}(e) \mapsto [x \mapsto \langle \varnothing, v_O\rangle]]\rangle \\
\quad \} \textbf{ else } \langle \{\langle v, \psi\rangle\}, h\rangle \\
\} \textbf{ else } \langle \{\langle v, \psi\rangle\}, h\rangle
\end{array}$$

$$\mathsf{return}(x)(\langle \psi, h\rangle) := \langle \{\langle x, \psi\rangle\}, h\rangle$$

$$\begin{array}{l}
\mathsf{bind}(X)(f)(\langle \psi, h\rangle) := \textbf{let } \langle Y, h'\rangle := X(\langle \psi, h\rangle) \textbf{ in} \\
\qquad\qquad \bigcup_{\langle x, \psi'\rangle \in Y} f(x)(\langle \psi', h'\rangle)
\end{array}$$

$$\mathsf{get\text{-}Env}(\langle\langle \rho, s, ka, ks, t\rangle, h\rangle) := \langle \{\langle \rho, \langle \rho, s, ka, ks, t\rangle\rangle\}, h\rangle$$

$$\mathsf{put\text{-}Env}(\rho')(\langle\langle \rho, s, ka, ks, t\rangle, h\rangle) := \langle \{\langle 1, \langle \rho', s, ka, ks, t\rangle\rangle\}, h\rangle$$

$$\mathsf{get\text{-}Store}(\langle\langle \rho, s, ka, ks, t\rangle, h\rangle) := \langle \{\langle s, \langle \rho, s, ka, ks, t\rangle\rangle\}, h\rangle$$

$$\mathsf{put\text{-}Store}(s')(\langle\langle \rho, s, ka, ks, t\rangle, h\rangle) := \langle \{\langle 1, \langle \rho, s', ka, ks, t\rangle\rangle\}, h\rangle$$

$$\mathsf{get\text{-}KAddr}(\langle\langle \rho, s, ka, ks, t\rangle, h\rangle) := \langle \{\langle ka, \langle \rho, s, ka, ks, t\rangle\rangle\}, h\rangle$$

$$\mathsf{put\text{-}KAddr}(ka')(\langle\langle \rho, s, ka, ks, t\rangle, h\rangle) := \langle \{\langle 1, \langle \rho, s, ka', ks, t\rangle\rangle\}, h\rangle$$

$$\mathsf{get\text{-}KStore}(\langle\langle \rho, s, ka, ks, t\rangle, h\rangle) := \langle \{\langle ks, \langle \rho, s, ka, ks, t\rangle\rangle\}, h\rangle$$

$$\mathsf{put\text{-}KStore}(ks')(\langle\langle \rho, s, ka, ks, t\rangle, h\rangle) := \langle \{\langle 1, \langle \rho, s, ka, ks', t\rangle\rangle\}, h\rangle$$

$$\mathsf{get\text{-}Time}(\langle\langle \rho, s, ka, ks, t\rangle, h\rangle) := \langle \{\langle t, \langle \rho, s, ka, ks, t\rangle\rangle\}, h\rangle$$

$$\mathsf{put\text{-}Time}(t')(\langle\langle \rho, s, ka, ks, t\rangle, h\rangle) := \langle \{\langle 1, \langle \rho', s, ka, ks, t'\rangle\rangle\}, h\rangle$$

$$\mathsf{mzero}(\langle \psi, h\rangle) := \langle \{\}, h\rangle$$

$$(X_1 \langle + \rangle X_2)(\langle \psi, h\rangle) := X_1(\langle \psi, h\rangle) \cup X_2(\langle \psi, h\rangle)$$

(c) Monad definition

$$\begin{array}{rcl}
\alpha & : & \overline{\Sigma} \to \widehat{\Sigma}_o \\
\alpha(\overline{\sigma}) & := & \langle \{\alpha(\sigma) \mid \sigma \in \overline{\sigma}\}, \bot\rangle
\end{array}$$

(d) Abstraction map α from $\overline{\Sigma}$ to $\widehat{\Sigma}_o$

$$\begin{array}{rcl}
\widehat{[\![\cdot]\!]}_{mO} & : & Exp^- \times Obs \to \widehat{\Sigma}_o \\
\widehat{[\![e]\!]}_{mO}(o) & := & \textbf{lfp } \lambda(x).\ x \sqcup \widehat{\mathsf{init\text{-}States}}_o(e) \\
& & \sqcup\ (\gamma^{\widehat{\Sigma} \leftrightarrow \widehat{m}}(\widehat{\mathsf{step}^{\mathsf{m}}_O}))(x)(o)
\end{array}$$

(e) Abstract semantics

Figure 3.8: λ_S observational abstract interpreter for interval analysis

$$
\begin{array}{rcl}
i \in Intrvl & : & \mathbb{Z}^{\infty} \times \mathbb{Z}^{\infty} \\
d & : & Intrvl \times Intrvl \to \mathbb{R} \cup \{-\infty, \infty\} \\
d(\langle i, i' \rangle) & := & \textbf{let } X := \{-\infty, \infty\} \textbf{ in} \\
 & & \textbf{if}(i.1 \in X \vee i.2 \in X \vee i = \bot \vee i'.1 \in X \vee i'.2 \in X \vee i' = \bot) \\
 & & \textbf{then } \infty \\
 & & \textbf{else } \max(\langle |i.1 - i'.1|, |i.2 - i'.2| \rangle)
\end{array}
$$

Figure 3.9: Metric structure on intervals

$$
\begin{array}{rcl}
\mathsf{embed_t} & : & Exp^{-}{}_{\lambda_S} \times (Lbl \times Var \times \widehat{Val}) \to Exp^{-}{}_{\lambda_{SA}} \\
\mathsf{embed_t}(\langle e, \langle l, x, v \rangle \rangle) & := & \textbf{let } v' := v.2 \textbf{ in} \\
 & & \textbf{let } e' := \textbf{if0}(\alpha(x) \sqsubseteq v')\{e\}\{\textbf{abort}\} \textbf{ in} \\
 & & \textbf{if}(\mathsf{get\text{-}Label}(e) = l) \textbf{ then } e' \textbf{ else } e
\end{array}
$$

Figure 3.10: Translation of λ_S programs in to λ_{SA} programs with embedded dynamic checks (assuming that $\sqsubseteq$ returns 1 for $\mathbf{tt}$ and 0 for $\mathbf{ff}$)

than" a hypotheses map h' if for every label and variable for which h includes a hypothesis, h' has a stricter hypotheses, i.e., assumes a narrower interval of $\mathbb{Z}$. The join operation for hypotheses maps h and h' looks messy but the intuition is simple - whenever a hypothesis is defined for one map but not the other, we defer to the map with the definition, but in case both the maps have hypotheses defined for a particular combination of label and variable, then we require the two hypotheses be equal, or the join produces the bottom element of the $\widehat{Val}$ lattice as the joined hypothesis. The bottom element of the Hyp lattice makes no hypotheses whereas the top element of the Hyp lattice makes the strictest possible possible hypothesis for every label and variable.

Figure 3.8c defines the monad $\widehat{m}_o$ for observational abstract interpreters. The only interesting definition is that of obs-Store. The other monad operations are similar to the definition of the monad operations for the monadic abstract interpreter in Figure 3.6e. obs-Store expects a 4-tuple of expression, variable name, value, and the observations ($\langle e, x, v, o \rangle$). It extracts the label of the expression e using the get-Label function, and checks if a hypothesis has been already made for variable x at label t, and if so, it replaces the second element

of v (recall that an abstract value is a pair of a set of closures and an interval) with the hypothesis. In case there is no preexisting hypothesis, and if the observations map includes a set of observed values of x at label t, then the set of observed $\mathbb{Z}$ values is first abstracted to an interval $\widehat{v}_o$ (assumed here to be tightest possible interval abstraction of the set of observed values, though other choices are possible). Next, the distance between the intervals $\widehat{v}$ and $v.2$ is computed. Such a distance computation is possible because we give a metric structure to the set of intervals (defined in Figure 3.9). Finally, if the distance is greater than a fixed constant ω (we expect value of ω to be empirically derived), and if $\widehat{v}_o \sqsubseteq v.2$, then we replace $v.2$ with $\widehat{v}_o$, and update the hypotheses map accordingly.

Figure 3.8d defines the abstraction map α from $\overline{\Sigma}$ to $\widehat{\Sigma}_o$. The abstraction map reuses the definition of the abstraction map from Figure 3.6d for the set of abstract machines states, but the hypotheses map is always assumed to be $\perp$. Finally, the observational abstract semantics, defined in Figure 3.8e take the standard least fixed point form, except that the observations map is expected as an input.

The metric structure on the set of intervals in defined in Figure 3.9. A set X has a metric structure for all elements x, y, z in X, if a function $d(\cdot, \cdot)$ producing a value of type $\mathbb{R}$ is defined for X, such that the following conditions hold true,

- $d(x, y) = 0 \iff x = y$
- $d(x, y) = d(y, x)$
- $d(x, z) \leqslant d(x, y) + d(y, z)$

Finally, Figure 3.10 describes the manner in which a hypothesis can be embedded in a λ_S program. For ease of presentation, we define a function $\mathsf{embed_t}$ that given a program e from the set ${Exp^-}_{\lambda_S}$ of λ_S programs with free variables of type $\mathbb{Z}$, and a triple of a label, variable name, and an abstract value ($\langle l, x, v \rangle$), produces a λ_{SA} program e' with the original expression e wrapped in a dynamic check. We assume here that the abstraction map α and the lattice operation $\sqsubseteq$ are computable functions that can be expressed in λ_{SA}. $\alpha(x)$ is

written assuming that, whenever the labels match, variable x is in the scope of the object program e and a normalized value is bound to x in the environment. The design of λ_{SA} and the definition of $\mathsf{embed_t}$ should be viewed in the same spirit as the design of a cast calculus and the definition of a compiler of programs from a gradually typed surface language to the cast calculus in the gradual typing literature.

Proposition 16 is a formal statement of the notion of soundness for our observational abstract interpreter, relating the observational abstract semantics ($\widehat{[\![\cdot]\!]}_{mO}$) of a λ_S program in Exp^- with the monadic concrete semantics ($\overline{[\![\cdot]\!]}_m$) of the same. Intuitively, the proposition states that for any expression $e \in Exp^-$ and for any observations map o, if we compute the observational abstract semantics of e, producing the pair $\langle \widehat{\sigma}, h \rangle$, then the λ_{SA} program e' obtained by embedding the hypotheses map h in e is such that an abstraction of the computed collecting semantics of e' is less than or equal to $\widehat{\sigma}$ extended with an $\mathbf{abort}$ abstract machine state. Note that $\overline{[\![\cdot]\!]}_m$ here denotes the monadic collecting semantics of λ_{SA} which is exactly the same as for λ_S (in Figure 3.5) except for the $\mathbf{abort}$ operation. Also, the statement here uses embed while Figure 3.10 defines $\mathsf{embed_t}$. $\mathsf{embed}(e, h)$ invokes $\mathsf{embed_t}$ on every subexpression of e, with every triple of a label, variable name, and an abstract value ($\langle l, x, v \rangle$) from h. We skip presenting the formal definition of embed.

Proposition 16. *(Soundness of $\widehat{[\![\cdot]\!]}_{mO}$ with respect to $\overline{[\![\cdot]\!]}_m$)*
$\forall e \in Exp^-, o \in Obs.$
If $\langle \widehat{\sigma}, h \rangle := \widehat{[\![e]\!]}_{mO}(o)$, *then*, $\alpha(\overline{[\![\mathsf{embed}(e,h)]\!]}_m) \sqsubseteq (\widehat{\sigma} \cup \{\langle \mathbf{abort}, \top, \top, \top, \top, \top \rangle\})$

We do not present a proof of proposition Proposition 16 in this paper, though we believe that the observational abstract interpreter in Figure 3.8 is sound with respect to the monadic concrete semantics of λ_S. The proof is challenging primarily because the function $\gamma^{\widehat{\Sigma} \leftrightarrow \widehat{m}_o}(\mathsf{step}^{\mathsf{m}}_O)$ of type $\widehat{\Sigma}_o \to \widehat{\Sigma}_o$ is not monotonic.

3.4.2 Discussion

Why do we expect program observations to help construct program proofs? The effectiveness of program observations in helping construct program proofs can be explained if the following three assumptions hold true: (i) Program environments have statistical information. For instance, consider a program with a free variable of integer type, and imagine that this program is "deployed" as a component of some system. There is some statistical model describing how these integer inputs to the program are generated. (ii) Statistical models of real-world phenomena rarely yield uniform distributions. It is much more common for the probabilistic mass of the distributions to be concentrated in small regions. (iii) We may not know the statistical model describing the program environment but can collect data about this model, i.e., we can observe independent and identically distributed (i.i.d.) samples generated by the model. These i.i.d. samples or observations are either in the form of program inputs, or in the form of subsequent abstract machine states.

If the above assumptions are valid, it is conceivable to make hypotheses, either about program inputs or subsequent abstract machine states, that are true with a high probability with respect to the input statistical model and to compute stronger program invariants under these hypotheses. Since these hypotheses likely to be true, they are unlikely to trigger run time aborts. We do not actually expect to know the input statistical model, but we use the samples or observations from this distribution, in the form of program inputs or subsequent abstract machine states, to infer likely hypotheses.

Comparison against strawmen approaches. We compare the style of program reasoning adopted by observational abstract interpreters with two other strawmen approaches: (i) Most existing abstract interpreters can easily handle hypothetical reasoning by refining the program environment based on observations of program inputs. However, observational abstract interpreters enable a strictly more general style of program reasoning. Since observations can either be in the form of program inputs or in the form of subsequently reachable abstract machine states, the refinements or hypotheses allowed by observational

abstract interpreters can also either be about the program environment or about subsequent reachable abstract machine states. This generality of observational abstract interpreters is useful because it avoids the strong requirement that the program inputs be observable. Instead, partial observability of abstract machine states suffices.

(ii) Instead of designing a sophisticated observational abstract interpreter with the "hypothesis" mechanism, one could just use the embedding mechanism to generate a program with dynamic checks and then run an off-the-shelf abstract interpreter on this program. In other words, this second proposed strawman approach "separates" the process of inferring dynamic checks and the process of computing invariants via an abstract interpreter. In contrast, observational abstract interpreters allow fine-grained intermingling of these two processes, with the information computed by the abstract interpreter guiding the choice of dynamic checks, which in turn affects the invariants computed by the abstract interpreter. This fine-grained intermingling of invariant computation with hypotheses inference allows designing strategies for embedding dynamic checks in the program that can balance the benefits of stronger invariants with the cost of dynamic checks.

Proofs of soundness and termination. Proving Proposition 16 as well proving that observational abstract interpreters terminate is a next step for this work. One of the challenges in these proofs is that replacing computed abstract values with observed abstract values leads to a non-monotonic abstract transformer. We believe that there are conditions that can be imposed on the use of observations that would us allow to bypass the absence of monotonicity and complete these proofs.

3.5 Related Work

There are many threads of work related to the ideas presented in this chapter, and we described some of these connections in section 3.1. In this section, we further elaborate on the use of data (or observations) for constructing program proofs and on embedding dynamic checks in the programs to help construct program proofs.

The program verification community has been exploring ways of combining the standard deductive/symbolic approaches in verification algorithms with ”data-driven” reasoning techniques. Observational abstract interpreters also fall in this bucket. In this section, we briefly survey the space of ”data-driven” verification algorithms, organize this space, and then positioning observational abstract interpreters within this organization.

Say that we are verifying a program p. Then the space of data-driven verification algorithms can be divided into two classes: (i) algorithms that use observations or data about p for constructing program proofs, (ii) algorithms that use data about programs other than p, i.e. use “big code” in the form of repositories of existing programs and associated metadata, for reasoning aboout programs and constructing program proofs. Observational abstract interpreters belong to the first class of algorithms.

Using program observations for program proofs. There is a long history of using observations to make hypotheses about program behavior, and computing semantic invariants under these hypotheses [101, 102, 103, 104, 105, 106, 107, 108, 109]. Our work on observational abstract interpreters formalizes this style of reasoning. A different line of work uses program observations to guide CEGAR algorithms in their search for an appropriate abstract semantics [110, 111, 112]. More recently, with the advances in statistical learning algorithms, a number of techniques have been proposed that eschew the use of abstract interpreters and instead use the observational data to iteratively infer (or learn) candidate invariants that, if confirmed to be invariants (typically using an SMT-like decision procedure), are used to help in the construction of program proofs [113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125]. Program observations have also been used to compute candidate specifications [126, 127, 100, 128] or types of program modules [129, 130].

Using“big code” for program proofs. Using “big code”, i.e., a dataset of programs and corresponding program metadata (like test cases, bug reports, program analysis results, etc.), one can construct statistical models about the nature of programs that humans write, and use these models to help reason about programs. With the rapid advances in compu-

tational statistical modeling and machine learning in recent years, this style of reasoning has become increasingly feasible. We give a small sampling here of the literature on using statistical models for reasoning about programs. Statistical models have been used to, (i) help in the computation of program invariants by aiding CEGAR algorithms in their search for abstract semantics [131], as well as help tune abstract interpreter heuristics [132, 133, 134, 135, 136, 137, 138, 139], (ii) directly compute candidate program invariants or specifications [140, 141, 142, 122], (iii) rank the list of bugs reported by a program analysis tool, in order of the probability of the bug being a true program bug (as opposed to being a false positive) [143, 144, 145] and to allow the use of developer provided feedback in order to update the list of reported bugs [146, 143], (iv) guide the tactics to be used by proof search algorithms [147, 148, 149, 150, 151, 152, 153], (v) infer the likely types or annotations of a program [154, 155, 156, 157], or predict program behaviors [158, 159, 160].

Using dynamic checks for program proofs. The use of dynamic checks as a mechanism to help with static reasoning about programs has been a topic of intense investigation in recent years, particularly in the context of gradual typing [3, 4]. Gradual typing aims to reason about programs written in a mixture of typing disciplines, and employs dynamic checks, wherever necessary, to translate between the different typing discplines. However, the idea of dynamic checks as an aid for type-based reasoning [161, 162, 163, 164, 5, 6, 129] and for computing more precise invariants [165, 166, 109, 167] has been repeatedly used over the last thirty years. In the opposite direction, starting from programs already embedded with dynamic checks or constracts, static reasoning has been used to remove the dynamic checks, if possible and reduce the run time overhead [168, 169, 170, 171, 172, 173, 174, 175, 176, 177].

3.6 Conclusion

We study the proof strategies employed by algorithms that search for proofs of program judgments. We are particularly interested in three broad strategies, namely, computing se-

mantic program invariants, reasoning hypothetically about programs by embedding them with dynamic/run time checks, and using data representing observations about program behavior to help reason about a program. We present a meta-theoretic construction, referred as observational abstract interpreter, that combines these three reasoning strategies. An observational abstract interpreter uses program observations to infer hypotheses about program behavior, and computes hypothetical semantic invariants of the program. These hypotheses are embedded in the program as dynamic checks. Our design of observational abstract interpreters is heavily inspired by the abstracting abstract machines methodology of Van Horn and Might for constructing concrete and abstract interpreters of higher-order languages, and the monadically refactored design of these interpreters. We formalize our ideas in the context of a simple higher-order language (λ_S) with built-in integers. We construct an observational interpreter for interval analysis of λ_S programs.

CHAPTER 4
FUTURE DIRECTIONS

The ideas presented in this dissertation, focused on algorithms for reasoning about programs in statistically modeled environments, can be extended in a number of ways as will be described in this chapter. Additionally, inspired by the broader theme of combining ideas from programming languages theory and from theoretical statistics, we sketch two possible threads of future investigation, namely, algorithmic verification of probabilistic programs and proving generalization guarantees for statistical learning algorithms using tools from programming languages theory.

4.1 Neural Network Verification

We describe some possible extensions to our work on verification of neural networks.

1. Strategies for exploring the latent space: In algorithm 3, we do not describe the manner in which the latent space should be explored. The design of this exploration strategy is a key component controlling the performance of the algorithm. To tackle this problem, one may use reinforcement learning for learning an agent with an exploration strategy. Alternatively, one may analyze the generative model and the neural network under analysis to unearth more information such that an effective strategy maybe designed. However, what additional information might be useful remains an open question.

2. Extending $pcat$ language with loops: The $pcat$ language is unable to express neural network architectures like recurrent neural networks with looping constructions. Extending the languages with a looping construct is easy but verification of programs in such a language becomes challenging with the need to compute loop invariants.

3. Probabilistic robustness with respect to a family of input distributions: Our algorithm for verifying probabilistic robustness of neural networks assumes that the input distribution stays fixed. In other words, we certify a neural network to be probabilistically robust with respect to a specific distribution. In practice, however, a trained neural network might get deployed in settings where the environments are similar but not exactly the same, implying that the input distributions are not the same. Consequently, it can be beneficial to certify probabilistic robustness of a neural network with respect to a family of input distributions. Designing a verification algorithm for this more challenging setting is an interesting direction for future research.

4.2 Observational Abstract Interpreters

Our work on observational abstract interpreters can be extended in a number of ways and some of these are described below.

1. Languages with more features: We have formalized our ideas about observational abstract interpreters in the context of a simple higher-order programming language λ_S. In order to study and construct observational abstract interpreters for widely-used languages, we need to extend our formalization to higher-order languages with realistic features. Moreover, in present work, we only consider programs with first-order inputs and only allow dynamic checks on first-order values. To extend our work to programs with higher-order inputs, one can build on approaches for higher-order abstract interpretation [178, 179]. To allow higher-order dynamic checks, a starting point is to consider the work on higher-order contracts [180].

2. Statistical guarantees about probability of failure: Dynamic checks (or hypotheses) are inferred by observational abstract interpreters using the observed program behavior. The inferred dynamic checks should have a low probability of being violated at run time. Though our hypothesis inference strategy is designed with this goal, at

present, our framework provides no guarantees about the probability of failure of the dynamic checks. We would like extend our work to provide upper bounds on failure probabilities or estimates of the failure probability along with confidence intervals

3. Observational type inference: There is a close relationship between type inference and abstract interpretation [181, 182]. We believe that it would be very interesting to apply the notions of dynamic checks inferred from observations and hypothetical invariants in the setting of type inference. This would additionally make the relationship between our work and gradual type systems explicit.

4.3 Verification of Probabilistic Programs

We would like to construct fully automated algorithms capable of efficiently finding proofs or violations of correctness specifications of probabilistic programs. Previous work in this area has not provided an approach that is fully automatic and scales to large, realistic programs. Existing approaches are either based on interactive proofs [41, 42, 183], require manually-provided program annotations and complex side-conditions on program structure [44, 45], or are only capable of providing statistical guarantees of correctness of probabilistic programs [46, 47]. On the other hand, existing fully automated proof-search algorithms capable of exact guarantees [10, 50] do not scale to large programs. A more geometric perspective maybe beneficial in designing new verification algorithms for this task and is deserving of more investigation.

4.4 Generalization Guarantees for Learning Algorithms

Statistical learning theory is the standard framework for reasoning about the generalization guarantees of learning algorithms. [1] In this framework, the learning algorithm is described as searching over a set of hypotheses or functions or programs. The cardinality of this set

[1] Consider reading [184] for a concise introduction

can be finite or infinite. Assuming access to a finite number of samples, one would like to give upper bounds on the error in the models learnt by a learning algorithm. From the programming languages perspective, the learning algorithm is a program (usually, a probabilistic program) that consumes samples and produces a model from the set of hypotheses (or a program in a programming language). Further, we would like to prove that this probabilistic program representing the learning algorithm satisfies a specification expressing the required generalization guarantees. Generalization proofs crucially rely on the structure of the set being explored by the learning algorithm. By expressing this set as a programming language, can we simplify generalization proofs? Can we use this perspective to design new learning algorithms that only search over a restricted the set of programs (for instance, programs satisfying some logical specification) and prove stronger generalization guarantees for these algorithms? There have been some investigations into such questions [185, 186, 187] but much exploration remains to be done.

REFERENCES

[1] L. A. Levin, "Average Case Complete Problems," *SIAM Journal on Computing*, vol. 15, no. 1, pp. 285–286, Feb. 1986.

[2] H. G. Rice, "Classes of Recursively Enumerable Sets and Their Decision Problems," *Transactions of the American Mathematical Society*, vol. 74, no. 2, pp. 358–366, 1953.

[3] J. G. Siek and W. Taha, "Gradual typing for functional languages," in *In Scheme and Functional Programming Workshop*, 2006.

[4] S. Tobin-Hochstadt and M. Felleisen, "Interlanguage migration: From scripts to programs," in *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '06, Portland, Oregon, USA: Association for Computing Machinery, Oct. 2006, pp. 964–974, ISBN: 978-1-59593-491-8.

[5] C. Flanagan, "Hybrid type checking," in *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '06, Charleston, South Carolina, USA: Association for Computing Machinery, Jan. 2006, pp. 245–256, ISBN: 978-1-59593-027-9.

[6] K. Knowles and C. Flanagan, "Hybrid type checking," *ACM Transactions on Programming Languages and Systems*, vol. 32, no. 2, 6:1–6:34, Feb. 2010.

[7] A. Takikawa, D. Feltey, B. Greenman, M. S. New, J. Vitek, and M. Felleisen, "Is sound gradual typing dead?" In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '16, 2016.

[8] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '77, Los Angeles, California: Association for Computing Machinery, Jan. 1977, pp. 238–252, ISBN: 978-1-4503-7350-0.

[9] ——, "Abstract Interpretation Frameworks," *Journal of Logic and Computation*, vol. 2, no. 4, pp. 511–547, Aug. 1992.

[10] J. Geldenhuys, M. B. Dwyer, and W. Visser, "Probabilistic symbolic execution," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA 2012, Minneapolis, MN, USA: Association for Computing Machinery, Jul. 2012, pp. 166–176, ISBN: 978-1-4503-1454-1.

[11] M. B. Dwyer, A. Filieri, J. Geldenhuys, M. Gerrard, C. S. Păsăreanu, and W. Visser, "Probabilistic Program Analysis," in *Grand Timely Topics in Software Engineering*, J. Cunha, J. P. Fernandes, R. Lämmel, J. Saraiva, and V. Zaytsev, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2017, pp. 1–25, ISBN: 978-3-319-60074-1.

[12] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative Adversarial Nets," in *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds., Curran Associates, Inc., 2014, pp. 2672–2680.

[13] D. P. Kingma and M. Welling, "Auto-Encoding Variational Bayes," *ArXiv:1312.6114 [cs, stat]*, May 2014. arXiv: 1312.6114 [cs, stat].

[14] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," in *International Conference on Learning Representations*, 2014.

[15] D. Van Horn and M. Might, "Abstracting abstract machines," in *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '10, Baltimore, Maryland, USA: Association for Computing Machinery, Sep. 2010, pp. 51–62, ISBN: 978-1-60558-794-3.

[16] I. Sergey, D. Devriese, M. Might, J. Midtgaard, D. Darais, D. Clarke, and F. Piessens, "Monadic abstract interpreters," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 399–410, Jun. 2013.

[17] D. Darais, M. Might, and D. Van Horn, "Galois transformers and modular abstract interpreters: Reusable metatheory for program analysis," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2015, Pittsburgh, PA, USA: Association for Computing Machinery, Oct. 2015, pp. 552–571, ISBN: 978-1-4503-3689-5.

[18] D. Darais, N. Labich, P. C. Nguyen, and D. Van Horn, "Abstracting definitional interpreters (functional pearl)," *Proceedings of the ACM on Programming Languages*, vol. 1, no. ICFP, 12:1–12:25, Aug. 2017.

[19] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., Curran Associates, Inc., 2012, pp. 1097–1105.

[20] R. Jia and P. Liang, "Adversarial Examples for Evaluating Reading Comprehension Systems," in *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, Copenhagen, Denmark: Association for Computational Linguistics, Sep. 2017, pp. 2021–2031.

[21] M. Alzantot, Y. Sharma, A. Elgohary, B.-J. Ho, M. Srivastava, and K.-W. Chang, "Generating Natural Language Adversarial Examples," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, Brussels, Belgium: Association for Computational Linguistics, Oct. 2018, pp. 2890–2896.

[22] N. Carlini, P. Mishra, T. Vaidya, Y. Zhang, M. Sherr, C. Shields, D. Wagner, and W. Zhou, "Hidden Voice Commands," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 513–530, ISBN: 978-1-931971-32-4.

[23] N. Carlini and D. Wagner, "Audio Adversarial Examples: Targeted Attacks on Speech-to-Text," in *2018 IEEE Security and Privacy Workshops (SPW)*, May 2018, pp. 1–7.

[24] Y. Qin, N. Carlini, G. Cottrell, I. Goodfellow, and C. Raffel, "Imperceptible, Robust, and Targeted Adversarial Examples for Automatic Speech Recognition," in *International Conference on Machine Learning*, May 2019, ch. Machine Learning, pp. 5231–5240.

[25] X. Yuan, P. He, Q. Zhu, and X. Li, "Adversarial Examples: Attacks and Defenses for Deep Learning," *ArXiv:1712.07107 [cs, stat]*, Jul. 2018. arXiv: 1712.07107 [cs, stat].

[26] C. Liu, T. Arnon, C. Lazarus, C. Barrett, and M. J. Kochenderfer, "Algorithms for Verifying Deep Neural Networks," *ArXiv:1903.06758 [cs, stat]*, Mar. 2019. arXiv: 1903.06758 [cs, stat].

[27] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine Learning Testing: Survey, Landscapes and Horizons," *IEEE Transactions on Software Engineering*, pp. 1–1, 2020.

[28] T.-W. Weng*, H. Zhang*, P.-Y. Chen, J. Yi, D. Su, Y. Gao, C.-J. Hsieh, and L. Daniel, "Evaluating the Robustness of Neural Networks: An Extreme Value Theory Approach," in *International Conference on Learning Representations*, Feb. 2018.

[29] N. Benton, "Simple relational correctness proofs for static analyses and program transformations," in *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '04, Venice, Italy: Association for Computing Machinery, Jan. 2004, pp. 14–25, ISBN: 978-1-58113-729-3.

[30] G. Barthe, J. M. Crespo, and C. Kunz, "Relational Verification Using Product Programs," in *FM 2011: FORMAL METHODS*, M. Butler and W. Schulte, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2011, pp. 200–214, ISBN: 978-3-642-21437-0.

[31] M. R. Clarkson and F. B. Schneider, "Hyperproperties," in *2008 21st IEEE Computer Security Foundations Symposium*, Jun. 2008, pp. 51–65.

[32] L. Lamport, "Proving the Correctness of Multiprocess Programs," *IEEE Transactions on Software Engineering*, vol. 3, no. 2, pp. 125–143, Mar. 1977.

[33] G. Singh, T. Gehr, M. Püschel, and M. Vechev, "An abstract domain for certifying neural networks," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, 41:1–41:30, Jan. 2019.

[34] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. Vechev, "AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation," in *2018 IEEE Symposium on Security and Privacy (SP)*, May 2018, pp. 3–18.

[35] P. Cousot and N. Halbwachs, "Automatic discovery of linear restraints among variables of a program," in *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '78, Tucson, Arizona: Association for Computing Machinery, Jan. 1978, pp. 84–96, ISBN: 978-1-4503-7348-7.

[36] O. Bastani, Y. Ioannou, L. Lampropoulos, D. Vytiniotis, A. Nori, and A. Criminisi, "Measuring Neural Net Robustness with Constraints," in *Advances in Neural Information Processing Systems 29*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, Eds., Curran Associates, Inc., 2016, pp. 2613–2621.

[37] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks," in *Computer Aided Verification*, R. Majumdar and V. Kunčak, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2017, pp. 97–117, ISBN: 978-3-319-63387-9.

[38] G. Katz, D. A. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljić, D. L. Dill, M. J. Kochenderfer, and C. Barrett, "The Marabou Framework for Verification and Analysis of Deep Neural Networks," in *Computer Aided Verification*, I. Dillig and S. Tasiran, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2019, pp. 443–452, ISBN: 978-3-030-25540-4.

[39] M. Fischer, M. Balunovic, D. Drachsler-Cohen, T. Gehr, C. Zhang, and M. Vechev, "DL2: Training and Querying Neural Networks with Logic," in *International Conference on Machine Learning*, May 2019, ch. Machine Learning, pp. 1931–1941.

[40] R. Mangal, A. V. Nori, and A. Orso, "Robustness of neural networks: A probabilistic and practical approach," in *Proceedings of the 41st International Conference on Software Engineering: NEW IDEAS and Emerging Results*, ser. ICSE-NIER '19, Montreal, Quebec, Canada: IEEE Press, May 2019, pp. 93–96.

[41] G. Barthe, T. Espitau, M. Gaboardi, B. Grégoire, J. Hsu, and P.-Y. Strub, "An Assertion-Based Program Logic for Probabilistic Programs," in *European Symposium on Programming Languages and Systems*, A. Ahmed, Ed., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2018, pp. 117–144, ISBN: 978-3-319-89884-1.

[42] G. Barthe, T. Espitau, B. Grégoire, J. Hsu, and P.-Y. Strub, "Proving expected sensitivity of probabilistic programs," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, 57:1–57:29, Dec. 2017.

[43] J.-P. Katoen, A. K. McIver, L. A. Meinicke, and C. C. Morgan, "Linear-Invariant Generation for Probabilistic Programs:" in *Static Analysis*, R. Cousot and M. Martel, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2010, pp. 390–406, ISBN: 978-3-642-15769-1.

[44] A. Chakarov and S. Sankaranarayanan, "Probabilistic Program Analysis with Martingales," in *Computer Aided Verification*, N. Sharygina and H. Veith, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2013, pp. 511–526, ISBN: 978-3-642-39799-8.

[45] G. Barthe, T. Espitau, L. M. Ferrer Fioriti, and J. Hsu, "Synthesizing Probabilistic Invariants via Doob's Decomposition," in *Computer Aided Verification*, S. Chaudhuri and A. Farzan, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2016, pp. 43–61, ISBN: 978-3-319-41528-4.

[46] A. Sampson, P. Panchekha, T. Mytkowicz, K. S. McKinley, D. Grossman, and L. Ceze, "Expressing and verifying probabilistic assertions," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14, Edinburgh, United Kingdom: Association for Computing Machinery, Jun. 2014, pp. 112–122, ISBN: 978-1-4503-2784-8.

[47] O. Bastani, X. Zhang, and A. Solar-Lezama, "Probabilistic verification of fairness properties via concentration," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, 118:1–118:27, Oct. 2019.

[48] P. Cousot and M. Monerau, "Probabilistic Abstract Interpretation," in *European Symposium on Programming Languages and Systems*, H. Seidl, Ed., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2012, pp. 169–193, ISBN: 978-3-642-28869-2.

[49] D. Wang, J. Hoffmann, and T. Reps, "PMAF: An algebraic framework for static analysis of probabilistic programs," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018, Philadelphia, PA, USA: Association for Computing Machinery, Jun. 2018, pp. 513–528, ISBN: 978-1-4503-5698-5.

[50] S. Sankaranarayanan, A. Chakarov, and S. Gulwani, "Static analysis for probabilistic programs: Inferring whole program properties from finitely many paths," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13, Seattle, Washington, USA: Association for Computing Machinery, Jun. 2013, pp. 447–458, ISBN: 978-1-4503-2014-6.

[51] A. Albarghouthi, L. D'Antoni, S. Drews, and A. V. Nori, "FairSquare: Probabilistic verification of program fairness," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, 80:1–80:30, Oct. 2017.

[52] S. Chaudhuri, S. Gulwani, R. Lublinerman, and S. Navidpour, "Proving programs robust," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11, Szeged, Hungary: Association for Computing Machinery, Sep. 2011, pp. 102–112, ISBN: 978-1-4503-0443-6.

[53] P. L. Combettes and J.-C. Pesquet, "Lipschitz Certificates for Neural Network Structures Driven by Averaged Activation Operators," *ArXiv:1903.01014 [math]*, Jul. 2019. arXiv: 1903.01014 [math].

[54] M. Fazlyab, A. Robey, H. Hassani, M. Morari, and G. Pappas, "Efficient and Accurate Estimation of Lipschitz Constants for Deep Neural Networks," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d\textquotesingle Alché-Buc, E. Fox, and R. Garnett, Eds., Curran Associates, Inc., 2019, pp. 11 427–11 438.

[55] F. Latorre, P. Rolland, and V. Cevher, "Lipschitz constant estimation of Neural Networks via sparse polynomial optimization," *ArXiv:2004.08688 [cs, stat]*, Apr. 2020. arXiv: 2004.08688 [cs, stat].

[56] A. Virmaux and K. Scaman, "Lipschitz regularity of deep neural networks: Analysis and efficient estimation," in *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., Curran Associates, Inc., 2018, pp. 3835–3844.

[57] L. Weng, H. Zhang, H. Chen, Z. Song, C.-J. Hsieh, L. Daniel, D. Boning, and I. Dhillon, "Towards Fast Computation of Certified Robustness for ReLU Networks," in *International Conference on Machine Learning*, Jul. 2018, ch. Machine Learning, pp. 5276–5285.

[58] H. Zhang, P. Zhang, and C.-J. Hsieh, "RecurJac: An Efficient Recursive Algorithm for Bounding Jacobian Matrix of Neural Networks and Its Applications," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, pp. 5757–5764, Jul. 2019.

[59] Y. Tsuzuku, I. Sato, and M. Sugiyama, "Lipschitz-margin training: Scalable certification of perturbation invariance for deep neural networks," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, ser. NIPS'18, Montréal, Canada: Curran Associates Inc., Dec. 2018, pp. 6542–6551.

[60] H. Gouk, E. Frank, B. Pfahringer, and M. Cree, "Regularisation of Neural Networks by Enforcing Lipschitz Continuity," *ArXiv:1804.04368 [cs, stat]*, Sep. 2018. arXiv: 1804.04368 [cs, stat].

[61] J. Slepak, O. Shivers, and P. Manolios, "An Array-Oriented Language with Static Rank Polymorphism," in *European Symposium on Programming Languages and Systems*, Z. Shao, Ed., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2014, pp. 27–46, ISBN: 978-3-642-54833-8.

[62] J. Gibbons, "APLicative Programming with Naperian Functors," in *European Symposium on Programming Languages and Systems*, H. Yang, Ed., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2017, pp. 556–583, ISBN: 978-3-662-54434-1.

[63] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ser. ICML'15, Lille, France: JMLR.org, Jul. 2015, pp. 448–456.

[64] M. D. Zeiler, D. Krishnan, G. W. Taylor, and R. Fergus, "Deconvolutional networks," in *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, Jun. 2010, pp. 2528–2535.

[65] G. Barthe, J. M. Crespo, and C. Kunz, "Relational verification using product programs," in *FM 2011: FORMAL Methods*.

[66] M. R. Clarkson and F. B. Schneider, "Hyperproperties," *J. Comput. Secur.*, vol. 18, no. 6, 2010.

[67] G. Barthe, P. R. D'Argenio, and T. Rezk, "Secure information flow by self-composition," in *Proceedings of the 17th IEEE Workshop on Computer Security Foundations*, ser. CSFW '04, 2004.

[68] P. Cousot and N. Halbwachs, "Automatic discovery of linear restraints among variables of a program," in *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1978.

[69] S. Sankaranarayanan, F. Ivančić, I. Shlyakhter, and A. Gupta, "Static analysis in disjunctive numerical domains," in *Proceedings of the 13th International Conference on Static Analysis*, ser. SAS'06, 2006.

[70] B. Cousins and S. Vempala, "Gaussian Cooling and $O^{\hat{*}}(n\hat{3})$ Algorithms for Volume and Gaussian Volume," *SIAM Journal on Computing*, vol. 47, no. 3, pp. 1237–1273, Jan. 2018.

[71] C. P. Robert and G. Casella, *Monte Carlo Statistical Methods*. 2010.

[72] K. Ghorbal, E. Goubault, and S. Putot, "The Zonotope Abstract Domain Taylor1+," in *Computer Aided Verification*, A. Bouajjani and O. Maler, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2009, pp. 627–633, ISBN: 978-3-642-02658-4.

[73] I. Bárány and Z. Füredi, "Computing the volume is difficult," *Discrete & Computational Geometry*, vol. 2, no. 4, pp. 319–326, Dec. 1987.

[74] G. Elekes, "A geometric inequality and the complexity of computing volume," *Discrete & Computational Geometry*, vol. 1, no. 4, pp. 289–292, Dec. 1986.

[75] M. E. Dyer and A. M. Frieze, "On the Complexity of Computing the Volume of a Polyhedron," *SIAM Journal on Computing*, vol. 17, no. 5, pp. 967–974, Oct. 1988.

[76] M. Dyer, A. Frieze, and R. Kannan, "A random polynomial-time algorithm for approximating the volume of convex bodies," *Journal of the ACM*, vol. 38, no. 1, pp. 1–17, Jan. 1991.

[77] C. Singh, *Csinva/gan-vae-pretrained-pytorch*, May 2020.

[78] A. Chen, *Aaron-xichen/pytorch-playground*, May 2020.

[79] G. Barthe, P. D'Argenio, and T. Rezk, "Secure information flow by self-composition," in *Proceedings. 17th IEEE Computer Security Foundations Workshop, 2004.*, Jun. 2004, pp. 100–114.

[80] S. Webb, T. Rainforth, Y. W. Teh, and M. P. Kumar, "A Statistical Approach to Assessing Neural Network Robustness," in *International Conference on Learning Representations*, Sep. 2018.

[81] L. Weng, P.-Y. Chen, L. Nguyen, M. Squillante, A. Boopathy, I. Oseledets, and L. Daniel, "PROVEN: Verifying Robustness of Neural Networks with a Probabilistic Approach," in *International Conference on Machine Learning*, May 2019, ch. Machine Learning, pp. 6727–6736.

[82] T. Baluta, S. Shen, S. Shinde, K. S. Meel, and P. Saxena, "Quantitative Verification of Neural Networks and Its Security Applications," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19, London, United Kingdom: Association for Computing Machinery, Nov. 2019, pp. 1249–1264, ISBN: 978-1-4503-6747-9.

[83] J. C. Reynolds, "The Meaning of Types From Intrinsic to Extrinsic Semantics," *BRICS Report Series*, no. 32, Jun. 2000.

[84] S. Keidel, C. B. Poulsen, and S. Erdweg, "Compositional soundness proofs of abstract interpreters," *Proceedings of the ACM on Programming Languages*, vol. 2, no. ICFP, 72:1–72:26, Jul. 2018.

[85] S. Keidel and S. Erdweg, "Sound and reusable components for abstract interpretation," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, 176:1–176:28, Oct. 2019.

[86] P. Cousot, "The calculational design of a generic abstract interpreter," *Calculational system design*, pp. 421–505, 1999.

[87] T. Reps, M. Sagiv, and G. Yorsh, "Symbolic Implementation of the Best Transformer," in *Verification, Model Checking, and Abstract Interpretation*, B. Steffen and G. Levi, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2004, pp. 252–266, ISBN: 978-3-540-24622-0.

[88] A. Thakur, A. Lal, J. Lim, and T. Reps, "PostHat and All That: Automating Abstract Interpretation," *Electronic Notes in Theoretical Computer Science*, Fourth Workshop on Tools for Automatic Program Analysis (TAPAS 2013), vol. 311, pp. 15–32, Feb. 2015.

[89] D. Darais and D. V. Horn, "Constructive Galois Connections," *Journal of Functional Programming*, vol. 29, 2019/ed.

[90] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *Journal of the ACM*, vol. 50, no. 5, pp. 752–794, Sep. 2003.

[91] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang, "Compositional Shape Analysis by Means of Bi-Abduction," *Journal of the ACM*, vol. 58, no. 6, 26:1–26:66, Dec. 2011.

[92] I. Dillig, T. Dillig, and A. Aiken, "Automated error diagnosis using abductive inference," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12, Beijing, China: Association for Computing Machinery, Jun. 2012, pp. 181–192, ISBN: 978-1-4503-1205-9.

[93] N. Vazou, É. Tanter, and D. Van Horn, "Gradual liquid type inference," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, 132:1–132:25, Oct. 2018.

[94] J. Bader, J. Aldrich, and É. Tanter, "Gradual Program Verification," in *Verification, Model Checking, and Abstract Interpretation*, I. Dillig and J. Palsberg, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2018, pp. 25–46, ISBN: 978-3-319-73721-8.

[95] M. Felleisen and D. P. Friedman, "A calculus for assignments in higher-order languages," in *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '87, Munich, West Germany: Association for Computing Machinery, Oct. 1987, p. 314, ISBN: 978-0-89791-215-0.

[96] P. Wadler, "The essence of functional programming," in *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '92, Albuquerque, New Mexico, USA: Association for Computing Machinery, Feb. 1992, pp. 1–14, ISBN: 978-0-89791-453-6.

[97] S. Liang, P. Hudak, and M. Jones, "Monad transformers and modular interpreters," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '95, San Francisco, California, USA: Association for Computing Machinery, Jan. 1995, pp. 333–343, ISBN: 978-0-89791-692-9.

[98] P. Wadler and S. Blott, "How to make ad-hoc polymorphism less ad hoc," in *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '89, Austin, Texas, USA: Association for Computing Machinery, Jan. 1989, pp. 60–76, ISBN: 978-0-89791-294-5.

[99] P. Cousot and R. Cousot, "Systematic design of program analysis frameworks," in *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '79, San Antonio, Texas: Association for Computing Machinery, Jan. 1979, pp. 269–282, ISBN: 978-1-4503-7357-9.

[100] S. Padhi, R. Sharma, and T. Millstein, "Data-driven precondition inference with learned features," in *Proceedings of the 37th ACM SIGPLAN Conference on Pro-*

gramming Language Design and Implementation, ser. PLDI '16, Santa Barbara, CA, USA: Association for Computing Machinery, Jun. 2016, pp. 42–56, ISBN: 978-1-4503-4261-2.

[101] R. Gupta, M. L. Soffa, and J. Howard, "Hybrid slicing: Integrating dynamic information with static analysis," *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 4, pp. 370–397, Oct. 1997.

[102] M. Mock, D. C. Atkinson, C. Chambers, and S. J. Eggers, "Improving program slicing with dynamic points-to data," in *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, ser. SIGSOFT '02/FSE-10, Charleston, South Carolina, USA: Association for Computing Machinery, Nov. 2002, pp. 71–80, ISBN: 978-1-58113-514-5.

[103] B. Dufour, B. G. Ryder, and G. Sevitsky, "Blended analysis for performance understanding of framework-based applications," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ser. ISSTA '07, London, United Kingdom: Association for Computing Machinery, Jul. 2007, pp. 118–128, ISBN: 978-1-59593-734-6.

[104] C. Csallner, Y. Smaragdakis, and T. Xie, "DSD-Crasher: A hybrid analysis tool for bug finding," *ACM Transactions on Software Engineering and Methodology*, vol. 17, no. 2, 8:1–8:37, May 2008.

[105] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini, "Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11, Waikiki, Honolulu, HI, USA: Association for Computing Machinery, May 2011, pp. 241–250, ISBN: 978-1-4503-0445-0.

[106] J. Kinder and D. Kravchenko, "Alternating Control Flow Reconstruction," in *Verification, Model Checking, and Abstract Interpretation*, V. Kuncak and A. Rybalchenko, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2012, pp. 267–282, ISBN: 978-3-642-27940-9.

[107] S. Wei and B. G. Ryder, "Practical blended taint analysis for JavaScript," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ser. ISSTA 2013, Lugano, Switzerland: Association for Computing Machinery, Jul. 2013, pp. 336–346, ISBN: 978-1-4503-2159-4.

[108] N. Grech, G. Fourtounis, A. Francalanza, and Y. Smaragdakis, "Heaps don't lie: Countering unsoundness with heap snapshots," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, 68:1–68:27, Oct. 2017.

[109] D. Devecsery, P. M. Chen, J. Flinn, and S. Narayanasamy, "Optimistic Hybrid Analysis: Accelerating Dynamic Analysis through Predicated Static Analysis," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18, Williamsburg, VA, USA: Association for Computing Machinery, Mar. 2018, pp. 348–362, ISBN: 978-1-4503-4911-6.

[110] A. Gupta, R. Majumdar, and A. Rybalchenko, "From Tests to Proofs," in *Tools and Algorithms for the Construction and Analysis of Systems*, S. Kowalewski and A. Philippou, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2009, pp. 262–276, ISBN: 978-3-642-00768-2.

[111] N. E. Beckman, A. V. Nori, S. K. Rajamani, R. J. Simmons, S. D. Tetali, and A. V. Thakur, "Proofs from Tests," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 495–508, Jul. 2010.

[112] M. Naik, H. Yang, G. Castelnuovo, and M. Sagiv, "Abstractions from tests," in *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '12, Philadelphia, PA, USA: Association for Computing Machinery, Jan. 2012, pp. 373–386, ISBN: 978-1-4503-1083-3.

[113] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1-3, pp. 35–45, Dec. 2007.

[114] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, and A. V. Nori, "Verification as Learning Geometric Concepts," in *Static Analysis*, F. Logozzo and M. Fähndrich, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2013, pp. 388–411, ISBN: 978-3-642-38856-9.

[115] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. V. Nori, "A Data Driven Approach for Algebraic Loop Invariants," in *Programming Languages and Systems*, M. Felleisen and P. Gardner, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2013, pp. 574–592, ISBN: 978-3-642-37036-6.

[116] P. Garg, C. Löding, P. Madhusudan, and D. Neider, "ICE: A Robust Framework for Learning Invariants," in *Computer Aided Verification*, A. Biere and R. Bloem, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2014, pp. 69–87, ISBN: 978-3-319-08867-9.

[117] H. Zhu, A. V. Nori, and S. Jagannathan, "Learning refinement types," in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP 2015, Vancouver, BC, Canada: Association for Computing Machinery, Aug. 2015, pp. 400–411, ISBN: 978-1-4503-3669-7.

[118] R. Sharma and A. Aiken, "From invariant checking to invariant inference using randomized search," *Formal Methods in System Design*, vol. 48, no. 3, pp. 235–256, Jun. 2016.

[119] P. Garg, D. Neider, P. Madhusudan, and D. Roth, "Learning invariants using decision trees and implication counterexamples," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '16, St. Petersburg, FL, USA: Association for Computing Machinery, Jan. 2016, pp. 499–512, ISBN: 978-1-4503-3549-2.

[120] H. Zhu, G. Petri, and S. Jagannathan, "Automatically learning shape specifications," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16, Santa Barbara, CA, USA: Association for Computing Machinery, Jun. 2016, pp. 491–507, ISBN: 978-1-4503-4261-2.

[121] G. Fedyukovich, S. J. Kaufman, and R. Bodík, "Sampling invariants from frequency distributions," in *2017 Formal Methods in Computer Aided Design (FMCAD)*, Oct. 2017, pp. 100–107.

[122] X. Si, H. Dai, M. Raghothaman, M. Naik, and L. Song, "Learning Loop Invariants for Program Verification," in *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., Curran Associates, Inc., 2018, pp. 7751–7762.

[123] H. Zhu, S. Magill, and S. Jagannathan, "A data-driven CHC solver," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018, Philadelphia, PA, USA: Association for Computing Machinery, Jun. 2018, pp. 707–721, ISBN: 978-1-4503-5698-5.

[124] T. C. Le, G. Zheng, and T. Nguyen, "SLING: Using dynamic analysis to infer program invariants in separation logic," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019, Phoenix, AZ, USA: Association for Computing Machinery, Jun. 2019, pp. 788–801, ISBN: 978-1-4503-6712-7.

[125] A. Miltner, S. Padhi, T. Millstein, and D. Walker, "Data-driven inference of representation invariants," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020, London, UK: Association for Computing Machinery, Jun. 2020, pp. 1–15, ISBN: 978-1-4503-7613-6.

[126] G. Ammons, R. Bodík, and J. R. Larus, "Mining specifications," in *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Lan-*

guages, ser. POPL '02, Portland, Oregon: Association for Computing Machinery, Jan. 2002, pp. 4–16, ISBN: 978-1-58113-450-6.

[127] S. Sankaranarayanan, S. Chaudhuri, F. Ivančić, and A. Gupta, "Dynamic inference of likely data preconditions over predicates by tree learning," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ser. ISSTA '08, Seattle, WA, USA: Association for Computing Machinery, Jul. 2008, pp. 295–306, ISBN: 978-1-60558-050-0.

[128] O. Bastani, R. Sharma, A. Aiken, and P. Liang, "Active learning of points-to specifications," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018, Philadelphia, PA, USA: Association for Computing Machinery, Jun. 2018, pp. 678–692, ISBN: 978-1-4503-5698-5.

[129] M. Furr, J.-h. D. An, and J. S. Foster, "Profile-guided static typing for dynamic scripting languages," in *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '09, Orlando, Florida, USA: Association for Computing Machinery, Oct. 2009, pp. 283–300, ISBN: 978-1-60558-766-0.

[130] J.-h. D. An, A. Chaudhuri, J. S. Foster, and M. Hicks, "Dynamic inference of static types for ruby," *ACM SIGPLAN Notices*, vol. 46, no. 1, pp. 459–472, Jan. 2011.

[131] R. Grigore and H. Yang, "Abstraction refinement guided by a learnt probabilistic model," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '16, St. Petersburg, FL, USA: Association for Computing Machinery, Jan. 2016, pp. 485–498, ISBN: 978-1-4503-3549-2.

[132] H. Oh, H. Yang, and K. Yi, "Learning a strategy for adapting a program analysis via bayesian optimisation," *ACM SIGPLAN Notices*, vol. 50, no. 10, pp. 572–588, Oct. 2015.

[133] K. Heo, H. Oh, and H. Yang, "Learning a Variable-Clustering Strategy for Octagon from Labeled Data Generated by a Static Analysis," in *Static Analysis*, X. Rival, Ed., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2016, pp. 237–256, ISBN: 978-3-662-53413-7.

[134] S. Cha, S. Jeong, and H. Oh, "Learning a Strategy for Choosing Widening Thresholds from a Large Codebase," in *Programming Languages and Systems*, A. Igarashi, Ed., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2016, pp. 25–41, ISBN: 978-3-319-47958-3.

[135] K. Chae, H. Oh, K. Heo, and H. Yang, "Automatically generating features for learning program analysis heuristics for C-like languages," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, 101:1–101:25, Oct. 2017.

[136] S. Jeong, M. Jeon, S. Cha, and H. Oh, "Data-driven context-sensitivity for points-to analysis," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, 100:1–100:28, Oct. 2017.

[137] K. Heo, H. Oh, and K. Yi, "Machine-Learning-Guided Selectively Unsound Static Analysis," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, May 2017, pp. 519–529.

[138] G. Singh, M. Püschel, and M. Vechev, "Fast Numerical Program Analysis with Reinforcement Learning," in *Computer Aided Verification*, H. Chockler and G. Weissenbacher, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2018, pp. 211–229, ISBN: 978-3-319-96145-3.

[139] J. He, G. Singh, M. Püschel, and M. Vechev, "Learning fast and precise numerical analysis," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020, London, UK: Association for Computing Machinery, Jun. 2020, pp. 1112–1127, ISBN: 978-1-4503-7613-6.

[140] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler, "From uncertainty to belief: Inferring the specification within," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, ser. OSDI '06, Seattle, Washington: USENIX Association, Nov. 2006, pp. 161–176, ISBN: 978-1-931971-47-8.

[141] B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee, "Merlin: Specification inference for explicit information flow problems," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09, Dublin, Ireland: Association for Computing Machinery, Jun. 2009, pp. 75–86, ISBN: 978-1-60558-392-1.

[142] N. E. Beckman and A. V. Nori, "Probabilistic, modular and scalable inference of typestate specifications," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11, San Jose, California, USA: Association for Computing Machinery, Jun. 2011, pp. 211–221, ISBN: 978-1-4503-0663-8.

[143] T. Kremenek, K. Ashcraft, J. Yang, and D. Engler, "Correlation exploitation in error ranking," *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 6, pp. 83–93, Oct. 2004.

[144] T. Kremenek and D. Engler, "Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations," in *Static Analysis*, R. Cousot, Ed., ser.

Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2003, pp. 295–315, ISBN: 978-3-540-44898-3.

[145] M. Raghothaman, S. Kulkarni, K. Heo, and M. Naik, "User-guided program reasoning using Bayesian inference," *ACM SIGPLAN Notices*, vol. 53, no. 4, pp. 722–735, Jun. 2018.

[146] R. Mangal, X. Zhang, A. V. Nori, and M. Naik, "A user-guided approach to program analysis," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015, Bergamo, Italy: Association for Computing Machinery, Aug. 2015, pp. 462–473, ISBN: 978-1-4503-3675-8.

[147] J. Chen, J. Wei, Y. Feng, O. Bastani, and I. Dillig, "Relational verification using reinforcement learning," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, 141:1–141:30, Oct. 2019.

[148] A. A. Alemi, F. Chollet, N. Een, G. Irving, C. Szegedy, and J. Urban, "DeepMath - deep sequence models for premise selection," in *Proceedings of the 30th International Conference on Neural Information Processing Systems*, ser. NIPS'16, Barcelona, Spain: Curran Associates Inc., Dec. 2016, pp. 2243–2251, ISBN: 978-1-5108-3881-9.

[149] S. Loos, G. Irving, C. Szegedy, and C. Kaliszyk, "Deep Network Guided Proof Search," in *LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, 2017, pp. 85–105.

[150] C. Kaliszyk, J. Urban, H. Michalewski, and M. Olšák, "Reinforcement Learning of Theorem Proving," in *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., Curran Associates, Inc., 2018, pp. 8822–8833.

[151] K. Bansal, S. Loos, M. Rabe, C. Szegedy, and S. Wilcox, "HOList: An Environment for Machine Learning of Higher Order Logic Theorem Proving," in *International Conference on Machine Learning*, May 2019, ch. Machine Learning, pp. 454–463.

[152] A. Sanchez-Stern, Y. Alhessi, L. Saul, and S. Lerner, "Generating correctness proofs with neural networks," in *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, ser. MAPL 2020, London, UK: Association for Computing Machinery, Jun. 2020, pp. 1–10, ISBN: 978-1-4503-7996-0.

[153] L. Blaauwbroek, J. Urban, and H. Geuvers, "Tactic Learning and Proving for the Coq Proof Assistant," in *EPIC SERIES in Computing*, vol. 73, EasyChair, May 2020, pp. 138–150.

[154] P. Bielik, V. Raychev, and M. Vechev, "Learning a Static Analyzer from Data," in *Computer Aided Verification*, R. Majumdar and V. Kunčak, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2017, pp. 233–253, ISBN: 978-3-319-63387-9.

[155] V. J. Hellendoorn, C. Bird, E. T. Barr, and M. Allamanis, "Deep learning type inference," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018, Lake Buena Vista, FL, USA: Association for Computing Machinery, Oct. 2018, pp. 152–162, ISBN: 978-1-4503-5573-5.

[156] J. Wei, M. Goyal, G. Durrett, and I. Dillig, "LambdaNet: Probabilistic Type Inference using Graph Neural Networks," in *International Conference on Learning Representations*, Sep. 2019.

[157] M. Allamanis, E. T. Barr, S. Ducousso, and Z. Gao, "Typilus: Neural type hints," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020, London, UK: Association for Computing Machinery, Jun. 2020, pp. 91–105, ISBN: 978-1-4503-7613-6.

[158] V. Raychev, M. Vechev, and A. Krause, "Predicting Program Properties from "Big Code"," *ACM SIGPLAN Notices*, vol. 50, no. 1, pp. 111–124, Jan. 2015.

[159] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "A general path-based representation for predicting program properties," *ACM SIGPLAN Notices*, vol. 53, no. 4, pp. 404–419, Jun. 2018.

[160] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A Survey of Machine Learning for Big Code and Naturalness," *ACM Computing Surveys*, vol. 51, no. 4, 81:1–81:37, Jul. 2018.

[161] S. Thatte, "Quasi-static typing," in *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '90, San Francisco, California, USA: Association for Computing Machinery, Dec. 1989, pp. 367–381, ISBN: 978-0-89791-343-0.

[162] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin, "Dynamic typing in a statically typed language," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 2, pp. 237–268, Apr. 1991.

[163] R. Cartwright and M. Fagan, "Soft typing," in *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, ser. PLDI '91, Toronto, Ontario, Canada: Association for Computing Machinery, May 1991, pp. 278–292, ISBN: 978-0-89791-428-4.

[164] F. Henglein, "Dynamic typing: Syntax and proof theory," *Science of Computer Programming*, vol. 22, no. 3, pp. 197–230, Jun. 1994.

[165] O. Bastani, S. Anand, and A. Aiken, "Interactively verifying absence of explicit information flows in Android apps," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2015, Pittsburgh, PA, USA: Association for Computing Machinery, Oct. 2015, pp. 299–315, ISBN: 978-1-4503-3689-5.

[166] N. Stulova, J. F. Morales, and M. V. Hermenegildo, "Reducing the overhead of assertion run-time checks via static analysis," in *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming*, ser. PPDP '16, Edinburgh, United Kingdom: Association for Computing Machinery, Sep. 2016, pp. 90–103, ISBN: 978-1-4503-4148-6.

[167] O. Bastani, R. Sharma, L. Clapp, S. Anand, and A. Aiken, "Eventually Sound Points-To Analysis with Specifications," in *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*, A. F. Donaldson, Ed., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 134, Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 11:1–11:28, ISBN: 978-3-95977-111-5.

[168] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan, "Efficient and precise datarace detection for multithreaded object-oriented programs," *ACM SIGPLAN Notices*, vol. 37, no. 5, pp. 258–269, May 2002.

[169] T. Elmas, S. Qadeer, and S. Tasiran, "Goldilocks: A race and transaction-aware java runtime," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 245–255, Jun. 2007.

[170] A. C. Myers, "JFlow: Practical mostly-static information flow control," in *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '99, San Antonio, Texas, USA: Association for Computing Machinery, Jan. 1999, pp. 228–241, ISBN: 978-1-58113-095-9.

[171] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "CETS: Compiler enforced temporal safety for C," in *Proceedings of the 2010 International Symposium on Memory Management*, ser. ISMM '10, Toronto, Ontario, Canada: Association for Computing Machinery, Jun. 2010, pp. 31–40, ISBN: 978-1-4503-0054-4.

[172] G. C. Necula, S. McPeak, and W. Weimer, "CCured: Type-safe retrofitting of legacy code," *ACM SIGPLAN Notices*, vol. 37, no. 1, pp. 128–139, Jan. 2002.

[173] D. Rhodes, C. Flanagan, and S. N. Freund, "BigFoot: Static check placement for dynamic race detection," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017, Barcelona,

Spain: Association for Computing Machinery, Jun. 2017, pp. 141–156, ISBN: 978-1-4503-4988-8.

[174] S. Tobin-Hochstadt and D. Van Horn, "Higher-order symbolic execution via contracts," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '12, Tucson, Arizona, USA: Association for Computing Machinery, Oct. 2012, pp. 537–554, ISBN: 978-1-4503-1561-6.

[175] P. C. Nguyen, S. Tobin-Hochstadt, and D. Van Horn, "Soft contract verification," *ACM SIGPLAN Notices*, vol. 49, no. 9, pp. 139–152, Aug. 2014.

[176] P. C. Nguyen, T. Gilray, S. Tobin-Hochstadt, and D. Van Horn, "Soft contract verification for higher-order stateful programs," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, 51:1–51:30, Dec. 2017.

[177] P. C. Nguyen, S. Tobin-Hochstadt, and D. V. Horn, "Higher order symbolic execution for contract verification and refutation*," *Journal of Functional Programming*, vol. 27, 2017/ed.

[178] B. Montagu and T. Jensen, "Stable relations and abstract interpretation of higher-order programs," *Proceedings of the ACM on Programming Languages*, vol. 4, no. ICFP, 119:1–119:30, Aug. 2020.

[179] P. Cousot and R. Cousot, "Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages)," in *Proceedings of 1994 IEEE International Conference on Computer Languages (ICCL'94)*, May 1994, pp. 95–112.

[180] R. B. Findler and M. Felleisen, "Contracts for higher-order functions," in *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '02, 2002.

[181] P. Cousot, "Types as abstract interpretations," in *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '97, New York, NY, USA: Association for Computing Machinery, Jan. 1997, pp. 316–331, ISBN: 978-0-89791-853-4.

[182] Z. Pavlinovic, Y. Su, and T. Wies, "Data Flow Refinement Type Inference," *ArXiv:2011.04876 [cs]*, Nov. 2020. arXiv: 2011.04876 [cs].

[183] A. Aguirre, G. Barthe, J. Hsu, B. L. Kaminski, J.-P. Katoen, and C. Matheja, "A Pre-Expectation Calculus for Probabilistic Sensitivity," *ArXiv:1901.06540 [cs]*, Aug. 2020. arXiv: 1901.06540 [cs].

[184] O. Bousquet, S. Boucheron, and G. Lugosi, "Introduction to Statistical Learning Theory," in *Advanced Lectures on Machine Learning: ML SUMMER SCHOOLS 2003, Canberra, Australia, February 2 - 14, 2003, Tübingen, Germany, August 4 - 16, 2003, Revised Lectures*, ser. Lecture Notes in Computer Science, O. Bousquet, U. von Luxburg, and G. Rätsch, Eds., Berlin, Heidelberg: Springer, 2004, pp. 169–207, ISBN: 978-3-540-28650-9.

[185] M. Grohe, C. Löding, and M. Ritzert, "Learning MSO-definable hypotheses on strings," in *International Conference on Algorithmic Learning Theory*, PMLR, Oct. 2017, pp. 434–451.

[186] M. Grohe and M. Ritzert, "Learning first-order definable concepts over structures of small degree," in *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, Jun. 2017, pp. 1–12.

[187] V. Belle and B. Juba, "Implicitly learning to reason in first-order logic," *Advances in Neural Information Processing Systems*, vol. 32, pp. 3381–3391, 2019.