DISS. ETH NO. 28120

SYMBOLIC METHODS FOR MACHINE INTELLIGENCE

A dissertation submitted to attain the degree of

DOCTOR OF SCIENCES OF ETH ZURICH
(Dr. sc. ETH Zurich)

presented by

TIMON GEHR

Bachelor in Computer Science, ETH Zurich
Master in Computer Science, ETH Zurich
born on 03.10.1991
citizen of Ebnat-Kappel

accepted on the recommendation of

Prof. Martin Vechev (advisor)
Prof. Sasa Misailovic
Prof. Swarat Chaudhuri
Prof. Armando Solar-Lezama

2022

ABSTRACT

Intelligent automated systems are increasingly deployed in critical areas. Therefore, it is important that such systems operate reliably, consistently, and accountably. While systems based on learning can achieve impressive results on natural data, they are usually imperfect and often easy to fool into making wrong decisions by a malicious adversary. Formal methods can be used to create systems with strong correctness guarantees based on symbolic reasoning. However, the formalization of many tasks that seem to be the natural realm of learning-based methods, such as natural image classification, has proven rather elusive. Therefore, to achieve safe yet useful outcomes, it seems likely that ultimately, we should deploy a combination of statistical learning and symbolic methods.

In this thesis, we present multiple systems that combine statistical methods with symbolic reasoning in two major domains of research: Neural networks and probabilistic programming.

We present AI^2, the first sound and scalable analyzer for deep neural networks. Based on overapproximation, AI^2 can automatically prove safety properties (e.g., robustness) of realistic neural networks (e.g., convolutional neural networks). The key insight behind AI^2 is to phrase reasoning about safety and robustness of neural networks in terms of classic abstract interpretation, enabling us to leverage decades of advances in that area. We present a complete implementation of AI^2 together with an extensive evaluation. Our results show that AI^{2}is significantly faster than existing analyzers based on symbolic analysis, which often take hours to verify simple fully connected networks. AI^2 can handle deep convolutional networks, which are beyond the reach of prior methods. In particular, we show that the Zonotope abstract domain is suitable for neural network analysis. Building on AI^2 and the Zonotope domain, we present DeepZ, a tool that additionally handles Tanh and Sigmoid activation functions, while being significantly more scalable and precise. These benefits are due to carefully designed abstract transformers tailored to neural networks as well as the Zonotope domain. In contrast to many existing analyzers, AI^2 and DeepZ are sound with respect to floating-point arithmetic.

We further present PSI, a novel symbolic analysis system for exact inference in probabilistic programs with both continuous and discrete random variables. PSI computes succinct symbolic representations of the joint posterior distribution represented by a given probabilistic program. PSI can compute answers to various posterior distribution, expectation, and assertion queries using its own backend for symbolic reasoning. Our evaluation shows that PSI is more effective than existing exact inference approaches: (i) it successfully computes a precise result for more programs, and (ii) simplifies expressions that existing computer algebra systems

(e.g., Mathematica and Maple) fail to handle. Building on PSI, we present λPSI, the first probabilistic programming language and system that supports higher-order exact inference for probabilistic programs with first-class functions, nested inference and discrete, continuous and mixed random variables. λPSI's solver is based on symbolic reasoning and computes the exact distribution represented by a program. We show that λPSI is practically effective – it automatically computes exact distributions for a number of interesting applications, from rational agents to information theory, many of which could only be handled approximately before.

ZUSAMMENFASSUNG

Intelligente automatisierte Systeme werden zunehmend in sensiblen Bereichen eingesetzt. Daher ist es wichtig, dass solche Systeme zuverlässig, konsistent und überprüfbar arbeiten.

Lernende Systeme können zwar auf natürlichen Daten beeindruckende Ergebnisse erzielen, doch sie sind in der Regel unvollkommen und können von böswilligen Widersachern oft leicht zu falschen Entscheidungen verleitet werden. Mit Hilfe formaler Methoden lassen sich Systeme mit umfassenden Korrektheitsgarantien auf der Grundlage symbolischer Verfahren entwickeln. Allerdings hat sich die Formalisierung vieler Aufgaben, die die natürliche Domäne auf Lernen basierter Methoden zu sein scheinen, wie z.B. die Klassifizierung natürlicher Bilder, als recht unzugänglich erwiesen. Es scheint daher wahrscheinlich dass wir, um sichere und dennoch nützliche Ergebnisse zu erzielen, letztendlich eine Kombination aus statistischem Lernen und symbolischen Methoden einsetzen sollten.

In dieser Arbeit stellen wir mehrere Systeme vor, die statistische Methoden mit symbolischen Verfahren kombinieren, in zwei wichtigen Forschungsbereichen: Neuronale Netze und probabilistische Programmierung.

Wir stellen AI^2 vor, den ersten korrekten und skalierenden Analysator für mehrschichtige neuronale Netze. Basierend auf Überapproximation kann AI^2 automatisch Sicherheitseigenschaften (z.B. Robustheit) von realistischen neuronalen Netzen (z.B. faltende neuronale Netze) verifizieren. Die wichtigste Erkenntnis hinter AI^2 ist die Formulierung der Argumentation über Sicherheit und Robustheit neuronaler Netze in Form klassischer abstrakter Interpretation, was uns ermöglicht, jahrzehntelange Fortschritte in diesem Bereich zu nutzen. Wir präsentieren eine vollständige Implementierung von AI^2 zusammen mit einer umfassenden Evaluierung. Unsere Ergebnisse zeigen, dass AI^2 deutlich schneller ist als bestehende Analysatoren, die auf symbolischen Verfahren basieren. Diese brauchen oft Stunden, um einfache, vollständig verbundene Netze zu verifizieren. AI^2 ist in der Lage, tiefe faltige Netze zu verarbeiten, was frühere Methoden nicht geschafft haben. Wir zeigen insbesondere, dass die abstrakte Zonotop-Domäne dazu geeignet ist, neuronale Netze zu analysieren.

Aufbauend auf AI^2 und der Zonotop-Domäne stellen wir DeepZ vor, ein Tool, das zusätzlich Tanh- and Sigmoid-Aktivierungsfunktionen unterstützt und dabei deutlich skalierbarer und präziser ist. Diese Vorteile sind auf sorgfältig entworfene abstrakte Transformatoren zurückzuführen, die sowohl auf neuronale Netze als auch auf die Zonotop-Domäne zugeschnitten sind. Im Gegensatz zu vielen existierenden Analysatoren behandeln AI^2 und DeepZ Gleitkomma-Arithmetik korrekt.

Ausserdem stellen wir PSI vor, ein neuartiges symbolisches Analysesystem für exakte Inferenz in probabilistischen Programmen mit kontinuierlichen und diskreten Zufallsvariablen. PSI berechnet prägnante symbolische Darstellungen der gemeinsamen posterioren Verteilung, die durch ein gegebenes probabilistisches Programm repräsentiert wird. PSI kann Antworten auf verschiedene Posteriorverteilungs-, Erwartungswerts- und Behauptungsabfragen berechnen, unter Verwendung seines eigenen Backends für symbolische Schlussfolgerungen. Unsere Evaluierung zeigt, dass PSI effektiver ist als bestehende exakte Inferenzansätze: (i) es berechnet erfolgreich ein präzises Ergebnis für mehr Programme, und (ii) vereinfacht Ausdrücke, die bestehende Computeralgebra-Systeme (z.B.., Mathematica und Maple) nicht verarbeiten können. Aufbauend auf PSI, stellen wir λPSI vor, die erste probabilistische Programmiersprache und System welche exakte Inferenz höherer Ordnung für probabilistische Programme mit Funktionen erster Klasse, verschachtelter Inferenz und diskreten, kontinuierlichen und gemischten Zufallsvariablen unterstützen. Der Problemlöser von λPSI basiert auf symbolischer Argumentation und berechnet die exakte Verteilung, die durch ein Programm dargestellt wird. Wir zeigen, dass λPSI praktisch effektiv ist – es berechnet automatisch exakte Verteilungen für eine Reihe interessanter Anwendungen, von rationalen Agenten bis zur Informationstheorie, von denen viele bisher nur näherungsweise behandelt werden konnten.

PUBLICATIONS

This thesis is based on research that was also presented in the following publications:

ABSTRACT INTERPRETATION OF NEURAL NETWORKS

- **Timon Gehr**, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, Martin Vechev
  AI^2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation
  IEEE Symposium on Security and Privacy (SP), 2018. [51]

- Gagandeep Singh, **Timon Gehr**, Matthew Mirman, Markus Püschel, Martin Vechev
  Fast and Effective Robustness Certification
  Advances in Neural Information Processing Systems (NeurIPS), 2018. [153]

SYMBOLIC REASONING FOR PROBABILISTIC PROGRAMS

- **Timon Gehr**, Sasa Misailovic, Martin Vechev
  PSI: Exact Symbolic Inference for Probabilistic Programs
  International Conference on Computer Aided Verification (CAV), 2016. [50]

- **Timon Gehr**, Samuel Steffen, Martin Vechev
  λPSI: Exact Inference for Higher-Order Probabilistic Programs
  ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2020. [53]

The following publications were part of my Ph.D. research and contain results that build upon the results of this thesis or are supplemental to this work:

ABSTRACT INTERPRETATION OF NEURAL NETWORKS

- Matthew Mirman, **Timon Gehr**, Martin Vechev
  Differentiable Abstract Interpretation for Provably Robust Neural Networks
  International Conference on Machine Learning (ICML), 2018. [114]

- Gagandeep Singh, **Timon Gehr**, Markus Püschel, Martin Vechev
  An Abstract Domain for Certifying Neural Networks
  ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 2019. [155]

- Gagandeep Singh, **Timon Gehr**, Markus Püschel, Martin Vechev
  Boosting Robustness Certification of Neural Networks
  International Conference on Learning Representations (ICLR), 2019. [156]

- Mislav Balunovic, Maximilian Baader, Gagandeep Singh, **Timon Gehr**, Martin Vechev
  Certifying Geometric Robustness of Neural Networks
  Advances in Neural Information Processing Systems (NeurIPS), 2019. [11]

- Matthew Mirman, Alexander Hägele, **Timon Gehr**, Pavol Bielik, Martin Vechev
  Robustness Certification with Generative Models
  ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2021. [115]

SYMBOLIC REASONING FOR PROBABILISTIC PROGRAMS

- Martin Kucera, Petar Tsankov, **Timon Gehr**, Marco Guarnieri, Martin Vechev
  Synthesis of Probabilistic Privacy Enforcement
  ACM SIGSAC Conference on Computer and Communications Security (CCS), 2017. [97]

- **Timon Gehr**, Sasa Misailovic, Petar Tsankov, Laurent Vanbever, Pascal Wiesmann, Martin Vechev
  Bayonet: Probabilistic Inference for Networks
  ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2018. [52]

- Benjamin Bichsel, **Timon Gehr**, Dana Drachsler-Cohen, Petar Tsankov, Martin Vechev
  DP-Finder: Finding Differential Privacy Violations by Sampling and Optimization
  ACM SIGSAC Conference on Computer and Communications Security (CCS), 2018. [17]

- Benjamin Bichsel, Maximilian Baader, **Timon Gehr**, Martin Vechev
  Silq: A High-Level Quantum Language with Safe Uncomputation and Intuitive Semantics
  ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2020. [18]

ACKNOWLEDGEMENTS

I would like to use this opportunity to thank some of the people that supported me throughout my doctoral studies, both directly and indirectly.

First of all, I am grateful to my advisor Prof. Martin Vechev. Your hands-on, solution-oriented and collaborative approach within a positive, uncomplicated, and extraordinarily productive atmosphere is what every advisor should strive for.

I also owe thanks to Prof. Sasa Misailovic, Prof. Swarat Chaudhuri and Prof. Armando Solar-Lezama, for agreeing to serve on my doctoral examination committee.

I would like to acknowledge all co-authors of papers we have published during my Ph.D. studies: Maximilian Baader, Mislav Balunovic, Benjamin Bichsel, Pavol Bielik, Prof. Swarat Chaudhuri, Marco Cusumano-Towner, Dimitar Dimitrov, Pavle Djordjevich, Prof. Dana Drachsler-Cohen, Marc Fischer, Marco Guarnieri, Alexander Hägele, Dimitar I. Dimitrov, Prof. Vikash K. Mansinghka, Martin Kucera, Matthew Mirman, Prof. Sasa Misailovic, Prof. Markus Püschel, Prof. Gagandeep Singh, Samuel Steffen, Petar Tsankov, Prof. Laurent Vanbever, Prof. Martin Vechev, Pascal Wiesmann, and Prof. Ce Zhang.

I acknowledge Maria Christakis, Valentin Wüstholz and Prof. Peter Müller for showing me what to look for in a research collaboration.

I acknowledge Siegfried Clauss and Dima Nikolenkov for introducing me to the Swiss Science Olympiads. I am also very grateful to the volunteers and participants of the Swiss Olympiad in Informatics: To Josef Ziegler, Samuel Grütter, Ruben Andrist, Sandro Feuz, Johannes Josi, Sebastian Millius, Silvan Brüllmann, Yannick Stucki, Monika Steinová, Richard Královič and Prof. Juraj Hromkovič for introducing me to computer science, encouraging me to develop my skills and fostering my excitement for the field. To Daniel Wolleb, Johannes Kapfhammer, Stefanie Zbinden, Daniel Rutschmann, Benjamin Schmid, and Timon Stampfli, for their extraordinary organisatorial engagement for the olympiad while I was increasingly busy with my doctoral studies. For many inspiring discussions and memorable events at the Swiss Olympiad in Informatics, I would like to thank Christopher Burckhardt, Jan Schär, Yunshu Ouyang, André Ryser, Bibin Muttappillil, Jonas Meier, Adrian Roos, Alberts Reisons, Cédric Neukom, David Jenny, Elena Morbach, Elias Boschung, Erwan Serandour, Fabian Haller, Fabian Lyck, Florian Gatignon, Florian Wernli, Giovanni Serafini, Hanna Müller, Ian Boschung, Ivana Klasovita, Joël Huber, Joël Mathys, Kevin De Keyser, Kieran Nirkko, Lazar Todorovic, Lorenz Widmer, Luc Haller, Madlaina Signer, Marco Keller, Martin Chikov, Martin Raszyk, Mathieu Zufferey, Mikós Horváth, Nikola Djokic, Pascal Sommer, Petr Mitrichev, Rada Kamysheva, Remo Spichtig, Serge Balzan, Simon Meinhard, Tobias Feigenwinter, and Viera Klasovita.

I am grateful to Ema Skottova for her relentless support throughout the final months of writing, as well as for proofreading parts of this work. I acknowledge Balu, Möhrli, Samori, Camino, Charlie, Merlin, and Nemo.

Many thanks to Marlies Weissert, Mirella Rutz and Fiorella Meyer, for providing a smooth and pleasant interface to the administrative apparatus at ETH.

Finally, I would like to thank my family, Fabian Gehr, Lorena Gehr, Romina Gehr, Marvin Gehr, Manfred A. Gehr-Huber, Gabriela Gehr-Huber, Hedwig Huber and Erika Gehr, for their essential support.

CONTENTS

ACKNOWLEDGEMENTS ix
List of Figures xv
List of Tables xix
1 INTRODUCTION 1
1.1 Abstract Interpretation of Neural Networks 2
1.2 Symbolic Reasoning for Probabilistic Programs 7
1.3 Chapter Overview 13
1.3.1 Part i: Robustness of Neural Networks 13
1.3.2 Part ii: Symbolic Reasoning for Probabilistic Programs 14
1.3.3 Conclusion and Future Work 17

I ABSTRACT INTERPRETATION OF NEURAL NETWORKS
2 AI^2: ABSTRACT INTERPRETATION FOR ARTIFICIAL INTELLIGENCE 21
2.1 Neural Networks and Conditional Affine Transformations 25
2.2 Background: Abstract Interpretation 32
2.3 AI^2: AI for Neural Networks 36
2.3.1 Abstract Interpretation for CAT Functions 36
2.3.2 Neural Network Analysis with AI 39
2.4 Implementation of AI^2 40
2.5 Evaluation of AI^2 41
2.5.1 Experimental Setup 41
2.5.2 Discussion of Results 43
2.6 Comparing Defenses with AI^2 47
2.7 Related Work 48
2.8 Discussion 50
3 FAST AND EFFECTIVE ROBUSTNESS CERTIFICATION 51
3.1 Abstract Interpretation for Verifying Robustness of Neural Networks 52
3.2 Fast Zonotope Abstract Transformers 54
3.2.1 ReLU 54
3.2.2 Sigmoid 55
3.2.3 Tanh 56
3.3 Experiments 57
3.3.1 Experimental setup 57
3.3.2 Experimental results 57
3.3.3 Dataset Normalization 62
3.3.4 Neural Networks Evaluated 62
3.4 Discussion 63

II SYMBOLIC REASONING FOR PROBABILISTIC PROGRAMS

4 PSI: EXACT SYMBOLIC INFERENCE FOR PROBABILISTIC PROGRAMS 67
- 4.1 Overview . . . 69
  - 4.1.1 Analysis . . . 69
  - 4.1.2 Applications of PSI . . . 71
- 4.2 Symbolic Inference . . . 72
  - 4.2.1 Source Language . . . 73
  - 4.2.2 Symbolic Domain for Probability Distributions . . . 73
  - 4.2.3 Analysis of Expressions . . . 76
  - 4.2.4 Analysis of Statements . . . 77
  - 4.2.5 Final Result and Renormalization . . . 79
  - 4.2.6 Discussion . . . 79
- 4.3 Symbolic Optimizations . . . 81
  - 4.3.1 Algebraic Optimizations . . . 81
  - 4.3.2 Guard Simplifications . . . 81
  - 4.3.3 Symbolic Integration . . . 86
- 4.4 Evaluation . . . 88
  - 4.4.1 Comparison with Exact Symbolic Inference Engines . . . 89
  - 4.4.2 Comparison with Approximate Symbolic Inference Engine . 93
  - 4.4.3 Comparison with Approximate Numeric Inference Engines . 93
- 4.5 Related Work . . . 95
  - 4.5.1 Symbolic Inference . . . 95
  - 4.5.2 Probabilistic Program Analysis . . . 96
- 4.6 Discussion . . . 97

5 λPSI: EXACT INFERENCE FOR HIGHER-ORDER PROBABILISTIC PROGRAMS 99
- 5.1 Motivation and Overview . . . 100
- 5.2 The λPSI probabilistic programming language (PPL) . . . 102
- 5.3 A Symbolic Domain for Distributions . . . 106
  - 5.3.1 Representation . . . 107
  - 5.3.2 Interpretation . . . 108
  - 5.3.3 Examples . . . 109
  - 5.3.4 Comparison to PSI . . . 110
- 5.4 From Programs to Symbolic Representations . . . 111
  - 5.4.1 Translating Programs to the Symbolic Domain . . . 111
  - 5.4.2 Accounting for Error States . . . 113
- 5.5 Inference by Symbolic Simplification . . . 114
  - 5.5.1 Dirac Delta Substitution . . . 115
  - 5.5.2 Dirac Delta Linearization . . . 115
  - 5.5.3 Guard Simplifications . . . 117
  - 5.5.4 Symbolic Integration . . . 117
  - 5.5.5 Symbolic Disintegration . . . 118
  - 5.5.6 Comparison to PSI . . . 119

5.5.7 Limitations . . . 119
5.5.8 Correctness . . . 120
5.6 Evaluation . . . 120
5.6.1 Comparison to Previous Work . . . 121
5.6.2 Case Studies . . . 122
5.7 Related Work . . . 126
5.8 Discussion . . . 126
6 CONCLUSION AND FUTURE WORK 127
6.1 Neural Network Robustness . . . 128
6.1.1 Certification . . . 128
6.1.2 Defenses . . . 129
6.2 Symbolic Probabilistic Inference . . . 129
6.2.1 Additional Features . . . 130
6.2.2 Performance and Completeness . . . 133
6.2.3 Correctness . . . 134
6.2.4 Approximate Methods . . . 135
7 BIBLIOGRAPHY 137

LIST OF FIGURES

Figure 1.1 Adversarial attacks on neural networks. 4

Figure 1.2 Abstract interpretation overview. Sets of concrete neuron activations, starting with a norm ball of radius ϵ around input $\mathbf{x}$, are enclosed by concretizations (γ) of abstract elements that symbolically describe overapproximations of those intermediate results after each layer, up to the image of the input set $B_\epsilon(\mathbf{x})$ under the neural network f_θ. Robustness properties are verified by checking whether a' describes a set $\gamma(a')$ enclosed within a region C of safe outputs (not shown). 5

Figure 1.3 Standard interface of PSI for users. *Input*: A probabilistic program. *Output*: A symbolic description of the output distribution/properties. (Here, we have numerically evaluated the symbolic expressions for illustration, resulting in a plot of the output density, as well as truncated decimal values.) . 10

Figure 1.4 One component of the generic abstract transformer for the ReLU activation function, instantiated to the Zonotope domain. 14

Figure 1.5 Internal operation of PSI: Simplify symbolic expressions derived from the program (top) into a usable closed form (bottom). A small section of the unsimplified expression has been magnified. 15

Figure 1.6 Rejection sampling (top) vs. symbolic inference in λPSI (bottom). 16

Figure 2.1 Attacks applied to MNIST images [102]. 22

Figure 2.2 A high-level illustration of how AI^2 checks that all perturbed inputs are classified the same way. AI^2 first creates an abstract element A_1 capturing all perturbed images. (Here, we use a 2-bounded set of zonotopes.) It then propagates A_1 through the abstract transformer of each layer, obtaining new shapes. Finally, it verifies that all points in A_4 correspond to outputs with the same classification. 24

Figure 2.3 Definition of CAT functions. 26

Figure 2.4 One example computation for each of the three layer types supported by AI^2. 26

Figure 2.5 The operation of the transformed max pooling layer. 30

Figure 2.6 (a) Abstracting four points with a polyhedron (gray), zonotope (green), and box (blue). (b) The points and abstractions resulting from the affine transformer. 33

Figure 2.7 Illustration of how AI^2 overapproximates neural network states. Blue circles show the concrete values, while green zonotopes show the abstract elements. The gray box shows the steps in one application of the ReLU transformer ($ReLU_1$). 36
Figure 2.8 Abstract transformers for CAT functions. 38
Figure 2.9 Verified properties by AI^2 on the MNIST and CIFAR convolutional networks for each bound $\delta \in \Delta$ (x-axis). 42
Figure 2.10 Verified properties as a function of the abstract domain used by AI^2 for the 9×200 network. Each point represents the fraction of robustness properties for a given bound (as specified in the legend) verified by a given abstract domain (x-axis). . 44
Figure 2.11 Average running time of AI^2 when proving robustness properties on MNIST networks as a function of the abstract domain used by AI^2 (x-axis). Axes are scaled logarithmically. 44
Figure 2.12 Comparing the performance of AI^2 to Reluplex. Each point is an average of the results for all 60 robustness properties for the MNIST networks. Each point in (a) represents the average time to completion, regardless of the result of the computation. While not shown, the result of the computation could be a failure to verify, timeout, crash, or discovery of a counterexample. Each point in (b) represents the fraction of the 60 robustness properties that were verified. 45
Figure 2.13 Box-and-whisker plot of the verified bounds for the Original, GSS, Ensemble, and MMSTV networks. The boxes represent the δ for the middle 50% of the images, whereas the whiskers represent the minimum and maximum δ. The inner-lines are the averages. 49
Figure 3.1 Two zonotope approximations for the ReLU function parameterized by the slope λ. 54
Figure 3.2 Zonotope approximation for the sigmoid function parameterized by slope λ, where $0 \leqslant \lambda \leqslant \min(f'(l_x), f'(u_x))$. 56
Figure 3.3 Comparing the performance and precision of DeepZ with the state of the art. 58
Figure 3.4 Verified robustness by DeepZ on the MNIST networks with ReLU activations. 59
Figure 3.5 Verified robustness by DeepZ on the MNIST networks with Sigmoid and Tanh activations. 60
Figure 3.6 Verified robustness by DeepZ on the CIFAR10 networks with ReLU activations. 61
Figure 3.7 Verified robustness by DeepZ on the CIFAR10 networks with Sigmoid and Tanh activations. 62
Figure 4.1 ClickGraph Example 70
Figure 4.2 PSI's Source Language Syntax 72

Figure 4.3 Symbolic domain for probability distributions 73
Figure 4.4 Symbolic Analysis of Expressions 75
Figure 4.5 Symbolic Analysis of Statements 77
Figure 4.6 Analysis of Statements - Helper Functions 78
Figure 4.7 `Tracking.query2`: PSI (solid; exact) and SVE (dashed). 93
Figure 4.8 `ClickGraph`: PSI (solid; exact) and Infer.NET (dashed). 93
Figure 4.9 `AddFun/max`: PSI (solid; exact) and Infer.NET (dashed). 93
Figure 5.1 Inference on "Epistemic States": approximate inference in Webchurch (top) vs. exact inference in λPSI (bottom). 100
Figure 5.2 A λPSI program (left) and its exact joint probability distribution (right) computed by λPSI. 103
Figure 5.3 Core syntax of λPSI (n, x, uop, and bop denote constants, variables, unary, and binary operations, respectively). 103
Figure 5.4 Full syntax of the higher-order probabilistic programming language λPSI (extension of Fig. 5.3). 104
Figure 5.5 Symbolic domain for expressing probability distributions. We write $e[x]$ to denote that x is a free variable in e. The highlighted elements are fundamental to λPSI and new compared to PSI. 106
Figure 5.6 Key translation rules, ignoring error states. The rules are recursive and we write $\boxed{a}$ to denote the translation of a. . . . 110
Figure 5.7 Nested inference example. 112
Figure 5.8 Selected steps of deriving a simplified representation of the code in Fig. 5.7. The terms affected by substitution (§5.5.1), linearization (§5.5.2), guard simplification (§5.5.3), and symbolic integration (§5.5.4) are highlighted. Equalities annotated with $\star$ denote recursive translation and simplification. 114
Figure 5.9 Simplifying Dirac deltas and Lebesgue measures. Here, f' is the derivative of f. 115
Figure 5.10 The function $f(x) = 1 + [-1 \leqslant x] \cdot [x \leqslant 1] \cdot (1 - x^2)$. 117
Figure 5.11 Bayesian linear regression. 122
Figure 5.12 Bayesian linear regression: posterior of $y = f(x)$ after 5 samples.123
Figure 5.13 Conditional probability $\Pr[x < X \mid y > Y]$ depending on X and Y. 124
Figure 5.14 Information-theoretic quantities associated with discrete distributions and an application: generalization capacity (of sorting algorithms). 125

LIST OF TABLES

Table 3.1 Neural network architectures used in our experiments. . . . 58
Table 3.2 Verified robustness by DeepZ on the ConvBig and ConvSuper networks trained with DiffAI. 60
Table 4.1 PDF and Correctness Conditions for Several Primitive Distributions. 74
Table 4.2 Comparison of Exact and Interactive Symbolic Inference Approaches. 90
Table 4.3 Performance (in s) of Exact and Interactive Symbolic Inference Approaches. 92
Table 4.4 R2 Benchmark Precision and Analysis Time 94
Table 4.5 Infer.NET Benchmark Precision and Analysis Time. 95
Table 5.1 Probabilistic programs used in evaluation (31 in total). For each program, we indicate if it involves higher-order functions ($\rightarrow$), nested inference ($\rightsquigarrow$), first-class expectations ($\mathbb{E}$), continuous distributions ($\wedge$), continuous observations ($\wedge$), or symbolic parameters ($\boxed{a}$). SocialCognition and TotalVarDist have multiple variants. Some programs are not expressible in Hakaru ($-$), while others lead to errors ($\times$), unsimplified ($\blacksquare$) or incorrect ($\times\times$) results. [a]Not directly expressible; rewritten as first-order programs without function calls, multiple manual steps. [b]For concrete instantiation of symbolic parameters. 120

1

INTRODUCTION

Intelligent systems are being widely deployed in areas that are critically important to society, such as medical diagnosis and devices, vehicle control, police and justice systems, military, financial services, credit rating, insurance policies, stock market prediction, weather reports, climate models, and many more. It is hence crucial to understand how those systems operate and to make their decision making consistent, as well as easy to explain in a satisfying manner to those affected by the decisions. In particular, automated decision making systems have to be robust to changing circumstances, including explicit attacks by adversaries. We must therefore strive for strong guarantees on formal correctness, safety and fairness of important decisions made by automated systems, as they will ultimately shape the future of humanity.

However, this appears to be significantly harder than achieving acceptable performance on cultivated test sets in the standard machine learning setting, where we assume independent samples from a data source with an unchanging (conditional) distribution. This is in contrast to more traditional approaches based on symbolic methods, which are explainable by construction. This apparent strength is also one of the most important drawbacks of symbolic methods: the most impressive feats of modern statistical machine learning are precisely on problems where we do not actually understand how to manually explain to a machine the exact symbolic steps required for a solution.

It is therefore plausible that in order to address the aforementioned challenges, the automated systems of the future should incorporate a combination of aspects of machine learning systems and symbolic methods. This leads us to the main research question driving this dissertation:

> What are ways to design more effective automated intelligent systems by incorporating symbolic methods in addition to statistical approaches?

MAIN CONTRIBUTIONS We have selected two major directions of research in intelligent systems and we contribute to both as part of this dissertation. To this end, we have developed two distinct novel tools that combine statistical approaches with symbolic methods and address problems that are out of reach for prior work. Both systems draw from programming language research.

Our contributions are therefore divided into two lines of work:

- *Abstract Interpretation of Neural Networks*: We instantiate abstract interpretation to the setting of neural network robustness certification. Our system achieves certification results that are out of reach of prior methods. This work is discussed in Part i of this dissertation.
- *Symbolic Reasoning for Probabilistic Programs*: We develop exact techniques for probabilistic inference based on automated manipulation of symbolic terms, including the first exact inference technique that supports at the same time: discrete and continuous distributions, higher-order functions, and first-class inference. This work is discussed in Part ii of this dissertation.

We now briefly introduce both research directions, we define the more specific problems we address within them, and we give a short overview of our methods.

1.1 ABSTRACT INTERPRETATION OF NEURAL NETWORKS

NEURAL NETWORKS Artificial neural networks are a popular class of statistical models [61]. For our purposes, a neural network is numerical program $f_\theta \colon \mathbb{R}^m \to \mathbb{R}^n$, parameterized by parameters θ, such that (we can pretend that) there is a gradient $\nabla_\theta f_\theta(x)$ for all $x \in \mathbb{R}^m$, which we can efficiently compute using the chain rule with the backpropagation algorithm [136]. In practice, neural networks are often implemented approximately using floating-point arithmetic. In the most standard setting of classification, we additionally have a differentiable loss function $\mathcal{L} \colon \mathbb{R}^n \times [n] \to \mathbb{R}$ (often, this is cross-entropy). Given a finite training set

$$T = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)\} \subseteq \mathbb{R}^m \times [n]$$

drawn from some underlying, unknown distribution $\mathcal{D}$, we heuristically optimize the function $\mathbb{E}_{(\mathbf{x},y)\sim\mathcal{U}(T)}[\mathcal{L}(f_\theta(\mathbf{x}), y)]$ with respect to θ using some variant of gradient descent. We then hope that f_θ together with the optimization procedure has a high enough capacity to capture the underlying distribution $\mathcal{D}$ but is still not able to memorize the training set T, resulting in generalization: we want to obtain a similar expected loss $\mathbb{E}_{(\mathbf{x},y)\sim\mathcal{D}}[\mathcal{L}(f_\theta(\mathbf{x}), y]$ on the underlying distribution $\mathcal{D}$ as on the finite subsample T where we have explicitly optimized it to be small.

PROBLEM STATEMENT While this works surprisingly well, unfortunately, we cannot expect the neural network to do well on distributions $\mathcal{D}' \neq \mathcal{D}$. This is troubling in particular in case $\mathcal{D}$ is easy to turn into $\mathcal{D}'$ by an adversary or if $\mathcal{D}$ and $\mathcal{D}'$ are even indistinguishable under manual inspection. Such $\mathcal{D}'$ have been observed to typically exist for neural networks in many application domains. Fig. 1.1 demonstrates multiple types of such adversarial attacks on neural networks. Pei et al. [131] (Fig. 1.1a) show, among other similar adversarial attacks, how a small number

of black rectangles can be added to an image of a road to fool neural networks into driving off the road. Sharif et al. [149] (Fig. 1.1b) show how to impersonate other people by wearing colorful glasses. Goodfellow et al. [60] (Fig. 1.1c) attack neural networks by adding small imperceivable perturbations to inputs, in order to fool a neural network, for example, into classifying a panda as a gibbon. This kind of attack is particularly subversive, as it is hard to detect. However, compared to other types of attacks, the task of certifying its absence is a particularly good match for static program analysis.

In this work, we will handle general *safety properties*. Our problem statement is:

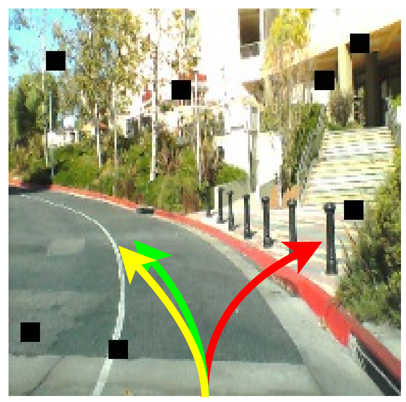
Given a neural network $f_\theta\colon \mathbb{R}^m \to \mathbb{R}^n$, a region $X \subseteq \mathbb{R}^m$ of possible inputs, and a region $C \subseteq \mathbb{R}^n$ of safe outputs, automatically verify that for each possible input $\mathbf{x} \in X$, the corresponding output $f(\mathbf{x})$ is safe, i.e., $f(\mathbf{x}) \in C$.

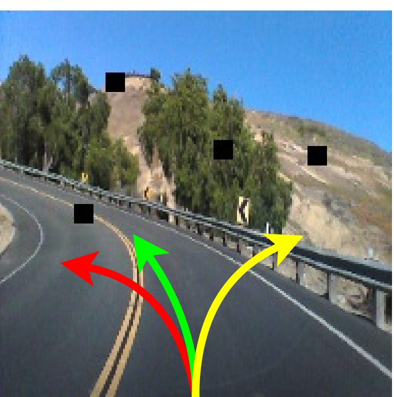
A special case of safety is *local robustness*, where the goal is to defend against a malicious adversary who is able to slightly perturb inputs to the neural network after they have been sampled. The constraints on the capabilities of the adversary are chosen to be a proxy for a human's ability to detect that the input has been tampered with. Given a single example $(\mathbf{x}, y)$, the adversary can select a perturbed input $\mathbf{x}' \in X$, where $X \ni \mathbf{x}$ is some region around $\mathbf{x}$. We then let the safe set C be some set of outputs of f_θ such that the neural network correctly classifies the perturbed inputs given that $f_\theta[X] \subseteq C$.

Note that we will address this problem in a best-effort but sound manner: our approach may fail to verify true properties, but if it verifies a property, it indeed has to hold. Under this constraint, the approach should be as precise as possible, i.e., we want to verify as many of the true properties as we can.

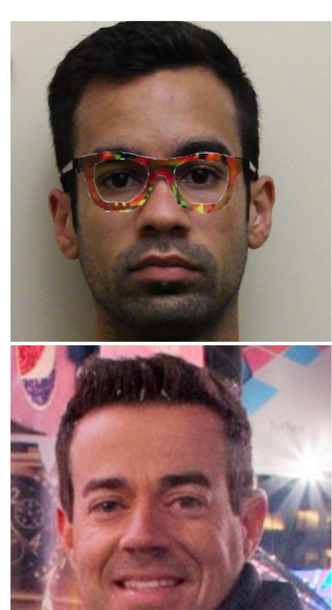
ABSTRACT INTERPRETATION Abstract interpretation [35] is a general theory of this kind of sound overapproximations of program behaviors. An abstract domain $\mathcal{A}$ contains abstract elements $a \in \mathcal{A}$ that describe sets of program states $\sigma \in \Sigma$ in a finite, symbolic way. The concretization function $\gamma\colon \mathcal{A} \to \mathcal{P}(\Sigma)$ maps a symbolic abstract element to the concrete set of program states it describes. For some function $f\colon \Sigma \to \Sigma$, the concrete transformer $T_f\colon \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ maps a set X to its image $f[X]$. The abstract transformer does the same to symbolic descriptions, possibly losing some precision in the process. Formally, an abstract transformer $T_f^\#\colon \mathcal{A} \to \mathcal{A}$ has to satisfy the soundness condition $T_f(\gamma(a)) \subseteq \gamma(T_f^\#(a))$ for all $a \in \mathcal{A}$.

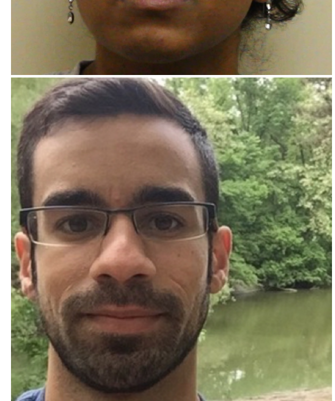
(Disclaimer: While this more abstract treatment of abstract interpretation is already sufficient for our purposes, we note that abstract interpretation usually also assumes a partial order $\sqsubseteq$ on abstract elements. We then also have an abstraction function $\alpha\colon \mathcal{P}(\Sigma) \to \mathcal{A}$. The functions γ and α are then also assumed to be monotone and we have $\alpha(\gamma(a)) \sqsubseteq a$ for all $a \in A$ as well as $X \subseteq \gamma(\alpha(X))$ for all $X \in \mathcal{P}(\Sigma)$, such that the entire structure forms a Galois connection, which is an example of adjoint functors. In order to treat unbounded loops, a crucial ingredient of abstract inter-





(a) Black rectangle defects in natural images. Each image fools one of three different classifiers [131].





(b) Physically realizable impersonation attack [149].

$+.007\times$

$=$

$\mathbf{x}$	$\operatorname{sign}(\nabla_{\mathbf{x}} J(\theta, \mathbf{x}, y))$	$\mathbf{x}+$ $\epsilon \cdot \operatorname{sign}(\nabla_{\mathbf{x}} J(\theta, \mathbf{x}, y))$
"panda"	"nematode"	"gibbon"
57.7% confidence	8.2% confidence	99.3% confidence

(c) Fooling a neural network with a tiny, imperceivable global perturbation [60].

Figure 1.1: Adversarial attacks on neural networks.

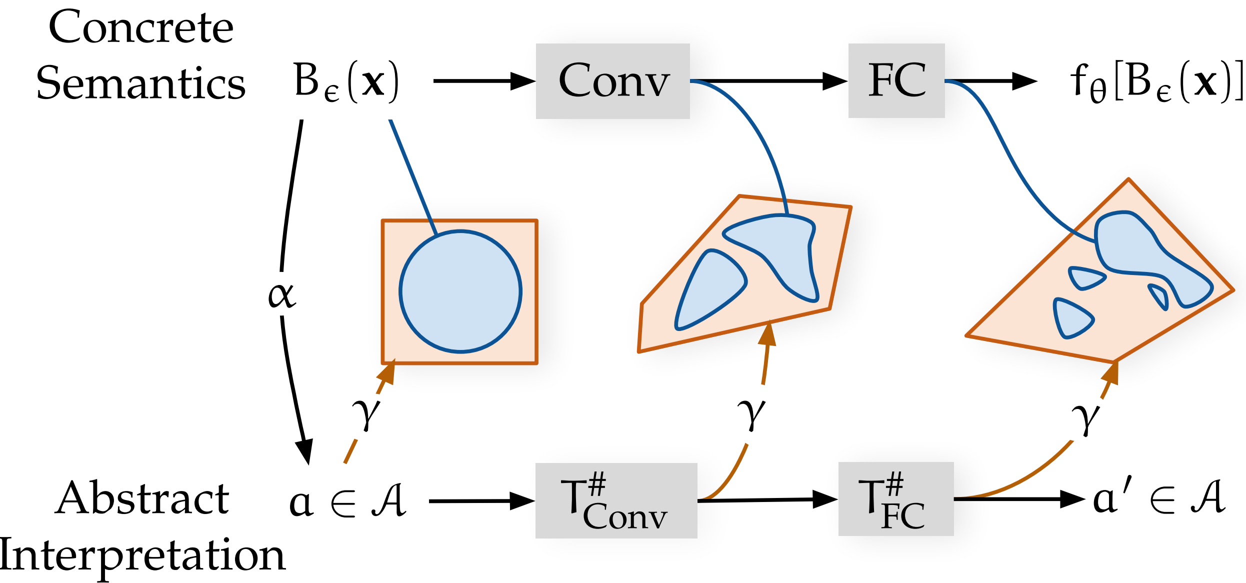


Figure 1.2: Abstract interpretation overview. Sets of concrete neuron activations, starting with a norm ball of radius ϵ around input $\mathbf{x}$, are enclosed by concretizations (γ) of abstract elements that symbolically describe overapproximations of those intermediate results after each layer, up to the image of the input set $B_\epsilon(\mathbf{x})$ under the neural network f_θ. Robustness properties are verified by checking whether a' describes a set $\gamma(a')$ enclosed within a region C of safe outputs (not shown).

pretation is fixed-point iteration. As our application does not include unbounded loops, we do not require fixed-point iteration nor any widening operators.)

Given the neural network f_θ, the region X and abstract domains A_n for concrete domains $\mathbb{R}^n$, we can construct an abstract transformer $T^\#_{f_\theta} : \mathcal{A}_m \to \mathcal{A}_n$ for f_θ. We can then select an abstract element a with $X \subseteq \gamma(a)$ to get a symbolic description $a' = T^\#_{f_\theta}(a)$ of some superset of the image of X under f_θ. From soundness of the abstract transformer, we obtain $f_\theta[X] \subseteq \gamma(a')$. It hence suffices to show that $\gamma(a') \subseteq C$. Then, all outputs in $f_\theta[X]$ will also be safe. Fig. 1.2 illustrates the relationship of concrete and abstract transformers. Analysis with input set $X = B_\epsilon(\mathbf{x})$ proceeds compositionally through each neural network layer.

CONTRIBUTION In this thesis, we demonstrate how to do all of those steps. We have developed AI^2, a system for abstract interpretation of neural networks. This poses some unique challenges: (i) the scale of those neural networks (hundreds or thousands of live variables at one time) and (ii) the kind of operations that are executed, in particular, our abstract interpretation has to be sound with respect to floating-point roundoff error. We make the following contributions:

- **Abstract Interpretation Framework for Neural Networks** We instantiate the general theory of abstract interpretation for analysis of local robustness of layered[1] neural networks.
- **Generic Abstract Transformers** We provide abstract transformers for common types of neural network layers that are based on operations typically supported by implementations of abstract domains.
- **Specialized Abstract Transformers for Zonotope Domain** We design specialized abstract transformers that exploit useful properties of the Zonotope domain to gain scalability and precision.
- **Scalable Certifiers** We present AI^2 and DeepZ. Those are systems in which we implemented the ideas mentioned above to obtain high-performance analysis of comparatively large neural networks and input region specifications, all while correctly handling floating-point roundoff error. Both of our systems achieve certification results that are out of reach of prior methods.

IMPACT Among many others, AI^2 has inspired the following follow-up works:

- **Training for Certifiability**: DiffAI [114] performs abstract interpretation within an automatic differentiation framework, such that neural networks can be optimized directly for certifiable robustness on the training set.
- **Abstract Interpretation with Backsubstitution**: DeepPoly [155], explores another trade-off between speed and precision. It is based on a linear relaxation of the constraints that can be derived between neuron activations when symbolically evaluating the neural network. Concrete bounds on neuron activations are derived by backsubstituting constraints until reaching the input layer. (Like AI^2 and DeepZ, DeepPoly is sound with respect to floating-point roundoff error.) In its evaluation, this work already explores certification of invariance of neural network classifications against rotations of the input image parameterized by a symbolic rotation angle restricted to some range.
- **Joint Relaxation**: kPoly [154] further improves on DeepPoly by grouping multiple neurons for linear relaxation of ReLU activations instead of relaxing all neurons independently. It is based on exact computation of convex hulls. PRIMA [120] extends this approach to arbitrary activation functions and is based on an efficient approximation of convex hulls to improve scalability.
- **Combination with MILP**: RefineZono [156] combines abstract interpretation using DeepZ (presented in Chapter 3) with mixed integer linear programming (MILP) to provide certification results that are more precise than DeepZ in

1 Our work is easily extended to more general patterns of data flow, but we focus on neural networks organized in layers to keep the presentation simple.

a more scalable way than previous approaches based on MILP. (Like other MILP-based approaches, and in contrast to AI^2 and DeepZ, RefineZono is not sound with respect to floating-point roundoff error.)

- **Geometric Robustness**: DeepG [11], inspired by evaluation results of DeepPoly on rotation transformations, certifies invariance of neural network classifications against compositions of multiple geometric image transformations where each of them is parameterized by symbolic parameters. DeepG supports commonly-used image interpolation algorithms for all transformations.
- **ERAN Abstract Interpretation Framework**: ERAN [2] is our latest neural network analyzer based on abstract interpretation. It is the direct successor to AI^2 and provides implementations of DeepZ, DeepPoly, kPoly and DeepG, among others.
- **Probabilistic Abstract Interpretation**: Generalizing the framework of AI^2, ApproxLine [115] performs probabilistic abstract interpretation of compositions of neural networks, where the input sets are line segments and other curves. Instead of sets of neuron activations, ApproxLine's abstract domain represents sets of distributions over neuron activations.
- **Abstract Interpretation of Automatic Differentiation**: In addition to the forward pass, Jordan and Dimakis [85] also perform abstract interpretation of the backward automatic differentiation pass of a neural network, deriving bounds on the Jacobian matrix of the neural network within some input region. The approach builds on DeepZ, with an additional (non-linear) multiplication transformer for zonotopes. It also proposes a different way to derive Zonotope transformers that results in more precise transformers for neural network activation functions.

Overall, the line of work around AI^2 presented in this thesis has sparked substantial follow-up in the research community.

1.2 SYMBOLIC REASONING FOR PROBABILISTIC PROGRAMS

An important limitation of many popular machine learning approaches, including (standard) neural networks is that the results of training, and in turn, also the final results of classification or regression, are hard to interpret and therefore errors are typically unexplainable.

Probabilistic programming is an alternative modern approach to automated reasoning that is rooted in more traditional statistical methods. Probabilistic programs represent a statistical model that can then be conditioned on concrete evidence in a Bayesian fashion: Consider a weighted set of assignments of values to program vari-

ables.[2] Those values evolve under deterministic program statements independently, and if the resulting assignments for multiple different original assignments coincide, we combine their weight. Probabilistic programs then additionally have methods to sample new random values. Those may cause assignments to split into multiple other assignments, where each of the resulting assignments gets some portion of the original weight before the results for each original assignment are combined. To condition on concrete evidence, we simply ignore all assignments that are inconsistent with that evidence and eventually renormalize, such that the total weight of all assignments is 1, while weights stay the same relative to each other. Note that (already for just discrete random choices), the number of possible assignments can increase exponentially in the number of executed program statements and the resulting values can then be manipulated arbitrarily by a program from a (possibly) Turing complete language.

The semantics of conditioning directly implements the definition of conditional probability:

$$\Pr[A \mid B] = \Pr[A \cap B] / \Pr[B]$$

The weight of an assignment α is $\Pr[\alpha]$. A is a variable set of assignments of values to program variables and B is the set of assignments that are consistent with the observed evidence. Here, $A \cap B$ ignores all assignments in A that are inconsistent with the observed evidence. Effectively, this sets the probability of each assignment outside of the set B to 0. The probability of the remaining assignments then has to add up to 1. Their total probability is precisely $\Pr[B]$, so we divide by this probability to renormalize.

Consider the following probabilistic program:

```
def main(y){
    X := sampleX();
    Y := sampleY(X);
    observe(Y = y);
    return X;
}
```

This program samples a random value X and then, depending on X, it samples another random value Y. This represents a joint density $\Pr[X = x, Y = y]$ of X and Y programmatically, as a product

$$\Pr[X = x, Y = y] = \Pr[X = x] \cdot \Pr(Y = y \mid X = x),$$

where `sampleX` represents the density $\Pr[X = x]$ and `sampleY` represents the conditional density $\Pr[Y = y \mid X = x]$. In this model, X is unknown and we only observe the value Y that was derived from it. This is a common pattern in probabilistic programming. The program above computes a result distributed according to the

2 If continuous distributions are involved, the semantics becomes a bit more delicate than in the discrete case, but intuitively we can think of having an infinite set of assignments where each has an infinitesimal weight.

conditional density $\Pr[X = x \mid Y = y]$. Plugging in $A = \{X = x\}$ and $B = \{Y = y\}$ into the definition of conditional probability above, we get

$$\Pr[X = x \mid Y = y] = \Pr[Y = y, X = x] / \Pr[Y = y],$$

or, equivalently, using the factoring implemented by the program,

$$\Pr[X = x \mid Y = y] = \Pr[Y = y \mid X = x] \cdot \Pr[X = x] / \Pr[Y = y].$$

This is Bayes' theorem for density functions. In general, the probabilistic program implements a *generative model* that assigns probabilities to different outcomes. Conditioning partitions the assignment of values to program variables into observed and *latent* (non-observed) random values. We can then infer facts about the latent values based on the observed values. Probabilistic programs help automate this kind of Bayesian reasoning.

Probabilistic programs have a fully interpretable formal semantics in terms of probability theory. Hence, errors made by the model can in principle be explictly traced back to some specific modeling inaccuracy or simplification. One benefit of probabilistic programming is that we can run different inference algorithms on the same model (which is expressed as a probabilistic program). However, a big caveat is that the semantics of probablistic programs is intractable to compute in general, even for models that have only discrete probabilistic choices. This is because probabilistic programs can model arbitrary counting problems. Even making non-trivial approximation guarantees is NP-hard. This introduces another kind of non-interpretability: any approximate inference algorithm will fail to produce inference results accurately reflecting the underlying model on some probabilistic programs. This introduces a disconnect between the model that is represented by the probabilistic program and the inference results that are based on it: approximate inference can be a source of errors that are hard to interpret. In particular, it may not be clear if issues are caused by bad modeling or by bad inference.

PROBLEM STATEMENT Most prior work on inference for probabilistic programs cites intractability of inference as a justification for why approximation is necessary. However, note that this does not work around the intractability: approximate inference approaches will only ever work very well for some subset of probabilistic programs. In this work, we recognize that approximate and exact inference are similar in this respect and that therefore, exact inference deserves just as much consideration. We address the following problem statement:

> Given a probabilistic program with both discrete and continuous random choices, symbolically compute properties of the output distribution, such as probabilities or expectations.

As computing practically useful results is intractable (even undecidable) in general, we do this in a best-effort manner. However, note that in contrast to many popular

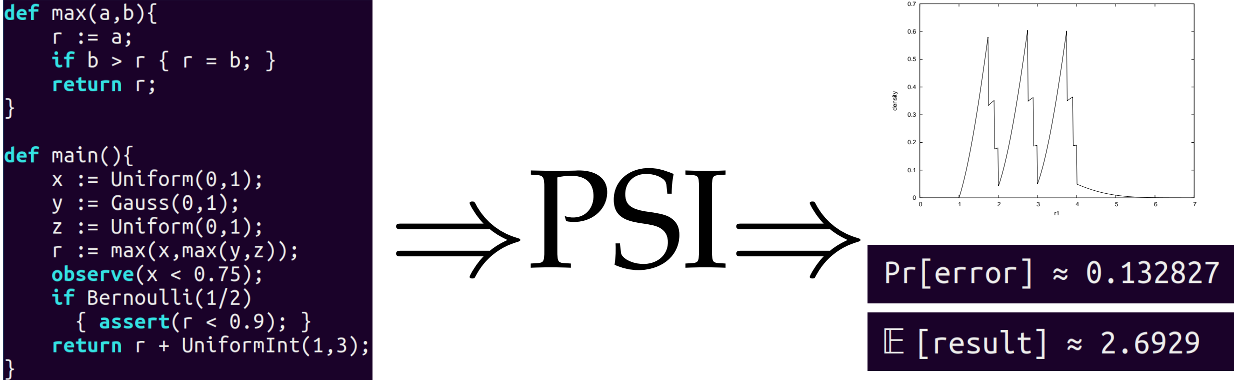


Figure 1.3: Standard interface of PSI for users. *Input*: A probabilistic program. *Output*: A symbolic description of the output distribution/properties. (Here, we have numerically evaluated the symbolic expressions for illustration, resulting in a plot of the output density, as well as truncated decimal values.)

approximate approaches, we aim to make it very easy to detect failure of inference. As a significant simplification, input programs need to have bounded loops and our inference backends do not support general recursion. Lifting those simplifications is an exciting research direction for future work.

CONTRIBUTION We have created PSI, a new convenient probabilistic programming language in which users can express their statistical models and which supports powerful exact symbolic inference. Fig. 1.3 illustrates PSI's user interface: The user writes a probabilistic program and PSI symbolically computes properties of the output distribution, such as density, probability of error, and expectation. We have then extended PSI to λPSI, which adds features such as tuples, arrays, higher-order functions, polymorphism and dependent typing, as well as nested inference. Exact symbolic inference is useful in many ways:

- **Precision**: When PSI succeeds, the produced exact results are typically more useful than approximations.
- **Parameters**: Symbolic inference can support symbolic parameters, such that the inference procedure can solve a potentially infinite number of inference queries at once. (I.e., we can solve all pertinent problems in infinitely many universes at once.) Symbolic parameters can in turn be optimized so that the resulting distribution has desirable properties.
- **Speed**: In some cases, symbolic inference is more efficient than alternative approaches that are readily available. This can be particularly pronounced with deeply nested inference problems.
- **Baseline**: PSI can be used as a baseline to motivate research on specialized or approximate inference methods, it is often necessary to point out that exact

inference is intractable in general. PSI provides an easy-to-use way to justify such statements experimentally.

- **Ground Truth**: To validate specialized or approximate inference methods by comparing to an exact ground truth. For example, if it is helpful to be able to highlight some special property of the resulting distribution that is correctly captured by some approximate inference method, it is helpful to validate that the exact underlying distribution in fact does have that property.

- **Mockup**: PSI can be used as a generic drop-in during development, before more specialized or approximate procedures are available. It can then automatically serve as both baseline and ground truth to validate specialized or approximate procedures.

- **Teaching**: To teach probabilistic program semantics, in particular to dispel the common notion that probabilistic programs are inherently fuzzy and approximate.

It can also be useful as a component of other approaches, for example:

- Verification of probabilistic programs.

- Compiler optimizations on probabilistic programs.

IMPACT Indeed, PSI has already been used to support subsequent research in a variety of contexts. For example,

- **Differential Privacy**: PSI has been used to verify counterexamples to differential privacy [17, 169]. The inputs and sets of outputs witnessing a violation of differential privacy are first obtained using heuristic approaches, then PSI is used to exactly determine the strength of the counterexample.

- **Privacy under Bayesian Inference**: Beyond differential privacy, SPIRE [97] uses PSI as a language to define priors on database contents as well as queries on those database contents. The exact inference backends are then used to compute probabilities that serve as a basis for enforcement of powerful Bayesian privacy constraints, including automated synthesis of new programs that satisfy the privacy constraints while still providing some useful data related to the query.

- **Networks**: Bayonet [52] uses PSI as a backend for verifying properties of software-defined computer networks. This served as inspiration for Net-Dice [159], a custom tool that can efficiently compute sound bounds on probabilities of events in computer networks, specialized to common protocols.

- **Discrete Exact Inference**: Dice [78] is a probabilistic programming language with specialized support for exact inference in discrete probabilistic programs that comes with a formal proof of correctness. PSI is used as a baseline.
- **Specialized Exact Inference**: SPPL [137] presents specialized exact inference algorithms and datastructures for a powerful class of probabilistic programs representing sum-product networks. In contrast to PSI, SPPL provides guarantees on completeness and efficiency by restricting the expressiveness of the input language. PSI is used as a baseline.
- **Incremental Exact Inference**: ISymb [174] extends PSI to support fast incremental updates of exact inference results under small changes of input data. PSI is also used as a baseline to demonstrate that the incremental inference approach is faster than recomputing the result from scratch after each change.
- **Bias Analysis**: FairSquare [7] is a tool that quantifies the extent to which programs, most notably statistical models obtained using machine learning, are biased against protected groups. The authors use PSI as a baseline that demonstrates that it is hard to obtain a closed-form solution for some of their benchmark examples. They then develop a custom approach to rigorously bound probabilities using an SMT-based enumeration approach with convergence guarantees.
- **Sensitivity Analysis**: PSense [81] uses PSI as a basis for analyzing the sensitivity of a probabilistic program's output distribution to changes in distribution parameters within the program.
- **Probabilistic Error Analysis**: Given a distribution on program inputs, Lohar et al. [104] compute sound bounds on the probability that programs take the wrong path through a program with discrete conditionals if computations using exact real numbers are replaced by floating-point approximations. PSI is used as a component of a baseline approach that also provides ground truth to validate a more scalable overapproximate approach that the authors specifically developed for this application.
- **Differential Testing**: ProbFuzz [43] is a system for automated testing of probabilistic programming systems that employs a variety of testing approaches and uses PSI to compute exact ground truth results for differential testing.
- **Inference Compilation**: Walia et al. [167] compile probabilistic programs with arrays into fast code that can then perform inference on data (using floating-point arithmetic). PSI is used as a baseline.
- **Probabilistic Programming on the Edge**: Statheros [100] is a probabilistic programming language that creates machine code performing MCMC inference on a given probabilistic program with fixed-point arithmetic. It is targeted

towards microcontrollers. The authors use PSI to obtain ground truth values for evaluation.

- **Weighted Model Integration**: PSI's robust integration engine and Iverson bracket simplifier has been used to extend knowledge compilation, previously used for weighted model counting, to support continuous variables and weighted model integration [42, 90].

- **Quantum Programming**: The language and type system of λPSI served as the basis for Silq [18], a high-level quantum programming language.

The capabilities of PSI will only increase in the future, as we systematically explore and support downstream applications that are in turn providing practically important benchmark programs. At the same time, we can extend the input language and write libraries, such that it becomes more easily applicable to a larger set of problems. In this way, we will be able to combine ideas from different specialized inference algorithms developed for different domains, to the benefit of all domain-specific applications of PSI. Meanwhile, PSI becomes a stronger baseline in all its application domains, fostering future research in exact inference.

1.3 CHAPTER OVERVIEW

We now give a short overview of the remaining contents of this dissertation. It is organized into two parts, corresponding to the two main research directions we pursued. Each part has two chapters. Finally, we conclude the dissertation in Chapter 6 with a list of future work items.

1.3.1 *Part i: Robustness of Neural Networks*

In the first part of the thesis, we present our tools for certifying robustness of neural networks to small input perturbations.

AI^2: SAFETY AND ROBUSTNESS CERTIFICATION OF NEURAL NETWORKS WITH ABSTRACT INTERPRETATION This is our initial system for abstract interpretation of neural networks. It is based on generic operations supported by existing abstract domains. In particular, we introduce the Zonotope domain in the context of neural network robustness certification and design generic abstract transformers for neural networks with ReLU activations, based on operations already supported by existing implementations of abstract domains. AI^2 thereby leverages existing implementations of abstract interpretation for neural network analysis.[3] We present AI^2 in Chapter 2.

3 Although we had to fix some bugs in existing libraries, as they had never been validated at the scale required by our system.

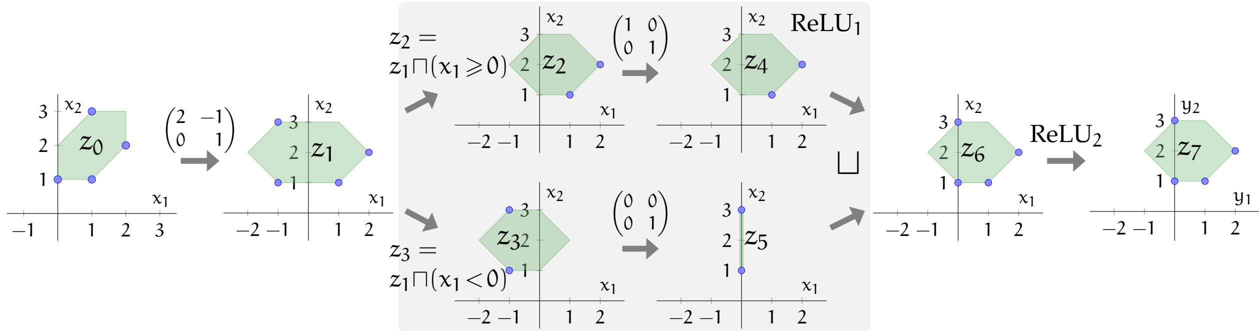


Figure 1.4: One component of the generic abstract transformer for the ReLU activation function, instantiated to the Zonotope domain.

FAST AND EFFECTIVE ROBUSTNESS CERTIFICATION AI^2 already produces better results than prior work, but it suffers from a few limitations due to its generic nature. Fig. 1.4 shows some intermediate results computed by one of AI^2's generic transformers. Note how it has to split the abstract element on the condition $x_1 \geqslant 0$. To this end, it uses the meet ($\sqcap$) and join ($\sqcup$) operations, which are particularly expensive and imprecise for the Zonotope domain. Furthermore, this computation has to be performed for all components of the layer in order and cannot be easily parallelized. Our system DeepZ overcomes those limitations using a custom implementation of the Zonotope domain with manual implementations of specialized transformers for neural network layers exploiting specific properties of the Zonotope domain. I.e., while AI^2 addresses the generic question of how to apply 40 years of existing research in abstract domains to neural networks, DeepZ is specialized to today's neural networks. An important feature of our specialized Zonotope transformers for neural network activation functions in DeepZ is that they do not use the expensive and imprecise meet and join operations. Furthermore, they operate on all neurons of a given layer independently, allowing easy parallelization. We demonstrate that, on a set of benchmark neural networks and robustness specifications, the analysis we implemented in DeepZ is at the same time faster and more precise than the analysis with the more generic transformers of AI^2 instantiated with the Zonotope domain. We present DeepZ in Chapter 3.

1.3.2 *Part ii: Symbolic Reasoning for Probabilistic Programs*

Note that to train a neural network, we optimize a statistical loss function. However, the final result of training is a deterministic function, whose behavior AI^2 and DeepZ analyze on sets of concrete inputs. In contrast, in the second part of the thesis, we explore how we can use symbolic methods to reason about probability distributions explicitly. Here, inputs and behaviour can both be random, but the methods we use to analyze them are fully deterministic.

⇓ simplify

```
Pr[error] = ((-(d/dx)⁻¹[e^(-x²)](9·⅒·1⁄√2)·2·1⁄√π+2)·9+2)·1⁄40
          ≈ 0.132827
```

Figure 1.5: Internal operation of PSI: Simplify symbolic expressions derived from the program (top) into a usable closed form (bottom). A small section of the unsimplified expression has been magnified.

PSI: EXACT SYMBOLIC INFERENCE FOR PROBABILISTIC PROGRAMS As we have mentioned above, PSI is our probabilistic programming language and system that supports exact symbolic probabilistic inference for first-order probabilistic programs with real variables, unrolled loops, and unrolled arrays. PSI operates by translating the input program into a symbolic domain statement by statement. The system then simplifies the symbolic terms using a number of custom rewrite rules, striving to derive a closed-form expression in a best-effort fashion. Fig. 1.5 illustrates simplification of an expression based on an input-output example. PSI also supports modular inference by analyzing one function at a time, deriving a symbolic distribution transformer that can be instantiated at call sites before further simplification. We evaluate PSI on a set of example applications of existing systems for approximate probabilistic inference and demonstrate that it is surprisingly effective on many practically interesting examples (while still struggling on others). We present PSI in Chapter 4.

λPSI: EXACT INFERENCE FOR HIGHER-ORDER PROBABILISTIC PROGRAMS The input language of our original implementation of PSI is rather limited: all variables store real numbers, and arrays are unrolled syntactically (in particular, PSI does not

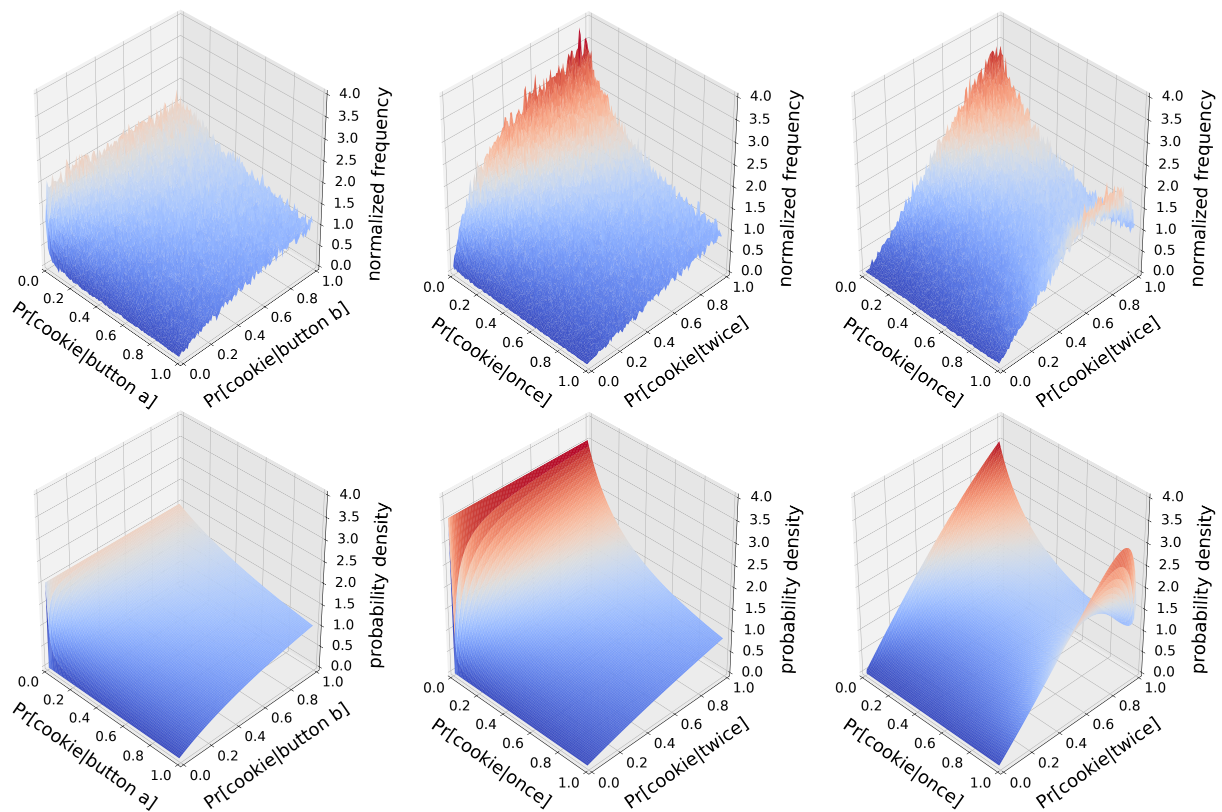


Figure 1.6: Rejection sampling (top) vs. symbolic inference in λPSI (bottom).

allow arrays of random length, nor random indices to index arrays). Many popular probabilistic programming languages support other types of values, including functions, tuples and arrays. In λPSI, we address all of those limitations of PSI. Furthermore, many approximate inference approaches allow for *nested inference* (often in an ad-hoc fashion): the model on which we want to perform probabilistic inference may itself perform (nested) probabilistic inference queries. λPSI is a statically typed probabilistic programming language supporting first-class functions, distributions, arrays and tuples, as well as nested inference. λPSI supports nested inference by making probabilistic inference a first-class language feature. The language includes some support for dependent types, including for the length of arrays. In comparison to PSI, we have also improved the symbolic simplification backend. E.g., we have added additional features to the intermediate symbolic language, representing first-class functions, tuples and arrays. We have also vastly improved the symbolic summation engine by modeling it after symbolic integration in PSI. Fig. 1.6 compares rejection sampling results with symbolic inference on three nested inference queries. Note that here, λPSI is not only more precise, but also significantly more performant: λPSI computed the results in a couple of seconds each, while rejection sampling took above 10 minutes. We present λPSI in Chapter 5.

1.3.3 *Conclusion and Future Work*

In Chapter 6, we finish the thesis with concluding remarks and present many promising items for future work.

Part I

ABSTRACT INTERPRETATION OF NEURAL NETWORKS

2

AI^2: SAFETY AND ROBUSTNESS CERTIFICATION OF NEURAL NETWORKS WITH ABSTRACT INTERPRETATION

Neural networks have become an integral part of many safety-critical applications, including vehicle control [19], medical diagnosis [8], malware detection [173], and aircraft collision avoidance detection [88]. This is because these models have obtained near-human accuracy in some domains [88, 96].

Despite all success, a fundamental challenge remains: to ensure that machine learning systems, and deep neural networks in particular, behave as intended. This challenge has become critical in light of recent research [162], showing that even highly accurate neural networks are vulnerable to *adversarial attacks*, where the network can be tricked into making wrong predictions by only slightly modifying an input, such that the network misclassifies the perturbed input, even though the original input is correctly classified by the network [9, 25, 47, 59, 98, 127, 129, 161]. Such a misclassified input is called an *adversarial example*. Various kinds of perturbations have been shown to successfully generate adversarial examples (e.g., [24, 60, 69, 70, 79, 80, 117, 123, 131, 147, 163]).

Fig. 2.1 illustrates two attacks, FGSM and brightening, against a digit classifier. For each attack, Fig. 2.1 shows an input in the Original column, the perturbed input in the Perturbed column, and the pixels that were changed in the Diff column. Brightened pixels are marked in yellow and darkened pixels are marked in purple. The FGSM [60] attack perturbs an image by adding to it a particular noise vector multiplied by a small number ϵ (in Fig. 2.1, $\epsilon = 0.3$). The brightening attack (e.g., [131]) perturbs an image by changing all pixels above the threshold $1 - \delta$ to the brightest possible value (in Fig. 2.1, $\delta = 0.085$).

Adversarial examples can be especially problematic when safety-critical systems rely on neural networks. For instance, it has been shown that attacks can be executed physically (e.g., [46, 99]) and against neural networks accessible only as a black box (e.g., [60, 162, 166]).

As a result, there is considerable interest in ensuring robustness of neural networks against such attacks, such as to mitigate these issues.

A line of research to ensure safety of neural networks focuses on *adversarial training* where networks are trained against a model of adversarial attack. Gu and Rigazio [71] add concrete noise to the training set and remove it statistically for defending against adversarial examples. Many works have focused on designing

Attack	Original	Perturbed	Diff
FGSM [60], $\epsilon = 0.3$			
Brightening, $\delta = 0.085$			

Figure 2.1: Attacks applied to MNIST images [102].

defenses that increase robustness by using modified procedures for training the network (e.g., [60, 70, 106, 128, 165]). Goodfellow et al. [59] generate adversarial examples misclassified by neural networks and then design a defense against this attack by explicit training against perturbations generated by the attack.

Madry et al. [107] show that training against an optimal attack also guards against non-optimal attacks. While this technique was highly effective in experiments, Carlini et al. [26] demonstrated an attack for the safety-critical problem of ground-truthing, where this defense occasionally exacerbated the problem.

There has also been increased interest in formally verifying robustness of neural networks, in particular *local robustness*. Local robustness (or robustness, for short) requires that all samples in the neighborhood of a given input are classified with the same label [128]. The work of Kolter and Wong [91] demonstrates a defense against adversarial attacks, providing certificates which prove that no training example could be adversarially permuted, as well as bounds on the capability of an adversary to influence performance on a test set. This method is based on linear duality and computes a convex overapproximation of the *adversarial polytope*, capturing the set of possible neural network outputs given the region of possible inputs. However, the approach incurs significant accuracy and scalability overheads. Dvijotham et al. [44] (concurrent to our work) proposes an improved dual approach to overapproximation, using a more accurate and more tractable formulation based on weak duality. A weakness of duality-based approaches is that they rely non-trivially on the commutativity and associativity of addition and may therefore not be sound in the face of floating-point roundoff error.

Raghunatan et al. [135] and Katz et al. [87] verify the robustness of small neural networks. Others have developed approaches that can show non-robustness by underapproximating neural network behaviors [14] or methods that decide robustness of small fully connected feed-forward neural networks [88]. However, no sound analyzer developed in prior work handles convolutional networks, one of the most popular architectures.

KEY CHALLENGE: SCALABILITY AND PRECISION The main challenge facing sound analysis of neural networks is scaling to large classifiers while maintaining a

precision that suffices to prove useful properties. The analyzer must consider all possible outputs of the network over a prohibitively large set of inputs, processed by a vast number of intermediate neurons. For instance, consider the image of the digit 8 in Fig. 2.1 and suppose we would like to prove that no matter how much we further brighten pixels with intensity above $1 - 0.085$, the network will still classify the image as 8 (in this example we have 84 such pixels, shown in yellow). Assuming 64-bit floating point numbers are used to express pixel intensity, we obtain more than 10^{1154} possible perturbed images. Thus, proving the property by running a network exhaustively on all possible input images and checking if all of them are classified as 8 is infeasible. To avoid this state space explosion, current methods (e.g., [80, 88, 134]) symbolically encode the network as a logical formula and then check robustness properties with a constraint solver. However, such solutions do not scale to larger (e.g., convolutional) networks, which usually involve many intermediate computations.

KEY CONCEPT: ABSTRACT INTERPRETATION FOR AI The key insight of our work is to address the above challenge by leveraging the classic framework of abstract interpretation (e.g., [33, 34]), a theory of sound, computable, and precise finite approximations of potentially infinite sets of behaviors. Concretely, we leverage numerical abstract domains – a particularly good match, as AI systems tend to heavily manipulate numerical quantities. By showing how to apply abstract interpretation to reason about AI safety, we enable users to leverage decades of research and any future advancements in that area (e.g., in numerical domains [152]). With abstract interpretation, a neural network computation is overapproximated using an *abstract domain*. An abstract domain consists of logical formulas that capture certain shapes (e.g., zonotopes, a restricted form of polyhedra). For example, in Fig. 2.2, the green zonotope A_1 overapproximates the set of blue points (each point represents an image). Of course, sometimes, due to abstraction, a shape may also contain points that will not occur in any concrete execution (e.g., the red points in A_2).

THE AI^2 ANALYZER Based on this insight, we developed a system called AI^2 (*Abstract Interpretation for Artificial Intelligence*)[1]. AI^2 is the first scalable analyzer that handles common network layer types, including fully connected and convolutional layers with rectified linear unit activations (ReLU) and max pooling layers.

To illustrate the operation of AI^2, consider the example in Fig. 2.2, where we have a neural network, an image of the digit 8 and a set of perturbations: brightening with parameter 0.085. Our goal is to prove that the neural network classifies all perturbed images as 8. AI^2 takes the image of the digit 8 and the perturbation type and creates an abstract element A_1 that captures all perturbed images. In particular, we can capture the entire set of brightening perturbations exactly with a single zonotope. However, in general, this step may result in an abstract element that contains additional inputs (that is, red points). In the second step, A_1 is automatically

1 AI^2 is available at: `http://ai2.ethz.ch`

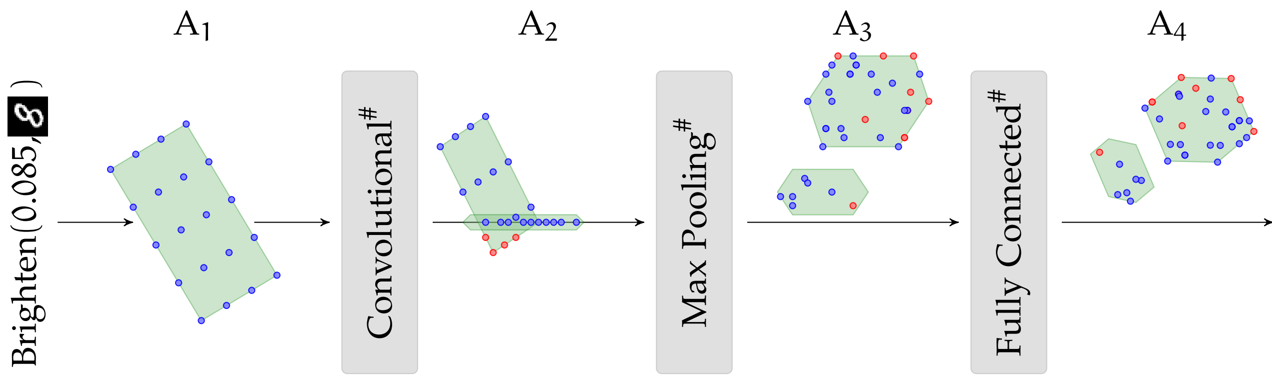


Figure 2.2: A high-level illustration of how AI^2 checks that all perturbed inputs are classified the same way. AI^2 first creates an abstract element A_1 capturing all perturbed images. (Here, we use a 2-bounded set of zonotopes.) It then propagates A_1 through the abstract transformer of each layer, obtaining new shapes. Finally, it verifies that all points in A_4 correspond to outputs with the same classification.

propagated through the layers of the network. Since layers work on concrete values and not abstract elements, this propagation requires us to define *abstract layers* (marked with $^\#$) that compute the effects of the layers on abstract elements. The abstract layers are commonly called the *abstract transformers* of the layers. Defining sound and precise, yet scalable abstract transformers is key to the success of an analysis based on abstract interpretation. We define abstract transformers for all three layer types shown in Fig. 2.2.

At the end of the analysis, the abstract output A_4 is an overapproximation of *all* possible concrete outputs. This enables AI^2 to verify safety properties such as robustness (e.g., are all images classified as 8?) directly on A_4. In fact, with a single abstract run, AI^2 was able to prove that a convolutional neural network classifies all of the considered perturbed images as 8.

We evaluated AI^2 on important tasks such as verifying robustness and comparing neural networks defenses. For example, for the perturbed image of the digit 0 in Fig. 2.1, we showed that while a non-defended neural network classified the FGSM perturbation with $\epsilon = 0.3$ as 9, this attack is *provably* eliminated when using a neural network trained with the defense of [106]. In fact, AI^2 proved that the FGSM attack is unable to generate adversarial examples from this image for any ϵ between 0 and 0.3.

MAIN CONTRIBUTIONS Our main contributions are:

- A sound and scalable method for analysis of deep neural networks based on abstract interpretation (§2.3).
- AI^2, an end-to-end analyzer, extensively evaluated on feed-forward and convolutional networks (computing with 53 000 neurons), far exceeding capabilities of current systems (§4.4).

- An application of AI^2 to evaluate *provable robustness* of neural network defenses (§2.6).

AI^2 is generic, i.e., it can be used with any abstract domain that provides a given number of standard operations. It therefore answers the question of how to apply 40 years of research in abstract interpreation to neural networks. The work presented in this chapter has been published in Gehr et al. [51].

2.1 NEURAL NETWORKS AND CONDITIONAL AFFINE TRANSFORMATIONS

In this section, we provide background on feedforward and convolutional neural networks and show how to transform them into a representation amenable to abstract interpretation. This representation helps us simplify the construction and description of our analyzer, which we discuss in later sections. We use the following notation: for a vector $\overline{x} \in \mathbb{R}^n$, x_i denotes its ith entry, and for a matrix $W \in \mathbb{R}^{n \times m}$, W_i denotes its ith row and $W_{i,j}$ denotes the entry in its ith row and j^{th} column.

CAT FUNCTIONS We express the neural network as a composition of *conditional affine transformations* (CAT), which are affine transformations guarded by logical constraints. The class of CAT functions, shown in Fig. 2.3, consists of functions $f\colon \mathbb{R}^m \to \mathbb{R}^n$ for $m, n \in \mathbb{N}$ and is defined recursively. Any affine transformation $f(\overline{x}) = W \cdot \overline{x} + \overline{b}$ is a CAT function, for a matrix W and a vector $\overline{b}$. Given sequences of conditions $E_1, \ldots, E_k$ and CAT functions $f_1, \ldots, f_k$, we write

$$f(\overline{x}) = \textbf{case } E_1\colon f_1(\overline{x}), \ldots, \textbf{case } E_k\colon f_k(\overline{x}).$$

This is also a CAT function, which returns $f_i(\overline{x})$ for the first E_i satisfied by $\overline{x}$. The conditions are conjunctions of constraints of the form $x_i \geqslant x_j$, $x_i \geqslant 0$ and $x_i < 0$. Finally, any composition of CAT functions is a CAT function. We often write $f'' \circ f'$ to denote the CAT function $f(\overline{x}) = f''(f'(\overline{x}))$.

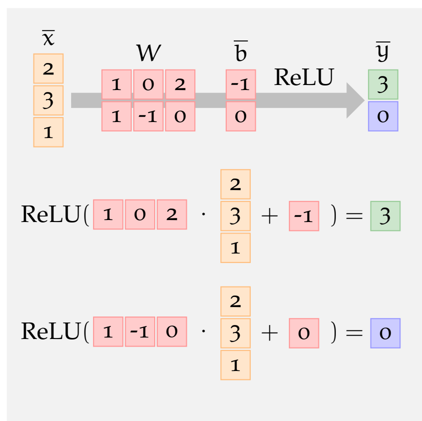
LAYERS Neural networks are often organized as a sequence of layers, such that the output of one layer is the input of the next layer. Layers consist of *neurons*, performing the same function but with different parameters. The output of a layer is formed by stacking the outputs of the neurons into a vector or three-dimensional array. We will define the functionality in terms of entire layers instead of in terms of individual neurons.

RESHAPING OF INPUTS Layers often take three-dimensional inputs (e.g., colored images). Such inputs are transformed into vectors by reshaping. A three-dimensional array $\overline{x} \in \mathbb{R}^{m \times n \times r}$ can be reshaped to $\overline{x}^v \in \mathbb{R}^{m \cdot n \cdot r}$ in a canonical way, first by depth, then by column, finally by row. That is, given $\overline{x}$, we have

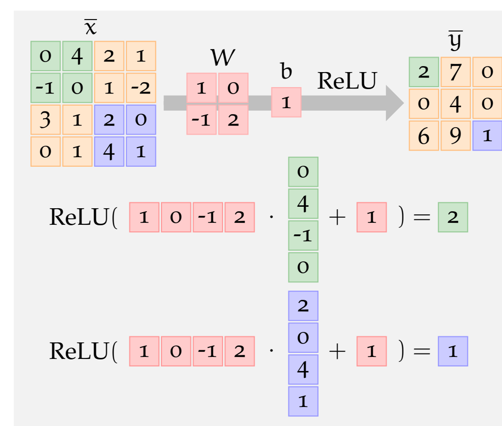
$$\overline{x}^v = (x_{1,1,1} \ldots x_{1,1,r}\ x_{1,2,1} \ldots x_{1,2,r} \ldots x_{m,n,1} \ldots x_{m,n,r})^T.$$

$$
\begin{aligned}
f(\overline{x}) \quad &::= \quad W \cdot \overline{x} + \overline{b} \\
&\;\;\,| \quad \textbf{case } E_1 \colon f_1(\overline{x}), \ldots, \textbf{case } E_k \colon f_k(\overline{x}) \\
&\;\;\,| \quad f(f'(\overline{x})) \\
E \quad &::= \quad E \wedge E \mid x_i \geqslant x_j \mid x_i \geqslant 0 \mid x_i < 0
\end{aligned}
$$

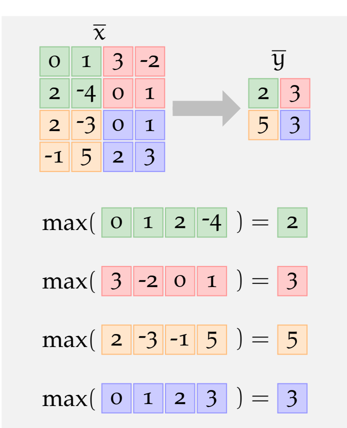
Figure 2.3: Definition of CAT functions.



(a) Fully connected layer $FC_{W,\overline{b}}$



(b) Convolutional layer $Conv_{(W,b)}$ (one filter)



(c) Max pooling layer $MaxPool_{2,2}$

Figure 2.4: One example computation for each of the three layer types supported by AI^2.

ACTIVATION FUNCTION Typically, layers in a neural network perform a linear transformation followed by a non-linear activation function. We focus on the commonly used rectified linear unit (ReLU) activation function, which for $x \in \mathbb{R}$ is defined as $\text{ReLU}(x) = \max(0, x)$, and for a vector $\overline{x} \in \mathbb{R}^n$ as $\text{ReLU}(\overline{x}) = (\text{ReLU}(x_1), \ldots, \text{ReLU}(x_n))$.

RELU TO CAT We can express the ReLU activation function as $\text{ReLU} = \text{ReLU}_n \circ \ldots \circ \text{ReLU}_1$ where ReLU_i processes the ith entry of the input $\overline{x}$ and is given by

$$
\begin{aligned}
\text{ReLU}_i(\overline{x}) = \; &\textbf{case } (x_i \geqslant 0) \colon \overline{x}, \\
&\textbf{case } (x_i < 0) \colon I_{i \leftarrow 0} \cdot \overline{x}.
\end{aligned}
$$

$I_{i \leftarrow 0}$ is the identity matrix with the ith row replaced by zeros.

FULLY CONNECTED (FC) LAYER An FC layer takes a vector of size m (the m outputs of the previous layer), and passes it to n neurons, each computing a function based on the neuron's *weights* and *bias*, one weight for each component of the input. Formally, an FC layer with n neurons is a function $FC_{W,\overline{b}} \colon \mathbb{R}^m \to \mathbb{R}^n$ parameterized by a weight matrix $W \in \mathbb{R}^{n \times m}$ and a bias $\overline{b} \in \mathbb{R}^n$. For $\overline{x} \in \mathbb{R}^m$, we have

$$FC_{W,\overline{b}}(\overline{x}) = \text{ReLU}(W \cdot \overline{x} + \overline{b}).$$

Fig. 2.4a shows an FC layer computation for $\overline{x} = (2, 3, 1)$.

CONVOLUTIONAL LAYER A convolutional layer is defined by a series of t filters $F^{p,q} = (F_1^{p,q}, .., F_t^{p,q})$, parameterized by the same p and q, where $p \leqslant m$ and $q \leqslant n$. A filter $F_i^{p,q}$ is a function parameterized by a three-dimensional array of weights $W \in \mathbb{R}^{p \times q \times r}$ and a bias $b \in \mathbb{R}$. A filter takes a three-dimensional array and returns a two-dimensional array:

$$F_i^{p,q} \colon \mathbb{R}^{m \times n \times r} \to \mathbb{R}^{(m-p+1) \times (n-q+1)}.$$

The entries of the output $\overline{y}$ for a given input $\overline{x}$ are given by

$$y_{i,j} = \text{ReLU}\left(\sum_{i'=1}^{p} \sum_{j'=1}^{q} \sum_{k'=1}^{r} W_{i',j',k'} \cdot x_{(i+i'-1),(j+j'-1),k'} + b\right).$$

Intuitively, this matrix is computed by sliding the filter along the height and width of the input three-dimensional array, each time reading a slice of size $p \times q \times r$, computing its dot product with W (resulting in a real number), adding b, and applying ReLU. The function Conv_F, corresponding to a convolutional layer with t filters, has the following type:

$$\text{Conv}_F \colon \mathbb{R}^{m \times n \times r} \to \mathbb{R}^{(m-p+1) \times (n-q+1) \times t}.$$

As expected, the function Conv_F returns a three-dimensional array of depth t, which stacks the outputs produced by each filter. Fig. 2.4b illustrates a computation of a convolutional layer with a single filter. For example:

$$y_{1,1,1} = \text{ReLU}((1 \cdot 0 + 0 \cdot 4 + (-1) \cdot (-1) + 2 \cdot 0) + 1) = 2.$$

Here, the input is a three-dimensional array in $\mathbb{R}^{4 \times 4 \times 1}$. As the input depth is 1, the depth of the filter's weights is also 1. The output depth is 1 because the layer has one filter.

CONVOLUTIONAL LAYER TO CAT For a convolutional layer $Conv_F$, we define a matrix W^F whose entries are those of the weight matrices for each filter (replicated to simulate sliding), and a bias $\overline{b}^F$ consisting of copies of the filters' biases. We then treat the convolutional layer Conv^F like the equivalent $\text{FC}_{W^F, \overline{b}^F}$.

We will provide formal definitions of W^F and $\overline{b}^F$ below. First, we provide an intuitive illustration of the translation on the example in Fig. 2.4b. Consider the first entry $y_{1,1} = 2$ of $\overline{y}$ in Fig. 2.4b:

$$y_{1,1} = \text{ReLU}(W_{1,1} \cdot x_{1,1} + W_{1,2} \cdot x_{1,2} + W_{2,1} \cdot x_{2,1} + W_{2,2} \cdot x_{2,2} + b).$$

When $\overline{x}$ is reshaped to a vector $\overline{x}^v$, the four entries $x_{1,1}, x_{1,2}, x_{2,1}$ and $x_{2,2}$ will be found in x_1^v, x_2^v, x_5^v and x_6^v, respectively. Similarly, when $\overline{y}$ is reshaped to $\overline{y}^v$, the entry

$y_{1,1}$ will be found in y_1^v. Thus, to obtain $y_1^v = y_{1,1}$, we define the first row in W^F such that its 1st, 2nd, 5th, and 6th entries are $W_{1,1}$, $W_{1,2}$, $W_{2,1}$ and $W_{2,2}$. The other entries are zeros. We also define the first entry of the bias to be b. For similar reasons, to obtain $y_2^v = y_{1,2}$, we define the second row in W^F such that its 2nd, 3rd, 6th, and 7th entries are $W_{1,1}$, $W_{1,2}$, $W_{2,1}$ and $W_{2,2}$ (also $b_2 = b$). By following this transformation, we obtain the matrix $W^F \in \mathbb{R}^9 \times \mathbb{R}^{16}$ and the bias $\overline{b}^F \in \mathbb{R}^9$:

$$W^F = \begin{pmatrix}
\mathbf{1} & \mathbf{0} & 0 & 0 & \mathbf{-1} & \mathbf{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & \mathbf{1} & \mathbf{0} & 0 & 0 & \mathbf{-1} & \mathbf{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & \mathbf{1} & \mathbf{0} & 0 & 0 & \mathbf{-1} & \mathbf{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{0} & 0 & 0 & \mathbf{-1} & \mathbf{2} & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{0} & 0 & 0 & \mathbf{-1} & \mathbf{2} & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{0} & 0 & 0 & \mathbf{-1} & \mathbf{2} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{0} & 0 & 0 & \mathbf{-1} & \mathbf{2} & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{0} & 0 & 0 & \mathbf{-1} & \mathbf{2} & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{0} & 0 & 0 & \mathbf{-1} & \mathbf{2}
\end{pmatrix} \qquad \overline{b}^F = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

To aid understanding, we show the entries from W that appear in the resulting matrix W^F in bold.

CONVOLUTIONAL LAYER DETAILS Formally, for filters $W^k \in \mathbb{R}^{p \times q \times r}$, $b^k \in \mathbb{R}$ for $1 \leqslant k \leqslant t$, we have

$$\begin{aligned}
&\mathrm{Conv}_F(\overline{x})\colon \mathbb{R}^{n \times m \times r} \to \mathbb{R}^{(m-p+1)\times(n-q+1)\times t} \\
&\mathrm{Conv}_F(\overline{x})_{i,j,k} = \mathrm{ReLU}(\sum_{i'=1}^{p} \sum_{j'=1}^{q} \sum_{k'=1}^{r} W^k_{i',j',k'} \cdot x_{(i+i'-1),(j+j'-1),k'} + b^k),
\end{aligned}$$

for $1 \leqslant i \leqslant m-p+1$, $1 \leqslant j \leqslant n-q+1$ and $1 \leqslant k \leqslant t$. Reshaping both the input and the output vector such that they have only one index, we obtain

$$\begin{aligned}
&\mathrm{Conv}'_F(\overline{x})\colon \mathbb{R}^{n \cdot m \cdot r} \to \mathbb{R}^{(m-p+1)\cdot(n-q+1)\cdot t} \\
&\mathrm{Conv}'_F(\overline{x})_{(n-q+1)\cdot t\cdot(i-1)+t\cdot(j-1)+k} \\
&\quad = \mathrm{ReLU}(\sum_{i'=1}^{p} \sum_{j'=1}^{q} \sum_{k'=1}^{r} W^k_{i',j',k'} \cdot x_{n\cdot r\cdot(i+i'-2)+r\cdot(j+j'-2)+k'} + b^k),
\end{aligned}$$

for $1 \leqslant i \leqslant m-p+1$, $1 \leqslant j \leqslant n-q+1$ and $1 \leqslant k \leqslant t$. The function Conv'_F is ReLU after an affine transformation, therefore there is a matrix $W^F \in \mathbb{R}^{((m-p+1)\cdot(n-q+1)\cdot t)\times(n\cdot m\cdot r)}$ and a vector $\overline{b}^F \in \mathbb{R}^{(m-p+1)\cdot(n-q+1)\cdot t}$ such that

$$\mathrm{Conv}_F(\overline{x})^v = \mathrm{Conv}'_F(\overline{x}^v) = \mathrm{ReLU}(W^F \cdot \overline{x}^v + b^F) = \mathrm{FC}_{W^F,\overline{b}^F}(\overline{x}).$$

The entries of W^F and $\overline{b}^F$ are obtained by equating

$$\begin{aligned}
&\mathrm{FC}(\overline{e}_l)_{(n-q+1)\cdot t\cdot(i-1)+t\cdot(j-1)+k} \\
&\qquad = \mathrm{ReLU}(W^F_{(n-q+1)\cdot t\cdot(i-1)+t\cdot(j-1)+k,l} + b^F_{(n-q+1)\cdot t\cdot(i-1)+t\cdot(j-1)+k}) \text{ with} \\
&\mathrm{Conv}'_F(\overline{e}_l)_{(n-q+1)\cdot t\cdot(i-1)+t\cdot(j-1)+k} \\
&\qquad = \mathrm{ReLU}(\sum_{i'=1}^{p}\sum_{j'=1}^{q}\sum_{k'=1}^{r} W^k_{i',j',k'} \cdot [l = n\cdot r\cdot(i+i'-2) + r\cdot(j+j'-2)+k'] + b^k),
\end{aligned}$$

for standard basis vectors $\overline{e}_l$ with $(\overline{e}_l)_i = [l = i]$ for $1 \leqslant l \leqslant n$ and $1 \leqslant i \leqslant n\cdot m\cdot r$. This way, we obtain

$$\begin{aligned}
&W^F_{(n-q+1)\cdot t\cdot(i-1)+t\cdot(j-1)+k,l} \\
&\qquad = \sum_{i'=1}^{p}\sum_{j'=1}^{q}\sum_{k'=1}^{r} W^k_{i',j',k'} \cdot [l = n\cdot r\cdot(i+i'-2) + r\cdot(j+j'-2)+k'] \text{ and} \\
&b^F_{(n-q+1)\cdot t\cdot(i-1)+t\cdot(j-1)+k} = b^k,
\end{aligned}$$

for $1 \leqslant i \leqslant m-p+1$, $1 \leqslant j \leqslant n-q+1$ and $1 \leqslant k \leqslant t$. Note that here, $[\varphi]$ is an Iverson bracket, which is equal to 1 if φ holds and equal to 0 otherwise.

MAX POOLING (MP) LAYER An MP layer takes a three-dimensional array $\overline{x} \in \mathbb{R}^{m\times n\times r}$ and reduces the height m of $\overline{x}$ by a factor of p and the width n of $\overline{x}$ by a factor of q (for p and q dividing m and n). Depth is kept as-is. Neurons take as input disjoint subrectangles of $\overline{x}$ of size $p \times q$ and return the maximal value in their subrectangle. Formally, the MP layer is a function $\mathrm{MaxPool}_{p,q} \colon \mathbb{R}^{m\times n\times r} \to \mathbb{R}^{\frac{m}{p}\times\frac{n}{q}\times r}$ that for an input $\overline{x}$ returns the three-dimensional array $\overline{y}$ given by:

$$y_{i,j,k} = \max(\{x_{i',j',k} \mid \begin{array}{l} p\cdot(i-1) < i' \leqslant p\cdot i \\ q\cdot(j-1) < j' \leqslant q\cdot j\}). \end{array}$$

Fig. 2.4c illustrates the max pooling computation for $p = 2$, $q = 2$ and $r = 1$. For example, here we have:

$$y_{1,1,1} = \max(\{x_{1,1,1}, x_{1,2,1}, x_{2,1,1}, x_{2,2,1}\}) = 2.$$

MAX POOLING TO CAT Let $\mathrm{MaxPool}'_{p,q} \colon \mathbb{R}^{m\cdot n\cdot r} \to \mathbb{R}^{\frac{m}{p}\cdot\frac{n}{q}\cdot r}$ be the function that is obtained from $\mathrm{MaxPool}_{p,q}$ by reshaping its input and output: $\mathrm{MaxPool}'_{p,q}(\overline{x}^v) = \mathrm{MaxPool}_{p,q}(\overline{x})^v$. To represent max pooling as a CAT function, we define a series of CAT functions whose composition is $\mathrm{MaxPool}'_{p,q}$:

$$\mathrm{MaxPool}'_{p,q} = f_{\frac{m}{p}\cdot\frac{n}{q}\cdot r} \circ \ldots \circ f_1 \circ f^{MP}.$$

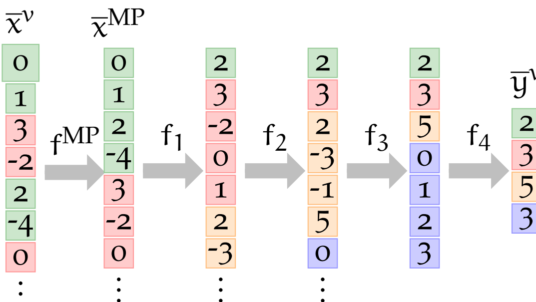


Figure 2.5: The operation of the transformed max pooling layer.

The first function is $f^{MP}(\overline{x}^\nu) = W^{MP} \cdot \overline{x}^\nu$, which reorders its input vector $\overline{x}^\nu$ to a vector $\overline{x}^{MP}$ in which the values of each max pooling subrectangle of $\overline{x}$ are adjacent. The remaining functions execute standard max pooling. Concretely, the function $f_i \in \{f_1, \ldots, f_{\frac{m}{p} \cdot \frac{n}{q} \cdot r}\}$ executes max pooling on the ith subrectangle by selecting the maximal value and removing the other values. We provide formal definitions of the CAT functions f^{MP} and f_i further below.

Here, we illustrate them on the example from Fig. 2.4c, where $r = 1$. The CAT computation for this example is shown in Fig. 2.5. The computation begins from the input vector $\overline{x}^\nu$, which is the reshaping of $\overline{x}$ from Fig. 2.4c. The values of the first 2×2 subrectangle in $\overline{x}$ (namely, 0, 1, 2 and -4) are separated in $\overline{x}^\nu$ by values from another subrectangle (3 and -2). To make them contiguous, we reorder $\overline{x}^\nu$ using a permutation matrix W^{MP}, yielding $\overline{x}^{MP}$. In our example, W^{MP} is given by

$$W^{MP} = \begin{pmatrix}
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0\\
0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0\\
0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1
\end{pmatrix}$$

One entry in each row of W^{MP} is 1, all other entries are zeros. If row i has entry j set to 1, then the jth value of $\overline{x}^\nu$ is moved to the ith entry of $\overline{x}^{MP}$. For example, we placed a one in the fifth column of the third row of W^{MP} to move the value x_5^ν to entry 3 of the output vector.

Next, for each $i \in \{1, \ldots, \frac{m}{p} \cdot \frac{n}{q}\}$, the function f_i takes as input a vector whose values at the indices between i and $i + p \cdot q - 1$ are those of the ith subrectangle of $\overline{x}$ in Fig. 2.4c. It then replaces those $p \cdot q$ values by their maximum:

$$f_i(\overline{x}) = (x_1, \ldots, x_{i-1}, x_k, x_{i+p \cdot q}, \ldots, x_{m \cdot n - (p \cdot q - 1) \cdot (i-1)}),$$

where the index $k \in \{i, \ldots, i + p \cdot q - 1\}$ is such that x_k is maximal. For k given, f_i can be written as a CAT function: $f_i(\overline{x}) = W^{(i,k)} \cdot \overline{x}$, where the rows of the matrix

$W^{(i,k)} \in \mathbb{R}^{(m \cdot n-(p \cdot q-1) \cdot i) \times (m \cdot n-(p \cdot q-1) \cdot (i-1))}$ are given by the following sequence of standard basis vectors:

$$\overline{e}_1, \ldots, \overline{e}_{i-1}, \overline{e}_k, \overline{e}_{i+p \cdot q}, \ldots, \overline{e}_{m \cdot n-(p \cdot q-1) \cdot (i-1)}.$$

For example, in Fig. 2.5, $f_1(\overline{x}^{MP}) = W^{(1,3)} \cdot \overline{x}^{MP}$ deletes 0, 1 and -4. Then it moves the value 2 to the first component, and the values at indices $5, \ldots, 16$ to components $2, \ldots, 13$. Overall, $W^{(1,3)}$ is given by:

$$W^{(1,3)} = \begin{pmatrix}
0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1
\end{pmatrix}$$

As, in general, k is not known in advance, we need to write f_i as a CAT function with a different case for each possible index k of the maximal value in $\overline{x}$. For example, in Fig. 2.5:

$$f_1(\overline{x}) = \begin{array}{ll}
\textbf{case } (x_1 \geqslant x_2) \wedge (x_1 \geqslant x_3) \wedge (x_1 \geqslant x_4): & W^{(1,1)} \cdot \overline{x}, \\
\textbf{case } (x_2 \geqslant x_1) \wedge (x_2 \geqslant x_3) \wedge (x_2 \geqslant x_4): & W^{(1,2)} \cdot \overline{x}, \\
\textbf{case } (x_3 \geqslant x_1) \wedge (x_3 \geqslant x_2) \wedge (x_3 \geqslant x_4): & W^{(1,3)} \cdot \overline{x}, \\
\textbf{case } (x_4 \geqslant x_1) \wedge (x_4 \geqslant x_2) \wedge (x_4 \geqslant x_3): & W^{(1,4)} \cdot \overline{x}.
\end{array}$$

In our example, the vector $\overline{x}^{MP}$ in Fig. 2.5 satisfies the third condition, and therefore $f_1(\overline{x}^{MP}) = W^{(1,3)} \cdot \overline{x}^{MP}$. Taking into account all four subrectangles, we obtain:

$$\text{MaxPool}'_{2,2} = f_4 \circ f_3 \circ f_2 \circ f_1 \circ f^{MP}.$$

In summary, each function f_i replaces $p \cdot q$ components of their input by the maximum value among them, suitably moving other values. For $\overline{x}^{\nu}$ in Fig. 2.5:

$$\text{MaxPool}'_{2,2}(\overline{x}^{\nu}) = W^{(4,7)} \cdot W^{(3,6)} \cdot W^{(2,2)} \cdot W^{(1,3)} \cdot W^{MP} \cdot \overline{x}^{\nu}.$$

MAX POOLING LAYER DETAILS $\text{MaxPool}_{p,q} \colon \mathbb{R}^{m \times n \times r} \to \mathbb{R}^{\frac{m}{p} \times \frac{n}{q} \times r}$ partitions the input vector into disjoint blocks of size $p \times q \times 1$ and replaces each block by its maximum value. Furthermore, $\text{MaxPool}'_{p,q} \colon \mathbb{R}^{m \cdot n \cdot r} \to \mathbb{R}^{\frac{m}{p} \cdot \frac{n}{q} \cdot r}$ is obtained from $\text{MaxPool}_{p,q}$ by reshaping both the input and output: $\text{MaxPool}'_{p,q}(\overline{x}^{\nu}) = \text{MaxPool}_{p,q}(\overline{x})^{\nu}$. We will represent $\text{MaxPool}'_{p,q}$ as a composition of CAT functions,

$$\text{MaxPool}'_{p,q} = f_{\frac{m}{p} \cdot \frac{n}{q} \cdot r} \circ \ldots \circ f_1 \circ f^{MP}.$$

Here, f^{MP} rearranges the input vector such that values from the same block are adjacent. Values from different blocks are brought into the same order as the output from each block appears in the output vector.

Note that $((i-1) \bmod p)+1, (j-1) \bmod q)+1, 1)$ are the indices of input value $x_{i,j,k}$ within its respective block and $\left(\left\lfloor \frac{i-1}{p} \right\rfloor + 1, \left\lfloor \frac{j-1}{q} \right\rfloor + 1, k\right)$ are the indices of the unique value in the output vector whose value depends on $x_{i,j,k}$. Recall that the permutation matrix M representing a permutation π is given by $M_{\pi(i)} = \overline{e}_i$.

The CAT function f^{MP} is a linear transformation $f^{MP}(\overline{x}^v) = W^{MP} \cdot \overline{x}^v$ where the permutation matrix W^{MP} is given by

$$W^{MP}_{r \cdot p \cdot q \cdot \left(\frac{n}{q}\left\lfloor \frac{i-1}{p} \right\rfloor + \left\lfloor \frac{j-1}{q} \right\rfloor\right) + p \cdot q \cdot (k-1) + q \cdot ((i-1) \bmod p) + ((j-1) \bmod q) + 1} = \overline{e}_{n \cdot r \cdot (i-1) + r \cdot (j-1) + k},$$

for $1 \leqslant i \leqslant m$, $1 \leqslant j \leqslant n$ and $1 \leqslant k \leqslant r$.

For each $1 \leqslant i \leqslant \frac{m}{p} \cdot \frac{n}{p} \cdot r$, the CAT function f_i selects the maximum value from a $(p \cdot q)$-segment starting from the ith component of the input vector. The function f_i consists of a sequence of cases, one for each of the $p \cdot q$ possible indices of the maximal value in the segment:

$$\begin{aligned} f_i(\overline{x}) = \textbf{case } & (x_i \geqslant x_{i+1}) \wedge \ldots \wedge (x_i \geqslant x_{i+p \cdot q-1}) \colon W^{(i,i)} \cdot \overline{x}, \\ \textbf{case } & (x_{i+1} \geqslant x_i) \wedge \ldots \wedge (x_{i+1} \geqslant x_{i+p \cdot q-1}) \colon W^{(i,i+1)} \cdot \overline{x}, \\ & \vdots \\ \textbf{case } & (x_{i+p \cdot q-1} \geqslant x_i) \wedge \ldots \wedge (x_{i+p \cdot q-1} \geqslant x_{i+p \cdot q-2}) \colon W^{(i,i+p \cdot q-1)} \cdot \overline{x}. \end{aligned}$$

We use the matrix $W^{(i,k)} \in \mathbb{R}^{(m \cdot n \cdot r-(p \cdot q-1) \cdot i) \times (m \cdot n \cdot r-(p \cdot q-1) \cdot (i-1))}$ to replace the segment $x_i, \ldots, x_{i+p \cdot q-1}$ of the input vector $\overline{x}$ by the value x_k. $W^{(i,k)}$ is given by

$$W^{(i,k)}_j = \begin{cases} \overline{e}_j, & \text{if } 1 \leqslant j \leqslant i-1 \\ \overline{e}_k, & \text{if } j = i \\ \overline{e}_{j+p \cdot q-1}, & \text{if } i+1 \leqslant j \leqslant m \cdot n \cdot r-(p \cdot q-1) \cdot i \end{cases}.$$

NETWORK ARCHITECTURES Two popular architectures of neural networks are fully connected feedforward (FNN) and convolutional (CNN). An FNN is a sequence of fully connected layers, while a CNN [82] consists of all previously described layer types: convolutional, max pooling, and fully connected.

2.2 BACKGROUND: ABSTRACT INTERPRETATION

We now provide a short introduction to Abstract Interpretation (AI). AI enables one to prove program properties on a set of inputs *without* actually running the program. Formally, given a function $f \colon \mathbb{R}^m \to \mathbb{R}^n$, a set of inputs $X \subseteq \mathbb{R}^m$, and

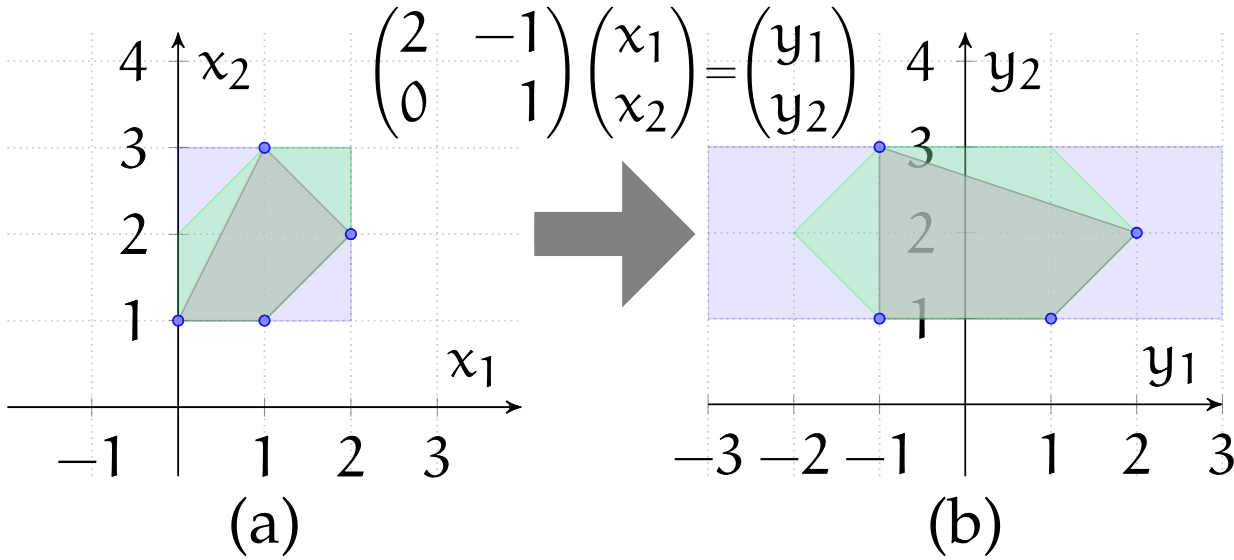


Figure 2.6: (a) Abstracting four points with a polyhedron (gray), zonotope (green), and box (blue). (b) The points and abstractions resulting from the affine transformer.

a property $C \subseteq \mathbb{R}^n$, the goal is to determine whether the property holds, that is, whether $\forall \bar{x} \in X.\, f(\bar{x}) \in C$.

Fig. 2.6 shows a CAT function $f\colon \mathbb{R}^2 \to \mathbb{R}^2$ that is defined as $f(\bar{x}) = \left(\begin{smallmatrix} 2 & -1 \\ 0 & 1 \end{smallmatrix}\right) \cdot \bar{x}$ and four input points for the function f, given as $X = \{(0,1),(1,1),(1,3),(2,2)\}$. Let the property be $C = \{(y_1, y_2) \in \mathbb{R}^2 \mid y_1 \geqslant -2\}$, which holds in this example. To reason about all inputs simultaneously, we lift the definition of f to be over a set of inputs X rather than a single input:

$$T_f\colon \mathcal{P}(\mathbb{R}^m) \to \mathcal{P}(\mathbb{R}^n), \qquad T_f(X) = \{f(\bar{x}) \mid \bar{x} \in X\}.$$

The function T_f is called the *concrete transformer* of f. With T_f, our goal is to determine whether $T_f(X) \subseteq C$ for a given input set X. Because the set X can be very large (or infinite), we cannot enumerate all points in X to compute $T_f(X)$. Instead, AI overapproximates sets with abstract elements (drawn from some abstract domain $\mathcal{A}$) and then defines a function, called an *abstract transformer* of f, which takes an abstract element and overapproximates the effect of T_f. Then, the property C can be checked on the resulting abstract element returned by the abstract transformer. Naturally, because AI employs overapproximation, it is sound, but may be imprecise (i.e., may fail to prove the property when it holds). Next, we explain the ingredients of AI in more detail.

ABSTRACT DOMAINS Abstract domains consist of shapes expressible as a set of logical constraints. A few popular numerical abstract domains are: Box (i.e., Interval), Zonotope, and Polyhedra. These domains provide different precision versus scalability trade-offs (e.g., Box's abstract transformers are significantly faster than Polyhedra's abstract transformers, but polyhedra are significantly more precise than boxes). The Box domain consists of boxes, captured by a set of constraints

of the form $a \leqslant x_i \leqslant b$, for $a, b \in \mathbb{R} \cup \{-\infty, +\infty\}$ and $a \leqslant b$. A box B contains all points which satisfy all constraints in B. In our example, X can be abstracted by the following box:

$$B = \{0 \leqslant x_1 \leqslant 2, 1 \leqslant x_2 \leqslant 3\}.$$

Note that B is not very precise since it includes 9 integer points (along with other points), whereas X has only 4 points.

The Zonotope domain [56] consists of *zonotopes*. A zonotope is a center-symmetric convex closed polyhedron $Z \subseteq \mathbb{R}^n$ that can be represented as an affine function:

$$z\colon [a_1, b_1] \times [a_2, b_2] \times \cdots \times [a_m, b_m] \to \mathbb{R}^n.$$

In other words, z has the form $z(\overline{\epsilon}) = M \cdot \overline{\epsilon} + \overline{b}$ where $\overline{\epsilon}$ is a vector of error terms satisfying interval constraints $\overline{\epsilon}_i \in [a_i, b_i]$ for $1 \leqslant i \leqslant m$. The bias vector $\overline{b}$ captures the center of the zonotope, while the matrix M captures the boundaries of the zonotope around $\overline{b}$. A zonotope z represents all vectors in the image of z (i.e., $z[[a_1, b_1] \times \cdots \times [a_m, b_m]]$). In our example, X can be abstracted by the zonotope $z\colon [-1, 1]^3 \to \mathbb{R}^2$:

$$z(\epsilon_1, \epsilon_2, \epsilon_3) = (1 + 0.5 \cdot \epsilon_1 + 0.5 \cdot \epsilon_2, 2 + 0.5 \cdot \epsilon_1 + 0.5 \cdot \epsilon_3).$$

Zonotope is a more precise domain than Box: for our example, z includes only 7 integer points.

The Polyhedra [36] domain consists of convex closed polyhedra, where a polyhedron is captured by a set of linear constraints of the form $A \cdot \overline{x} \leqslant \overline{b}$, for some matrix A and a vector $\overline{b}$. A polyhedron C contains all points which satisfy the constraints in C. In our example, X can be abstracted by the following polyhedron:

$$C = \{x_2 \leqslant 2 \cdot x_1 + 1, x_2 \leqslant 4 - x_1, x_2 \geqslant 1, x_2 \geqslant x_1\}.$$

Polyhedra is a more precise domain than Zonotope: for our example, C includes only 5 integer points.

To conclude, abstract elements (from an abstract domain) represent large, potentially infinite sets. There are various abstract domains, providing different levels of precision and scalability.

ABSTRACT TRANSFORMERS To compute the effect of a function on an abstract element, AI uses the concept of an *abstract transformer*. The concrete transformer $T_f\colon \mathcal{P}(\mathbb{R}^m) \to \mathcal{P}(\mathbb{R}^n)$ of a function $f\colon \mathbb{R}^m \to \mathbb{R}^n$ maps each set $X \in \mathbb{R}^m$ to its image $f[X] \subseteq \mathbb{R}^n$. An abstract transformer of T_f is a function over abstract domains, denoted by $T_f^{\#}\colon \mathcal{A}^m \to \mathcal{A}^n$. The superscripts denote the number of components of the represented vectors. For example, elements in $\mathcal{A}^m$ represent sets of vectors of dimension m. This also determines which variables can appear in the constraints

associated with an abstract element. For example, elements in $\mathcal{A}^m$ constrain the values of the variables $x_1, \ldots, x_m$.

Abstract transformers have to be *sound*. To define soundness, we introduce two functions: the *abstraction* function α and the *concretization* function γ. Such an abstraction function $\alpha^m \colon \mathcal{P}(\mathbb{R}^m) \to \mathcal{A}^m$ maps a set of vectors to an abstract element in $\mathcal{A}^m$ that overapproximates it. For example, in the Box domain:

$$\alpha^2(\{(0,1),(1,1),(1,3),(2,2)\}) = \{0 \leqslant x_1 \leqslant 2, 1 \leqslant x_2 \leqslant 3\}.$$

A concretization function $\gamma^m \colon \mathcal{A}^m \to \mathcal{P}(\mathbb{R}^m)$ does the opposite: it maps an abstract element to the set of concrete vectors that it represents. For example, for Box:

$$\begin{aligned}\gamma^2(\{0 \leqslant x_1 \leqslant 2, 1 \leqslant x_2 \leqslant 3\}) = &\{(0,1),(0,2),(0,3),\\ &(1,1),(1,2),(1,3),\\ &(2,1),(2,2),(2,3),\ldots\}.\end{aligned}$$

This only shows the 9 vectors with integer components. We can now define soundness. An abstract transformer $T_f^\#$ is *sound* if for all $a \in \mathcal{A}^m$, we have $T_f(\gamma^m(a)) \subseteq \gamma^n(T_f^\#(a))$, where T_f is the concrete transformer. That is, an abstract transformer has to overapproximate the effect of a concrete transformer. For example, the transformers illustrated in Fig. 2.6 are sound. For instance, if we apply the Box transformer on the box in Fig. 2.6a, it will produce the box in Fig. 2.6b. The box in Fig. 2.6b includes all points that f could compute in principle when given any point included in the concretization of the box in Fig. 2.6a. Analogous properties hold for the Zonotope and Polyhedra transformers. It is also important that abstract transformers are *precise*. That is, the abstract output should include as few points as possible. For example, as we can see in Fig. 2.6b, the output produced by the Box transformer is less precise (it is larger) than the output produced by the Zonotope transformer, which in turn is less precise than the output produced by the Polyhedra transformer.

PROPERTY VERIFICATION After obtaining the (abstract) output, we can check various properties of interest on the result. In general, an abstract output $a = T_f^\#(\alpha^m(X))$ proves a property $T_f(X) \subseteq C$ if $\gamma^n(a) \subseteq C$. If the abstract output proves a property, we know that the property holds for all possible concrete values. However, the property may hold even if it cannot be proven with a given abstract domain. For example, in Fig. 2.6b, for all concrete points, the property $C = \{(y_1, y_2) \in \mathbb{R}^2 \mid y_1 \geqslant -2\}$ holds. However, with the Box domain, the abstract output violates C, which means that the Box domain is not precise enough to prove the property. In contrast, the Zonotope and Polyhedra domains are precise enough to prove the property.

In summary, to apply AI successfully, we need to: (a) find a suitable abstract domain, and (b) define abstract transformers that are sound and as precise as possible. In the next section, we introduce abstract transformers for neural networks that are parameterized by the numerical abstract domain. This means that we can

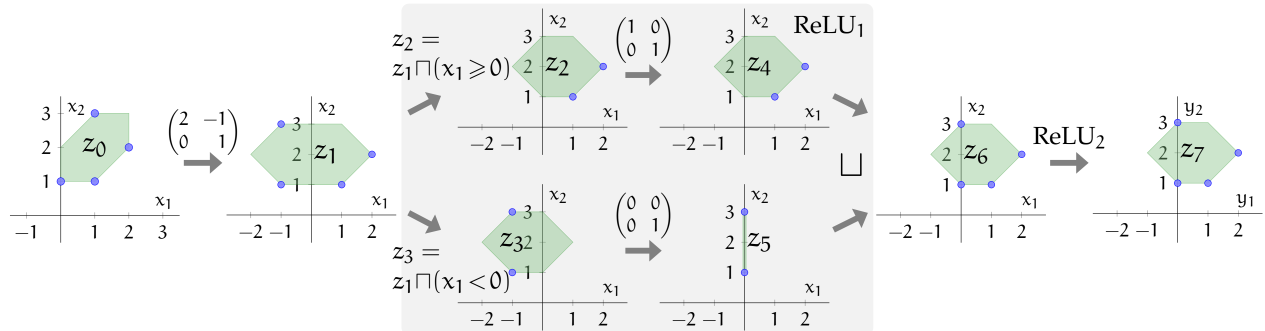


Figure 2.7: Illustration of how AI^2 overapproximates neural network states. Blue circles show the concrete values, while green zonotopes show the abstract elements. The gray box shows the steps in one application of the ReLU transformer (ReLU_1).

explore the precision-scalability trade-off by plugging in different abstract domains.

2.3 AI^2: AI FOR NEURAL NETWORKS

In this section we present AI^2, an abstract interpretation framework for sound analysis of neural networks. We begin by defining abstract transformers for the different kinds of neural network layers. Using these transformers, we then show how to prove robustness properties of neural networks.

2.3.1 *Abstract Interpretation for CAT Functions*

In this section, we show how to overapproximate CAT functions with AI. We illustrate the method on the example in Fig. 2.7, which shows a simple network that manipulates two-dimensional vectors using a single fully connected layer of the form

$$f(\overline{x}) = \text{ReLU}_2\left(\text{ReLU}_1\left(\left(\begin{smallmatrix}2 & -1\\ 0 & 1\end{smallmatrix}\right)\cdot\overline{x}\right)\right).$$

Recall that

$$\begin{aligned}\text{ReLU}_i(\overline{x}) \;=\; & \textbf{case}\ (x_i \geqslant 0)\colon \overline{x},\\ & \textbf{case}\ (x_i < 0)\colon I_{i\leftarrow 0}\cdot\overline{x},\end{aligned}$$

where $I_{i\leftarrow 0}$ is the identity matrix with the ith row replaced by the zero vector.

We overapproximate the network behavior on an abstract input. The input can be obtained directly (see §2.3.2) or by abstracting a set of concrete inputs to an abstract element (using the abstraction function α). For our example, we use the concrete inputs (the blue points) from Fig. 2.6. Those concrete inputs are abstracted to the green enclosing zonotope $z_0\colon [-1,1]^3 \to \mathbb{R}^2$, given by

$$z_0(\epsilon_1,\epsilon_2,\epsilon_3) = (1 + 0.5\cdot\epsilon_1 + 0.5\cdot\epsilon_2, 2 + 0.5\cdot\epsilon_1 + 0.5\cdot\epsilon_3).$$

Due to abstraction, more (spurious) points may be added. In this example, other than the blue points, the entire area of the zonotope is spurious. We then apply

abstract transformers to the abstract input. Note that, if a function f can be written as $f = f'' \circ f'$, the concrete transformer for f is $T_f = T_{f''} \circ T_{f'}$. Similarly, given abstract transformers $T^{\#}_{f'}$ and $T^{\#}_{f''}$, an abstract transformer for f is $T^{\#}_{f''} \circ T^{\#}_{f'}$. When a neural network $N = f'_\ell \circ \cdots \circ f'_1$ is a composition of multiple CAT functions f'_i of the shape either $f'_i(\overline{x}) = W \cdot \overline{x} + \overline{b}$ or $f_i(\overline{x}) = \textbf{case } E_1 \colon f_1(\overline{x}), \ldots, \textbf{case } E_k \colon f_k(\overline{x})$, we only have to define abstract transformers for these two kinds of functions. We then obtain the abstract transformer $T^{\#}_N = T^{\#}_{f'_\ell} \circ \cdots \circ T^{\#}_{f'_1}$.

ABSTRACTING AFFINE FUNCTIONS To abstract functions of the form

$$f(\overline{x}) = W \cdot \overline{x} + \overline{b},$$

we assume that the underlying abstract domain supports an operator Aff that overapproximates such functions. We note that for Zonotope and Polyhedra, this operation is supported and exact. Fig. 2.7 demonstrates Aff as the first step taken for overapproximating the effect of the fully connected layer. Here, the resulting zonotope $z_1 : [-1, 1]^3 \to \mathbb{R}^2$ is

$$\begin{aligned} z_1(\epsilon_1, \epsilon_2, \epsilon_3) &= (2 \cdot (1 + 0.5 \cdot \epsilon_1 + 0.5 \cdot \epsilon_2) - (2 + 0.5 \cdot \epsilon_1 + 0.5 \cdot \epsilon_3), 2 + 0.5 \cdot \epsilon_1 + 0.5 \cdot \epsilon_3) \\ &= (0.5 \cdot \epsilon_1 + \epsilon_2 - 0.5 \cdot \epsilon_3, 2 + 0.5 \cdot \epsilon_1 + 0.5 \cdot \epsilon_3). \end{aligned}$$

ABSTRACTING CASE FUNCTIONS To abstract functions of the form

$$f(\overline{x}) = \textbf{case } E_1 \colon f_1(\overline{x}), \ldots, \textbf{case } E_k \colon f_k(\overline{x}),$$

we first split the abstract element a into the different cases (each defined by one of the expressions E_i), resulting in k abstract elements $a_1, \ldots, a_k$. We then compute the result of $T^{\#}_{f_i}(a_i)$ for each a_i. Finally, we unify the results to a single abstract element. To split and unify, we assume two standard operators for abstract domains: (1) *meet* with a conjunction of linear constraints and (2) *join*. The meet ($\sqcap$) operator is an abstract transformer for set intersection: for an inequality expression E from Fig. 2.3, we have

$$\gamma^n(a) \cap \{x \in \mathbb{R}^n \mid x \models E\} \subseteq \gamma^n(a \sqcap E).$$

The *join* ($\sqcup$) operator is an abstract transformer for set union:

$$\gamma^n(a_1) \cup \gamma^n(a_2) \subseteq \gamma^n(a_1 \sqcup a_2).$$

We further assume that the abstract domain contains an element $\bot$, which satisfies

$$\gamma^n(\bot) = \{\},$$

$\bot \sqcap E = \bot$ and $a \sqcup \bot = a$ for all $a \in \mathcal{A}$.

$$\text{For } f(\overline{x}) = W \cdot \overline{x} + \overline{b},\ T_f^{\#}(a) = \text{Aff}(a, W, \overline{b}).$$

$$\text{For } f(\overline{x}) = \textbf{case } E_1 \colon f_1(\overline{x}), \ldots, \textbf{case } E_k \colon f_k(\overline{y}),$$

$$T_f^{\#}(a) = \bigsqcup_{1 \leqslant i \leqslant k} T_{f_i}^{\#}(a \sqcap E_i).$$

$$\text{For } f(\overline{x}) = f_2(f_1(\overline{x})),\ T_f^{\#}(a) = T_{f_2}^{\#}(T_{f_1}^{\#}(a)).$$

Figure 2.8: Abstract transformers for CAT functions.

For our example in Fig. 2.7, abstract interpretation continues on z_1 using the meet and join operators. To compute the effect of ReLU_1, we first split z_1 into two new zonotopes $z_2 = z_1 \sqcap (x_1 \geqslant 0)$ and $z_3 = z_1 \sqcap (x_1 < 0)$. One way to compute a meet between a zonotope and a linear constraint is to modify the intervals of the error terms (see [57]). In our example, the resulting zonotopes are $z_2 \colon [-1, 1] \times [0, 1] \times [-1, 1] \to \mathbb{R}^2$ such that $z_2(\overline{\epsilon}) = z_1(\overline{\epsilon})$ and $z_3 \colon [-1, 1] \times [-1, 0] \times [-1, 1] \to \mathbb{R}^2$ such that $z_3(\overline{\epsilon}) = z_1(\overline{\epsilon})$ for $\bar{\epsilon}$ common to their respective domains. Note that both z_2 and z_3 contain small spurious areas, because the intersections of the respective linear constraints with z_1 are not zonotopes. Therefore, they cannot be captured exactly by the domain. Here, the meet operator $\sqcap$ overapproximates set intersection $\cap$ to get a sound, but not perfectly precise, result.

Then, the two cases of ReLU_1 are processed separately. We apply the abstract transformer of $f_1(\overline{x}) = \overline{x}$ to z_2 and we apply the abstract transformer of $f_2(\overline{x}) = I_{0 \leftarrow 0} \cdot \overline{x}$ to z_3. The resulting zonotopes are $z_4 = z_2$ and $z_5 : [-1, 1]^2 \to \mathbb{R}^2$ such that $z_5(\epsilon_1, \epsilon_3) = (0, 2 + 0.5 \cdot \epsilon_1 + 0.5 \cdot \epsilon_3)$. These are then joined to obtain a single zonotope z_6. Since z_5 is contained in z_4, we get $z_6 = z_4$ (of course, this need not always be the case). Then, z_6 is passed to ReLU_2. Because $z_6 \sqcap (x_1 < 0) = \perp$, this results in $z_7 = z_6$. Finally, $\gamma^2(z_7)$ is our overapproximation of the network outputs for our initial set of points. The abstract element z_7 is a finite representation of this infinite set.

In summary, we define abstract transformers for every kind of CAT function (summarized in Fig. 2.8). These definitions are general and are compatible with any abstract domain $\mathcal{A}$ which has a minimum element $\perp$ and supports (1) a meet operator between an abstract element and a conjunction of linear constraints, (2) a join operator between two abstract elements, and (3) an affine transformer. We assume that the operations are sound. We note that these operations are standard or definable with standard operations. By definition of the abstract transformers, we get soundness:

Theorem 1. *For any CAT function* f *with transformer* $T_f \colon \mathcal{P}(\mathbb{R}^m) \to \mathcal{P}(\mathbb{R}^n)$ *and any abstract input* $a \in \mathcal{A}^m$,

$$T_f(\gamma^m(a)) \subseteq \gamma^n(T_f^{\#}(a)).$$

Theorem 1 is the key to sound neural network analysis with our abstract transformers, as we explain in the next section.

2.3.2 *Neural Network Analysis with AI*

In this section, we explain how to leverage AI with our abstract transformers to prove properties of neural networks. Below, we will focus on robustness properties. However, the framework can be applied to reason about any safety property.

For robustness, we aim to determine if for a (possibly unbounded) set of inputs, the outputs of a neural network satisfy a given condition. A robustness property for a neural network $N\colon \mathbb{R}^m \to \mathbb{R}^n$ is a pair $(X, C) \in \mathcal{P}(\mathbb{R}^m) \times \mathcal{P}(\mathbb{R}^n)$ consisting of a robustness region X and a robustness condition C. We say that the neural network N satisfies a robustness property (X, C) if $N(\overline{x}) \in C$ for all $\overline{x} \in X$.

LOCAL ROBUSTNESS This is a property (X, C_L) where X is a robustness region and C_L contains the outputs that describe the *same* label L:

$$C_L = \left\{ \overline{y} \in \mathbb{R}^n \;\middle|\; \underset{i \in \{1,\ldots,n\}}{\arg\max}(y_i) = L \right\}.$$

For example, Fig. 2.7 shows a neural network and a robustness property (X, C_2) for

$$X = \{(0,1), (1,1), (1,3), (2,2)\}$$

and

$$C_2 = \{\overline{y} \mid \arg\max(y_1, y_2) = 2\}.$$

In this example, (X, C_2) holds. Typically, we will want to check that there is *some* label L for which (X, C_L) holds.

We now explain how our abstract transformers can be used to prove a given robustness property (X, C).

ROBUSTNESS PROOFS USING AI Assume we are given a neural network $N\colon \mathbb{R}^m \to \mathbb{R}^n$, a robustness property (X, C) and an abstract domain $\mathcal{A}$ (supporting $\sqcup$, $\sqcap$ with a conjunction of linear constraints, Aff, and $\bot$) with an abstraction function α and a concretization function γ. Further assume that N can be written as a CAT function. Denote by $T^{\#}_N$ the abstract transformer of N, as defined in Fig. 2.8. Then, the following condition is sufficient to prove that N satisfies (X, C):

$$\gamma^n(T^{\#}_N(\alpha^m(X))) \subseteq C.$$

This follows from Theorem 1 and the properties of α and γ. Note that there may be abstract domains $\mathcal{A}$ that are not precise enough to prove that N satisfies (X, C), even if N in fact satisfies (X, C). On the other hand, if we are able to show that some abstract domain $\mathcal{A}$ proves that N satisfies (X, C), we know that it holds.

PROVING CONTAINMENT To prove the property (X, C) given the result $a = T^{\#}_{N}(\alpha^{m}(X))$ of abstract interpretation, we need to be able to show $\gamma^{n}(a) \subseteq C$. There is a general method if C is given by a CNF formula $\bigwedge_i \bigvee_j l_{i,j}$ where all literals $l_{i,j}$ are linear constraints: we show that the negated formula $\bigvee_i \bigwedge_j \neg l_{i,j}$ is inconsistent with the abstract element a by checking that $a \sqcap \left(\bigwedge_j \neg l_{i,j}\right) = \bot$ for all i.

For our example in Fig. 2.7, the goal is to check that all inputs are classified as 2. We can represent C using the formula $y_2 \geqslant y_1$. Its negation is $y_2 < y_1$, and it suffices to show that no point in the concretization of the abstract output satisfies this negated constraint. As indeed $z_7 \sqcap (y_2 < y_1) = \bot$, the property is successfully verified. However, note that we would not be able to prove some other true properties, such as $y_1 \geqslant 0$. This property holds for all concrete outputs, but some points in the concretization of the output z_7 do not satisfy it.

2.4 IMPLEMENTATION OF AI^2

The AI^2 framework is implemented in the D programming language and supports any neural network composed of fully connected, convolutional, and max pooling layers.

PROPERTIES AI^2 supports properties (X, C) where X is specified by a zonotope and C by a conjunction of linear constraints over the output vector's components. In our experiments, we illustrate the specification of local robustness properties where the region X is defined by a box or a line, both of which are precisely captured by a zonotope.

ABSTRACT DOMAINS The AI^2 system uses Apron, a popular library for numerical abstract domains. Therefore, our AI^2 implementation supports all abstract domains provided by Apron, such as Box [34], Zonotope [56], and Polyhedra [36].

BOUNDED POWERSET We also implemented bounded powerset domains (alternatively called disjunctive abstractions [133, 140]), which can be instantiated with any of the above abstract domains. An abstract element in the powerset domain $\mathcal{P}(\mathcal{A})$ of an underlying abstract domain $\mathcal{A}$ is a set of abstract elements from $\mathcal{A}$, concretizing to the union of the concretizations of the individual elements (i.e., $\gamma(A) = \bigcup_{a \in A} \gamma(a)$ for $A \in \mathcal{P}(\mathcal{A})$).

The powerset domain can implement a precise join operator using standard set union (potentially pruning redundant elements). However, since the increased precision can become prohibitively costly if many join operations are performed, the bounded powerset domain limits the number of abstract elements in a set to N (for some constant N).

We implemented bounded powerset domains on top of standard powerset domains using a greedy heuristic that repeatedly replaces two abstract elements in

a set by their join, until the number of abstract elements in the set is below the bound N. For joining, AI^2 heuristically selects two abstract elements that minimize the distance between the centers of their bounding boxes. In our experiments, we denote by ZonotopeN or ZN the bounded powerset domain with bound $N \geqslant 2$ and underlying abstract domain Zonotope.

2.5 EVALUATION OF AI^2

In this section, we present our empirical evaluation of AI^2. Before discussing the results in detail, we summarize our three most important findings:

- AI^2 can prove useful robustness properties for convolutional networks with 53 000 neurons and large fully connected feedforward networks with 1 800 neurons.
- AI^2 benefits from more precise abstract domains: Zonotope enables AI^2 to prove substantially more properties over Box. Further, ZonotopeN, with $N \geqslant 2$, can prove stronger robustness properties than Zonotope alone.
- AI^2 scales better than the SMT-based Reluplex [88]: AI^2 is able to verify robustness properties on large networks with $\geqslant$ 1200 neurons within few minutes, while Reluplex takes hours to verify the same properties.

In the following, we first describe our experimental setup. Then, we present and discuss our results.

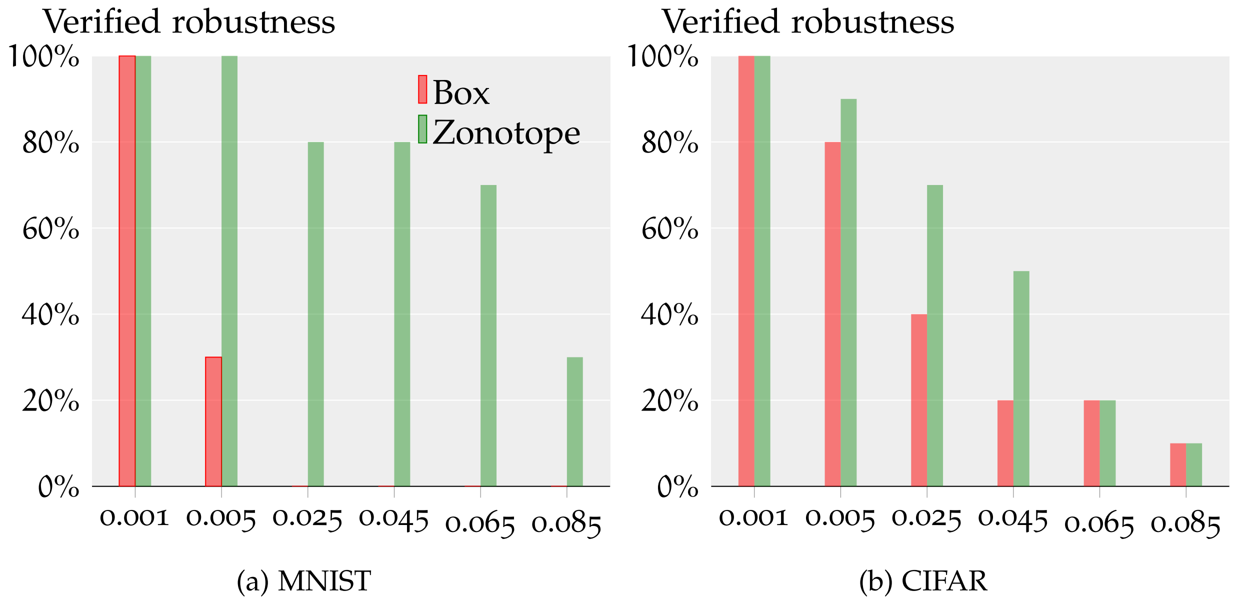
2.5.1 *Experimental Setup*

We describe the datasets, neural networks, and robustness properties used in our experiments.

DATASETS We used two popular datasets: MNIST [102] and CIFAR-10 [94] (referred to as CIFAR from now on). MNIST consists of 60 000 grayscale images of handwritten digits, whose resolution is 28×28 pixels. The images show white digits on a black background.

CIFAR consists of 60 000 colored photographs with 3 color channels, whose resolution is 32×32 pixels. The images are partitioned into 10 different classes (e.g., airplane or bird). Each photograph has a different background (unlike MNIST).

NEURAL NETWORKS We trained convolutional and fully connected feedforward networks on both datasets. All networks were trained using accelerated gradient descent with at least 50 epochs of batch size 128. The training completed when each network had a test set accuracy of at least 0.9.



(a) MNIST

(b) CIFAR

Figure 2.9: Verified properties by AI^2 on the MNIST and CIFAR convolutional networks for each bound $\delta \in \Delta$ (x-axis).

For the convolutional networks, we used the LeNet architecture [101], which consists of the following sequence of layers: 2 convolutional, 1 max pooling, 2 convolutional, 1 max pooling, and 3 fully connected layers. We write $n_{p \times q}$ to denote a convolutional layer with n filters of size $p \times q$, and m to denote a fully connected layer with m neurons. The hidden layers of the MNIST network are $8_{3\times3}, 8_{3\times3}, 14_{3\times3}, 14_{3\times3}, 50, 50, 10$, and those of the CIFAR network are $24_{3\times3}, 24_{3\times3}, 32_{3\times3}, 32_{3\times3}, 100, 100, 10$. The max pooling layers of both networks have a size of 2×2. We trained our networks using an open-source implementation [144].

We used 7 different architectures of fully connected feedforward networks (FNNs). We write $l \times n$ to denote the FNN architecture with l layers, each consisting of n neurons. Note that this determines the network's size; e.g., a 4×50 network has 200 neurons. For each dataset, MNIST and CIFAR, we trained FNNs with the following architectures: 3×20, 6×20, 3×50, 3×100, 6×100, 6×200, and 9×200.

ROBUSTNESS PROPERTIES In our experiments, we consider local robustness properties (X, C_L) where the region X captures changes to lighting conditions. This type of property is inspired by the work of [131], where adversarial examples were found by brightening the pixels of an image.

Formally, we consider robustness regions $S_{\overline{x},\delta}$ that are parameterized by an input $\overline{x} \in \mathbb{R}^m$ and a robustness bound $\delta \in [0, 1]$. The robustness region is defined as:

$$S_{\overline{x},\delta} = \{\overline{x}' \in \mathbb{R}^m \mid \forall i \in [1, m].\ 1 - \delta \leqslant x_i \leqslant x_i' \leqslant 1 \vee x_i' = x_i\}.$$

For example, the robustness region for $\overline{x} = (0.6, 0.85, 0.9)$ and bound $\delta = 0.2$ is given by the set:

$$\{(0.6, x, x') \in \mathbb{R}^3 \mid x \in [0.85, 1], x' \in [0.9, 1]\}.$$

Note that increasing the bound δ increases the region's size.

In our experiments, we used AI^2 to check whether all inputs in a given region $S_{\overline{x},\delta}$ are classified to the label assigned to $\overline{x}$. We consider 6 different robustness bounds δ, which are drawn from the set $\Delta = \{0.001, 0.005, 0.025, 0.045, 0.065, 0.085\}$.

We remark that our robustness properties are stronger than those considered in [131]. This is because, in a given robustness region $S_{\overline{x},\delta}$, each pixel of the image $\overline{x}$ can be brightened independently of the perturbations of other pixels. We remark that this is useful to capture scenarios where only part of the image is brightened (e.g., due to shadowing).

OTHER PERTURBATIONS Note that AI^2 is not limited to certifying robustness against such brightening perturbations. In general, AI^2 can be used whenever the set of perturbed inputs can be overapproximated with a set of zonotopes in a precise way (i.e., without adding too many inputs that do not capture actual perturbations to the robustness region). For example, the inputs perturbed by an ℓ_∞ attack [24] are captured exactly by a single zonotope. Further, rotations and translations have low-dimensional parameter spaces, and therefore can be overapproximated by a set of zonotopes in a precise way.

BENCHMARKS We selected 10 images from each dataset. Then, we specified a robustness property for each image and each robustness bound in Δ, resulting in 60 properties per dataset. We ran AI^2 to check whether each neural network satisfies the robustness properties for the respective dataset. We compared the results using different abstract domains, including Box, Zonotope, and ZonotopeN with N ranging between 2 and 128.

We ran all experiments on an Ubuntu 16.04.3 LTS server with two Intel Xeon E5-2690 processors and 512GB of memory. To compare AI^2 to existing solutions, we also ran the FNN benchmarks with Reluplex [88]. We have not run benchmarks on convolutional neural networks with Reluplex, as it currently does not support them.

2.5.2 *Discussion of Results*

In the following, we first present our results for convolutional networks. Then, we present experiments with different abstract domains and discuss how the domain's precision affects AI^2's ability to verify robustness. We also plot AI^2's running times for different abstract domains to investigate the trade-off between precision and scalability. Finally, we compare AI^2 to Reluplex.

PROVING ROBUSTNESS OF CONVOLUTIONAL NETWORKS We start with our results for convolutional networks. AI^2 terminated within 1.5 minutes when verifying properties on the MNIST network and within 1 hour when verifying the CIFAR network.

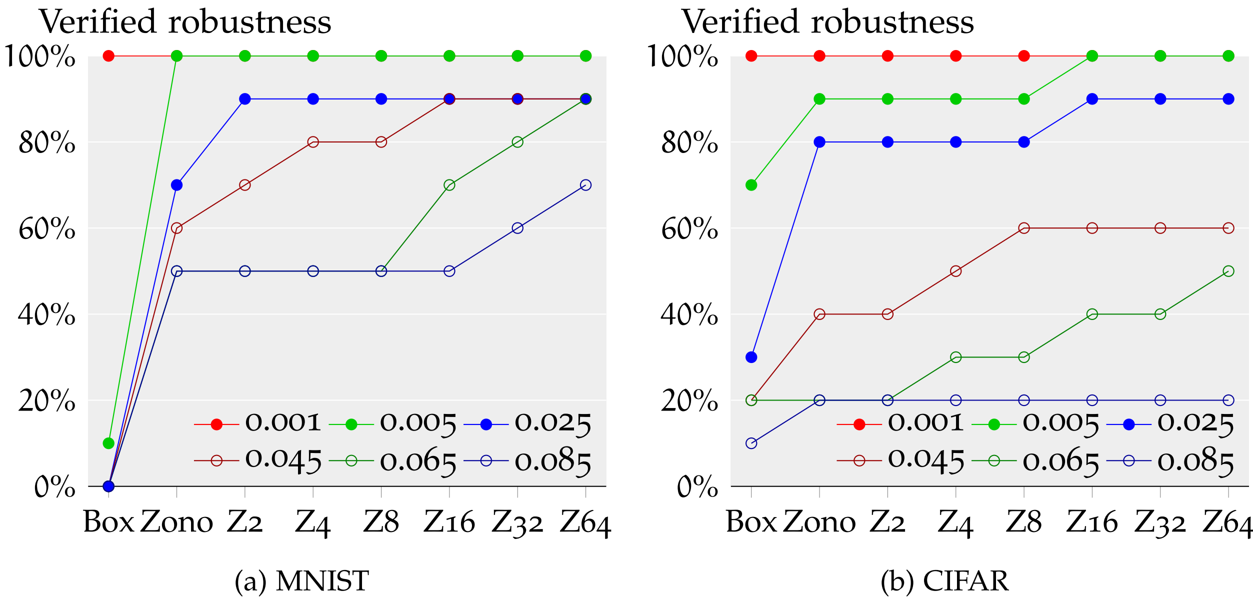


Figure 2.10: Verified properties as a function of the abstract domain used by AI^2 for the 9×200 network. Each point represents the fraction of robustness properties for a given bound (as specified in the legend) verified by a given abstract domain (x-axis).

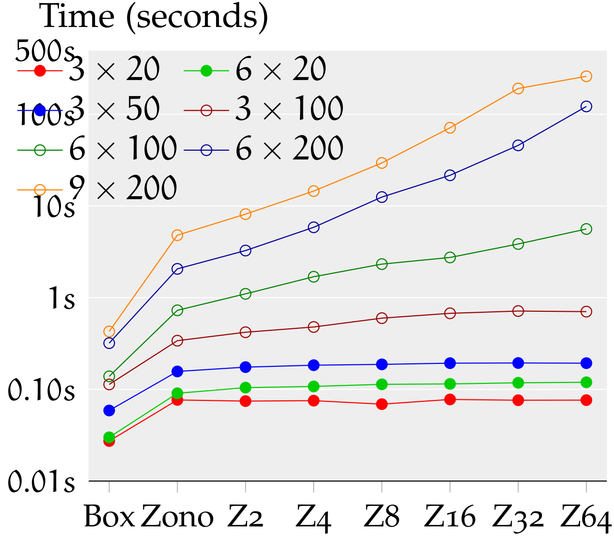


Figure 2.11: Average running time of AI^2 when proving robustness properties on MNIST networks as a function of the abstract domain used by AI^2 (x-axis). Axes are scaled logarithmically.

In Fig. 2.9, we show the fraction of robustness properties verified by AI^2 for each robustness bound. We plot separate bars for Box and Zonotope to illustrate the effect of the domain's precision on AI^2's ability to verify robustness.

For both networks, AI^2 verified all robustness properties for the smallest bound 0.001 and it verified at least one property for the largest bound 0.085. This demonstrates that AI^2 can verify properties of convolutional networks with rather wide

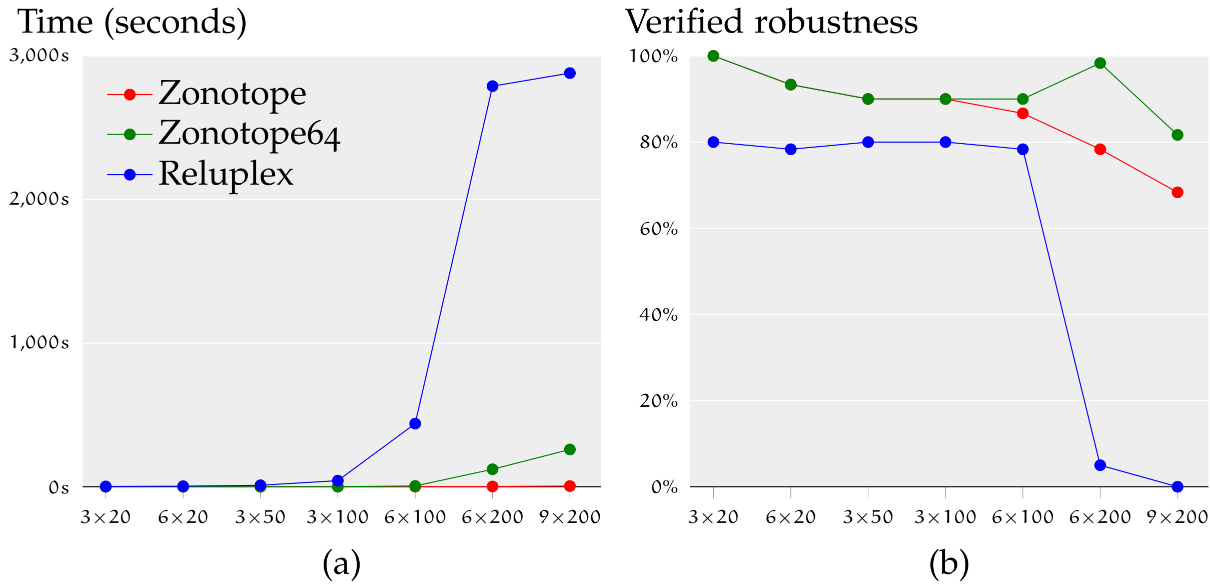


Figure 2.12: Comparing the performance of AI^2 to Reluplex. Each point is an average of the results for all 60 robustness properties for the MNIST networks. Each point in (a) represents the average time to completion, regardless of the result of the computation. While not shown, the result of the computation could be a failure to verify, timeout, crash, or discovery of a counterexample. Each point in (b) represents the fraction of the 60 robustness properties that were verified.

robustness regions. Further, the number of verified properties converges to zero as the robustness bound increases. This is expected, as larger robustness regions are more likely to contain adversarial examples.

In Fig. 2.9a, we observe that Zonotope proves significantly more properties than Box. For example, Box fails to prove any robustness properties with bounds at least 0.025, while Zonotope proves 80% of the properties with bounds 0.025 and 0.045. This indicates that Box is often imprecise and fails to prove properties that the network satisfies.

Similarly, Fig. 2.9b shows that Zonotope proves more robustness properties than Box also for the CIFAR convolutional network. The difference between these two domains is, however, less significant than that observed for the MNIST network. For example, both Box and Zonotope prove the same properties for bounds 0.065 and 0.085.

PRECISION OF DIFFERENT ABSTRACT DOMAINS Next, we demonstrate that more precise abstract domains enable AI^2 to prove stronger robustness properties. In this experiment, we consider our 9×200 MNIST and CIFAR networks, which are our largest fully connected feedforward networks. We evaluate the Box, Zonotope, and the ZonotopeN domains for exponentially increasing bounds of N between 2 and 64. We do not report results for the Polyhedra domain, which takes several days to terminate for our smallest networks.

In Fig. 2.10, we show the fraction of verified robustness properties as a function of the abstract domain used by AI^2. We plot a separate line for each robustness bound. All runs of AI^2 in this experiment completed within 1 hour.

The graphs show that Zonotope proves more robustness properties than Box. For the MNIST network, Box proves 11 out of all 60 robustness properties (across all 6 bounds), failing to prove any robustness properties with bounds above 0.005. In contrast, Zonotope proves 43 out of the 60 properties and proves at least 50% of the properties across the 6 robustness bounds. For the CIFAR network, Box proves 25 out of the 60 properties while Zonotope proves 35.

The data also demonstrates that bounded sets of zonotopes further improve AI^2's ability to prove robustness properties. For the MNIST network, Zonotope64 proves more robustness properties than Zonotope for all 4 bounds for which Zonotope fails to prove at least one property (i.e., for bounds $\delta \geqslant 0.025$). For the CIFAR network, Zonotope64 proves more properties than Zonotope for 4 out of the 5 the bounds. The only exception is the bound 0.085, where Zonotope64 and Zonotope prove the same set of properties.

TRADE-OFF BETWEEN PRECISION AND SCALABILITY In Fig. 2.11, we plot the running time of AI^2 as a function of the abstract domain. Each point in the graph represents the average running time of AI^2 when proving a robustness property for a given MNIST network (as indicated in the legend). We use a log-log plot to better visualize the trade-off in time.

The data shows that AI^2 can efficiently verify robustness of large networks. AI^2 terminates within a few minutes for all MNIST FNNs and all considered domains. Further, we observe that AI^2 takes less than 10 seconds on average to verify a property with the Zonotope domain.

As expected, the graph demonstrates that more precise domains increase AI^2's running time. More importantly, AI^2's running time is polynomial in the bound N of Zonotope$N$, which allows one to adjust AI^2's precision by increasing N.

COMPARISON TO RELUPLEX The current state-of-the-art system for verifying properties of neural networks is Reluplex [88]. Reluplex supports FNNs with ReLU activation functions, and its analysis is sound and complete. Reluplex would eventually either verify a given property or return a counterexample.

To compare the performance of Reluplex and AI^2, we ran both systems on all MNIST FNN benchmarks. We ran AI^2 using Zonotope and Zonotope64. For both Reluplex and AI^2, we set a 1 hour timeout for verifying a single property.

Fig. 2.12 presents our results: Fig. 2.12a plots the average running time of Reluplex and AI^2 and Fig. 2.12b shows the fraction of robustness properties verified by the systems. The data shows that Reluplex can analyze FNNs with at most 600 neurons efficiently, typically within a few minutes. Overall, both system verified roughly the same set of properties. However, Reluplex crashed during verification of some of

the properties. This explains why AI^2 was able to prove slightly more properties than Reluplex on the smaller FNNs.

For large networks with more than 600 neurons, the running time of Reluplex increases significantly and its analysis often times out. In contrast, AI^2 analyzes the large networks within a few minutes and verifies substantially more robustness properties than Reluplex. For example, Zonotope64 proves 57 out of the 60 properties on the 6×200 network, while Reluplex proves 3. Further, Zonotope64 proves 45 out of the 60 properties on the largest 9×200 network, while Reluplex proves none. We remark that while Reluplex did not verify any property on the largest 9×200 network, it did disprove some of the properties and returned counterexamples.

We also ran Reluplex without a predefined timeout to investigate how long it would take to verify properties on the large networks. To this end, we ran Reluplex on properties that AI^2 successfully verified. We observed that Reluplex often took more than 24 hours to terminate. Overall, our results indicate that Reluplex does not scale to larger FNNs whereas AI^2 succeeds on these networks.

2.6 COMPARING DEFENSES WITH AI^2

In this section, we illustrate a practical application of AI^2: evaluating and comparing neural network defenses. A defense is an algorithm whose goal is to reduce the effectiveness of a certain attack against a specific network, for example, by retraining the network with an altered loss function. Since the discovery of adversarial examples, many works have suggested different kinds of defenses to mitigate this phenomenon (e.g., [60, 106, 165]). A natural metric to compare defenses is the average "size" of the robustness region on some test set. Intuitively, the greater this size is, the more robust the defense.

We compared three state-of-the-art defenses:

- **GSS** [60] extends the loss with a regularization term encoding the fast gradient sign method (FGSM) attack.
- **Ensemble** [165] is similar to GSS, but includes regularization terms from attacks on other models.
- **MMSTV** [106] adds, during training, a perturbation layer before the input layer which applies the FGSM^k attack. FGSM^k is a multi-step variant of FGSM, also known as projected gradient descent.

All these defenses attempt to reduce the effectiveness of the FGSM attack [60]. This attack consists of taking a network N and an input $\overline{x}$ and computing a vector $\overline{\rho}_{N,\overline{x}}$ in the input space along which an adversarial example is likely to be found. An adversarial input $\overline{a}$ is then generated by taking a step ϵ along this vector: $\overline{a} = \overline{x} + \epsilon \cdot \overline{\rho}_{N,\overline{x}}$.

We define a new kind of robustness region, called *line*, that captures resilience with respect to the FGSM attack. The line robustness region captures all points from $\overline{x}$ to $\overline{x} + \delta \cdot \overline{\rho}_{N,\overline{x}}$ for some robustness bound δ:

$$L_{N,\overline{x},\delta} = \{\overline{x} + \epsilon \cdot \overline{\rho}_{N,\overline{x}} \mid \epsilon \in [0, \delta]\}.$$

This robustness region is a zonotope and can thus be precisely captured by AI^2.

We compared the three state-of-the-art defenses on the MNIST convolutional network described in §4.4; we call this the Original network. We trained the Original network with each of the defenses, which resulted in 3 additional networks: GSS, Ensemble, and MMSTV. We used 40 epochs for GSS, 12 epochs for Ensemble, and 10 000 training steps for MMSTV using their published frameworks.

We conducted 20 experiments. In each experiment, we randomly selected an image $\overline{x}$ and computed $\overline{\rho}_{N,\overline{x}}$. Then, for each network, our goal was to find the largest bound δ for which AI^2 proves the region $L_{N,\overline{x},\delta}$ robust. To approximate the largest robustness bound, we ran binary search to depth 6 and ran AI^2 with the Zonotope domain for each candidate bound δ. We refer to the largest robustness bound verified by AI^2 as the *verified bound*.

The average verified bounds for the Original, GSS, Ensemble, and MMSTV networks are 0.026, 0.031, 0.042, and 0.209, respectively. Fig. 2.13 shows a box-and-whisker plot which demonstrates the distribution of the verified bounds for the four networks. The bottom and top of each whisker show the minimum and maximum verified bounds discovered during the 20 experiments. The bottom and top of each whisker's box show the first and third quartiles of the verified bounds.

Our results indicate that MMSTV provides a significant increase in provable robustness against the FGSM attack. In all 20 experiments, the verified bound for the MMSTV network was larger than those found for the Original, GSS, and Ensemble networks. We observe that GSS and Ensemble defend the network in a way that makes it only slightly more provably robust, consistent with observations that these styles of defense are insufficient [73, 106].

2.7 RELATED WORK

In this section, we survey the works closely related to ours.

ADVERSARIAL EXAMPLES Szegedy et al. [162] showed that neural networks are vulnerable to small perturbations on inputs. Since then, many works have focused on constructing adversarial examples. For example, Nguyen et al. [123] showed how to find adversarial examples without starting from a test point, Tabacof and Valle [163] generated adversarial examples using random perturbations, Sabour et al. [138] demonstrated that even intermediate layers are not robust, and Grosse et al. [69] generated adversarial examples for malware classification. Other works presented ways to construct adversarial examples during the training phase, thereby

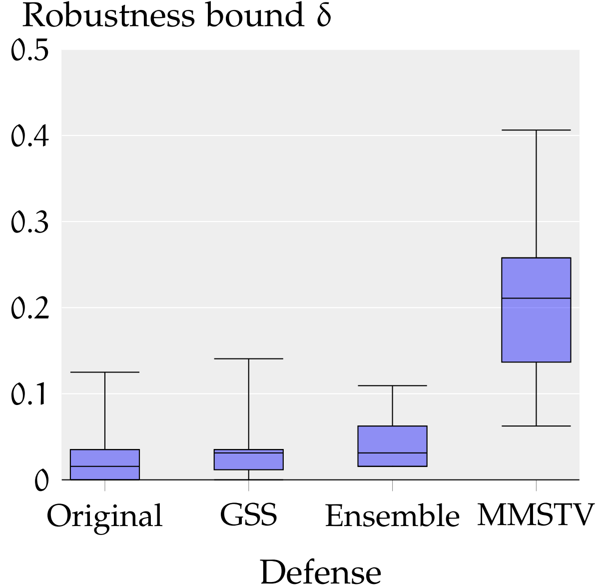


Figure 2.13: Box-and-whisker plot of the verified bounds for the Original, GSS, Ensemble, and MMSTV networks. The boxes represent the δ for the middle 50% of the images, whereas the whiskers represent the minimum and maximum δ. The inner-lines are the averages.

increasing the network robustness (see [24, 60, 70, 79, 117, 147]). Bastani et al. [14] formalized the notion of robustness in neural networks and defined metrics to evaluate the robustness of a neural network. Pei et al. [131] illustrated how to systematically generate adversarial examples that cover all neurons in the network.

NEURAL NETWORK ANALYSIS Many works have studied the robustness of networks. Pulina and Tacchella [134] presented an abstraction-refinement approach for FNNs. However, this was shown successful for a network with only 6 neurons. Scheibler et al. [144] introduced a bounded model checking technique to verify safety of a neural network for the Cart Pole system. Huang et al. [80] showed a verification framework, based on an SMT solver, which verified the robustness with respect to a certain set of functions that can manipulate the input and are *minimal* (a notion which they define). However, it is unclear how one can obtain such a set. Katz et al. [88] extended the simplex algorithm to verify properties of FNNs with ReLU.

ROBUSTNESS ANALYSIS OF PROGRAMS Many works deal with robustness analysis of programs (e.g., [28, 29, 67, 108]). Majumdar and Saha [108] considered a definition of robustness that is similar to the one in our work, and Chaudhuri et al. [29] used a combination of abstract interpretation and SMT-based methods to prove robustness of programs. The programs that have been considered in this literature tend to be small but have complex constructs such as loops and array operations. In contrast, neural networks (which are our focus) are closer to circuits, in that they lack high-level language features but are potentially massive in size.

2.8 DISCUSSION

In this chapter, we presented AI^2, the first system able to certify convolutional and large fully connected networks. The key insight behind AI^2 is to phrase the problem of analyzing neural networks in the classic framework of abstract interpretation. To achieve this, we defined abstract transformers that capture the behavior of common neural network layers and presented a bounded powerset domain that enables a trade-off between precision and scalability. Our experimental results showed that AI^2 can effectively handle neural networks that are beyond the reach of existing methods.

AI^2 has triggered substantial follow-up research, including support for other common perturbations such as rotations, smoothing, and erosion, certified training schemes, as well as more specialized domains and solvers. We will present one of those specialized domains in the next chapter.

3

FAST AND EFFECTIVE ROBUSTNESS CERTIFICATION

In Chapter 2, we have shown how to use the classic framework of abstract interpretation [35] to soundly approximate the behavior of neural networks. AI^2 successfully applies generic abstract domains to the neural network setting, leveraging 40 years of existing research. However, while the presented system scales to larger networks than those that prior approaches can handle, it still suffers from significant slowdowns and imprecision. The fundamental cause behind these issues is that we have only provided a fairly generic abstract transformer for the ReLU activation function and also have not shown how to approximate other important activation functions (e.g., Sigmoid, Tanh). Indeed, defining sound, scalable and precise abstract transformers is the most difficult aspect of analysis based on abstract interpretation. And while generic transformers tend to be easier to reason about and ensure soundness of, they often lack the scalability and precision of transformers that find creative ways to exploit the underlying properties of the abstract domain (e.g. Zonotope) and the function being approximated (e.g., ReLU). Furthermore, the evaluation of AI^2 was limited to brightening attacks [130] and did not consider L_∞-norm balls [25].

OUR CONTRIBUTIONS In this chapter, we address these limitations: Our new system DeepZ implements an abstract-interpretation-based analysis that is explicitly specialized to today's neural networks. We make the following contributions:

- We introduce new, point-wise Zonotope abstract transformers specifically designed for the ReLU, Sigmoid, and Tanh activations often used by neural networks. Our transformers minimize the area of the projection of the zonotope to the 2-D input-output plane. Further, our transformers are sound with respect to floating point arithmetic.

- We implemented both sequential and parallel versions of our transformers in DeepZ, an end-to-end automated verification system.

- We evaluated DeepZ on the task of verifying local robustness against L_∞-norm-based attacks on large MNIST and CIFAR10 feed forward and convolutional networks. In our evaluation, we considered undefended as well as defended versions of the same network (defended against L_∞ attacks using state-of-the-art defenses).

- Our experimental results demonstrate that DeepZ is more precise and faster than prior work. DeepZ precisely verifies large networks with >50 000 hidden units under L_∞-norm based perturbations within a few minutes.

To the best of our knowledge, DeepZ is more scalable than any prior system for certifying local robustness of neural networks, while guaranteeing soundness with respect to floating point operations. The work presented in this chapter has been published in Singh et al. [153].

3.1 ABSTRACT INTERPRETATION FOR VERIFYING ROBUSTNESS OF NEURAL NETWORKS

As discussed in Chapter 2, abstract Interpretation [35] is a classic method for sound and precise overapproximation of potentially unbounded or infinite sets of program behaviors. The key idea behind this framework is to define so-called abstract transformers for statements used by the program (e.g., affine arithmetic, ReLU functions, etc).

A key challenge when defining abstract transformers is striking a balance between scalability (how fast the transformer computes the approximation) and precision (how much precision it loses). Once transformers are defined, the analysis with abstract interpretation proceeds by executing them on the particular program (e.g., a neural network) and computing a final approximation (a fixed point). The relevant property can then be checked on this final approximation: if the property can be proved, then it holds for any concrete input to the program, otherwise, it may either hold but the abstraction was too coarse and unable to prove it (i.e., a false positive) or it may indeed not hold.

Verifying robustness properties of neural networks exactly is computationally expensive as it usually requires evaluating the network exhaustively on a prohibitively large set of inputs. Abstract interpretation can be leveraged for this problem by designing abstract transformers specifically for the computations used in the network, e.g., affine arithmetic and activation functions. The network can then be analyzed using these abstract transformers. For example, we can abstract a concrete input $\overline{x}$ and relevant perturbations to $\overline{x}$ (resulting in many different inputs) into one abstract element $\alpha_{\mathcal{R}}$ and then analyze the network starting from $\alpha_{\mathcal{R}}$, producing an abstract output $\alpha^{o}_{\mathcal{R}}$. We can then verify the robustness property of interest over $\alpha^{o}_{\mathcal{R}}$: if successful, it means we verified it over all concrete outputs corresponding to all perturbations of the concrete input.

Here, we consider local robustness properties $(\mathcal{R}_{\overline{x},\epsilon}, \mathcal{C}_L)$ where $\mathcal{R}_{\overline{x},\epsilon}$ represents the set of perturbed inputs around the original input $\overline{x} \in \mathbb{R}^m$ based on a small constant $\epsilon > 0$. $\mathcal{C}_L$ is a robustness condition which defines the set of outputs that all have the same label L:

$$C_L = \left\{ \overline{y} \in \mathbb{R}^n \mid \arg\max_{i \in \{1,\dots,n\}}(y_i) = L \right\}.$$

A robustness property $(\mathcal{R}_{\overline{x},\epsilon}, \mathcal{C}_L)$ holds iff the set of outputs $\mathcal{O}_{\mathcal{R}}$ corresponding to all inputs in $\mathcal{R}_{\overline{x},\epsilon}$ is included in $\mathcal{C}_L$. $(\mathcal{R}_{\overline{x},\epsilon}, \mathcal{C}_L)$ can be verified using abstract interpretation by checking if the abstract output $\alpha^{o}_{\mathcal{R}}$ resulting from analyzing the network with an abstraction of $\mathcal{R}_{\overline{x},\epsilon}$ is included in $\mathcal{C}_L$.

ZONOTOPE ABSTRACTION. In this chapter, we build on the classic Zonotope numerical abstract domain, which we discuss below. This domain was already shown to be a suitable basis for analyzing neural networks in Chapter 2. In the next section, we introduce our new abstract transformers which leverage properties of the domain and are the novel contribution of the work presented in this chapter.

Let $\mathcal{X}$ be the set of n variables. The Zonotope abstraction [55] builds on affine arithmetic by associating an affine expression $\hat{x}$ for each variable $x \in \mathcal{X}$:

$$\hat{x} := \alpha_0 + \sum_{i=1}^{p} \alpha_i \cdot \epsilon_i, \quad \text{where } \alpha_0, \alpha_i \in \mathbb{R}, \epsilon_i \in [a_i, b_i] \subseteq [-1, 1] \tag{3.1}$$

This expression consists of a center coefficient α_0, a set of noise symbols ϵ_i, and coefficients α_i representing partial deviations around the center. Crucially, the noise symbols ϵ_i can be shared between affine forms for different variables which creates implicit dependencies and constraints between the affine forms. This makes the Zonotope abstraction more powerful than an Interval abstraction which only maintains ranges of a variable x. A range $[l_x, u_x]$ can be simply derived from the affine form by computing the minimal and maximal value possible.

A zonotope $\mathcal{Z} \subseteq \mathbb{R}^n$ is represented by a collection of affine forms for all variables $x \in \mathcal{X}$, and is the set of all possible (joint) values of the affine forms for an arbitrary instantiation of the noise symbols ϵ_i. As in practice, it is impossible to compute with arbitrary real numbers, we instead use a slightly modified definition:

$$\hat{x} := [\alpha_0, \beta_0] + \sum_{i=1}^{p} [\alpha_i, \beta_i] \cdot \epsilon_i, \quad \text{where } \alpha_0, \beta_0, \alpha_i, \beta_i \in \mathbb{R}, \epsilon_i \in [a_i, b_i] \subseteq [-1, 1] \tag{3.2}$$

In this *interval affine form*, we have replaced all coefficients by intervals. All computations on intervals are performed using standard interval arithmetic. To ensure soundness with respect to different rounding modes and to account for the lack algebraic properties such as associativity and distributivity in the floating point world, the lower bounds and the upper bounds are rounded towards $-\infty$ and $+\infty$ respectively and suitable error values are added as explained in [112].

Since affine arithmetic is fast and exact for affine transformations, it is an attractive candidate for the verification of neural networks. However, the Zonotope abstraction is inherently not exact for non-linear activation functions such as ReLU, Sigmoid, and Tanh. Thus, approximation is needed, which creates a tradeoff between the cost of computation and precision. As mentioned earlier, a generic approximation of the ReLU function was proposed in Chapter 2. However, this approximation is both

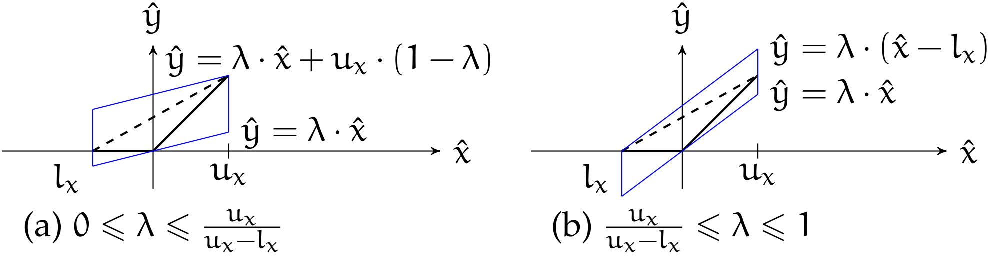


Figure 3.1: Two zonotope approximations for the ReLU function parameterized by the slope λ.

imprecise and costly as it relies on the expensive Zonotope join operator. Overall, this results in suboptimal precision and performance of the analysis.

3.2 FAST ZONOTOPE ABSTRACT TRANSFORMERS

We now introduce our fast and precise *pointwise* Zonotope abstract transformers for the ReLU, Sigmoid, and Tanh activations (Sigmoid and Tanh are not supported by the implementation in Chapter 2) and show their optimality in terms of area in the input-output plane. Our evaluation in Section 5.6 shows that our proposed approximations strike a good balance between precision and performance.

3.2.1 *ReLU*

The effect of applying the ReLU function on an input zonotope $\mathcal{Z}$ can be represented with the assignment $y := \max(0, x)$ where $x, y \in \mathcal{X}$. If x can only have positive ($l_x > 0$) or non-positive values ($u_x \leqslant 0$) in $\mathcal{Z}$, then $\hat{y} = \hat{x}$ or $\hat{y} = [0, 0]$ respectively. The affine forms for the remaining variables are not affected and the resulting zonotope is exact. When x can have both positive and negative values, then the output cannot be exactly captured by the zonotope abstraction and thus approximations are required. We define such an approximation for this case. The approximation can also be applied pointwise per layer, namely, only altering the affine form $\hat{y}$ while keeping all other affine forms in $\mathcal{Z}$ unaltered.

Fig. 3.1 shows the projections into the xy-plane of two sets of sound zonotope approximations. The projections have the shape of a parallelogram with two vertical lines and two parallel lines of slope λ, which is a parameter. To ensure soundness for all approximations in Fig. 3.1 (a), we require $0 \leqslant \lambda \leqslant \frac{u_x}{u_x - l_x}$. Similarly, $\frac{u_x}{u_x - l_x} \leqslant \lambda \leqslant 1$ for Fig. 3.1 (b). Notice that the two sets have one element in common at $\lambda = \frac{u_x}{u_x - l_x}$. Among the different candidate approximations in Fig. 3.1, we choose the one minimizing the area of the parallelogram in the xy-plane. The area $A_1(\lambda)$ of the parallelogram in Fig. 3.1 (a) is:

$$A_1(\lambda) = (1 - \lambda) \cdot u_x \cdot (u_x - l_x). \tag{3.3}$$

$A_1(\lambda)$ is a decreasing function of λ. Thus A_1 is minimized at $\lambda = \frac{u_x}{u_x - l_x}$. Similarly, the area $A_2(\lambda)$ of the parallelogram in Fig. 3.1 (b) is:

$$A_2(\lambda) = \lambda \cdot (-l_x) \cdot (u_x - l_x). \tag{3.4}$$

$A_2(\lambda)$ is an increasing function of λ and also minimized at $\lambda = \frac{u_x}{u_x - l_x}$. In summary, we obtain the following theorem:

Theorem 2. *Let $\mathcal{Z}$ be the input to a ReLU function $y = \text{ReLU}(x)$. Consider the set of pointwise Zonotope approximations $\mathcal{O}$ of the output that only alter the affine form $\hat{y}$ of the variable y. The new affine form $\hat{y}$ for the output with the minimal area in the xy-plane is given by:*

$$\hat{y} = \begin{cases} \hat{x}, & \textit{if } l_x > 0, \\ [0, 0], & \textit{if } u_x \leqslant 0, \\ [\lambda_l, \lambda_u] \cdot \hat{x} + [\mu_l, \mu_u] + [\mu_l, \mu_u] \cdot \epsilon_{new}, & \textit{otherwise.} \end{cases} \tag{3.5}$$

Here λ_l, λ_u are floating point representations of $\lambda_{\text{opt}} = \frac{u_x}{u_x - l_x}$ using rounding towards $-\infty$ and $+\infty$ respectively. Similarly, μ_l, μ_u are floating point representations of $\mu = -\frac{u_x \cdot l_x}{2 \cdot (u_x - l_x)}$ using rounding towards $-\infty$ and $+\infty$ respectively, and $\epsilon_{\text{new}} \in [-1, 1]$ is a new noise symbol.

The running time of the optimal transformer in Theorem 2 is linear in the number p of noise symbols. One can also define an optimal Zonotope transformer minimizing the volume of the output zonotope, however this is too expensive and the resulting transformer cannot be applied pointwise.

3.2.2 *Sigmoid*

The effect of applying the Sigmoid function on an input zonotope $\mathcal{Z}$ can be represented with the assignment $y := \sigma(x)$ where $x, y \in \mathcal{X}$ and $\sigma(x) = \frac{e^x}{1+e^x}$. For the assigned variable y, we have $[l_y, u_y] \subseteq [0, 1]$. When $l_x = u_x$, then $\hat{y} := \left[\frac{e^{u_x}}{1+e^{u_x}}, \frac{e^{u_x}}{1+e^{u_x}}\right]$ and the resulting zonotope is exact, otherwise the output cannot be exactly represented by a zonotope and thus approximations are required. We define pointwise approximations for the Sigmoid function such that $l_y = \sigma(l_x), u_y = \sigma(u_x)$ and then choose the one minimizing the area of its projection in the xy-plane.

Fig. 3.2 shows the projections into the xy-plane of a set of sound zonotope approximations for the output of the Sigmoid function which have $l_y = \sigma(l_x), u_y = \sigma(u_x)$. As for ReLU, the projections have the shape of a parallelogram with two vertical lines and two parallel lines of slope λ which parameterizes the set. To ensure soundness, we have $0 \leqslant \lambda \leqslant \min(\sigma'(l_x), \sigma'(u_x))$ where $\sigma'_x = \frac{e^x}{(1+e^x)^2}$.

The area $A(\lambda)$ of the parallelogram with slope λ in Fig. 3.2 is:

$$A(\lambda) = (\sigma(u_x) - \sigma(l_x) - \lambda \cdot (u_x - l_x)) \cdot (u_x - l_x) \tag{3.6}$$

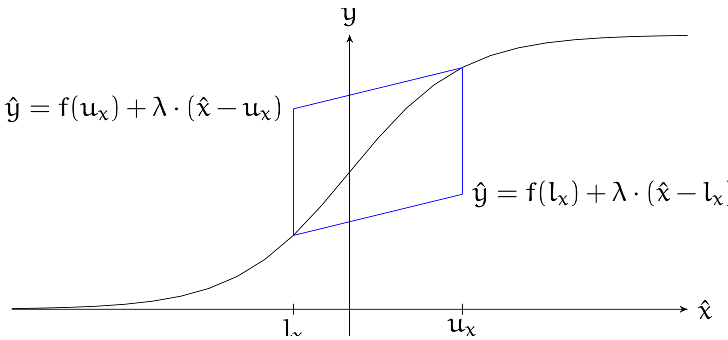


Figure 3.2: Zonotope approximation for the sigmoid function parameterized by slope λ, where $0 \leqslant \lambda \leqslant \min(f'(l_x), f'(u_x))$.

$A(\lambda)$ is a decreasing function of λ and thus $A(\lambda)$ is minimized at $\lambda_{opt} = \min(\sigma'(l_x), \sigma'(u_x))$. This yields the following theorem:

Theorem 3. *Let $\mathcal{Z}$ be the input to a smooth S-shaped*[1] *function $y = f(x)$ (such as the Sigmoid function $y = \sigma(x) = \frac{e^x}{1+e^x}$). Consider the set of pointwise Zonotope approximations $\mathcal{O}$ of the output that only alter the affine form $\hat{y}$ of the variable y and where the box concretization of $\hat{y}$ satisfies $l_y = \sigma(l_x), u_y = \sigma(u_x)$. The new affine form $\hat{y}$ for the output with the minimum area in the xy-plane is given by:*

$$\hat{y} = \begin{cases} [f(u_x)_l, f(u_x)_u], & \textit{if } l_x = u_x, \\ [\lambda_l, \lambda_u] \cdot \hat{x} + [\mu_l^1, \mu_u^1] + [\mu_l^2, \mu_u^2] \cdot \epsilon_{new}, & \textit{otherwise,} \end{cases} \tag{3.7}$$

Here, $f(u_x)_l, f(u_x)_u$ are floating point representations of $f(u_x)$ rounded towards $-\infty$ and $+\infty$ respectively and λ_l, λ_u are floating point representations of $\lambda_{opt} = \min(f'(l_x), f'(u_x))$ using rounding towards $-\infty$ and $+\infty$ respectively. Similarly μ_l^1, μ_u^1 and μ_l^2, μ_u^2 are floating point representations of $\mu_1 = \frac{1}{2}(f(u_x) + f(l_x) - \lambda_{opt} \cdot (u_x + l_x))$ and $\mu_2 = \frac{1}{2} f(u_x) - f(l_x) - \lambda_{opt} \cdot (u_x - l_x)$ computed using rounding towards $-\infty$ and $+\infty$ and adding the error due to the non-associativity of floating point addition, and $\epsilon_{new} \in [-1, 1]$ is a new noise symbol. As with ReLU, the optimal Sigmoid transformer in Theorem 3 has linear running time in the number of noise symbols and can be applied pointwise.

3.2.3 *Tanh*

The Tanh function is also S-shaped, like the Sigmoid function. A fast, optimal, and pointwise Tanh transformer can be defined using Theorem 3 by setting $f(x) = \tanh(x)$ and $f'(x) = 1 - \tanh^2(x)$.

1 A smooth function $f\colon \mathbb{R} \to \mathbb{R}$ is said to be S-shaped if $f'(x) \geqslant 0$ and there exists a value x' such that for all $x \in \mathbb{R}$, we have $f''(x) \leqslant 0 \Leftrightarrow x \leqslant x'$.

3.3 EXPERIMENTS

We now evaluate the effectiveness of our new Zonotope transformers for verifying local robustness of neural networks. Our implementation is available as an end-to-end automated verifier, called DeepZ. The verifier is implemented in Python, however, the underlying abstract transformers are implemented in C (for performance) in both the sequential and the parallel version, and are made available as part of the public ELINA [1, 151] library.

3.3.1 *Experimental setup*

EVALUATION DATASETS We used the popular MNIST [103] and CIFAR10 [95] datasets for our experiments. MNIST contains 60 000 grayscale images of size 28×28 pixels and CIFAR10 consists of 60 000 RGB images of size 32×32 pixels.

NEURAL NETWORKS Table 3.1 shows the fully connected feedforward networks (FFNNs) as well as convolutional networks (CNNs) for the MNIST and CIFAR10 datasets used in our experiments. We used both undefended and defended training procedures for training our networks. For adversarial training, we used DiffAI [114] and projected gradient descent (PGD) [41] parameterized with ϵ. In our evaluation, we refer to the undefended nets as *Point*, and to the defended networks with the name of the training procedure (either DiffAI or PGD). More details on our neural networks and the training procedures can be found in Sections 3.3.3 and 3.3.4.

ROBUSTNESS PROPERTY We consider the standard L_∞-norm-based perturbation regions $\mathcal{R}_{\overline{x},\epsilon}$ [25], where $\mathcal{R}_{\overline{x},\epsilon}$ contains all perturbed inputs $\overline{x}'$ where each pixel $\overline{x}'_i$ has a distance of at most ϵ from the corresponding pixel $\overline{x}_i$ in the original input $\overline{x}$. $\mathcal{R}_{\overline{x},\epsilon}$ can be exactly represented by a single zonotope.

BENCHMARKS We selected the first 100 images from the test set of each data set. Then, we specified a robustness property for each image using a set of robustness bounds ϵ.

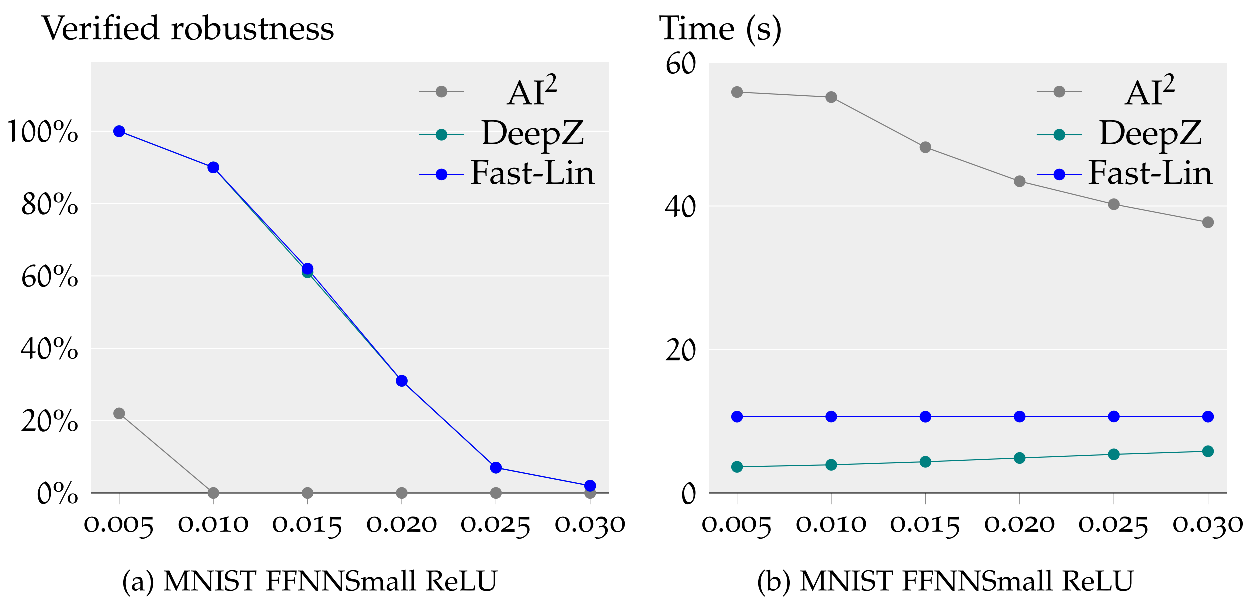
3.3.2 *Experimental results*

All experiments for the FFNNs were carried out on a 3.3 GHz 10 core Intel i9-7900X Skylake CPU with 64 GB main memory; the CNNs were evaluated on a 2.6 GHz 14 core Intel Xeon CPU E5-2690 with 512 GB main memory. We used a time limit of 10 minutes per run for all our experiments.

COMPARISON WITH PRIOR WORK We compare the precision and performance of the sequential version of DeepZ against two state-of-the-art certifiers Fast-Lin

Table 3.1: Neural network architectures used in our experiments.

Dataset	Model	Type	#Hidden units
MNIST	FFNNSmall	fully connected	610
	FFNNBig	fully connected	3 010
	ConvSmall	convolutional	3 604
	ConvMed	convolutional	4 804
	ConvBig	convolutional	34 688
	ConvSuper	convolutional	88 500
CIFAR10	FFNNBig	fully connected	3 010
	ConvSmall	convolutional	4 852
	ConvBig	convolutional	62 464



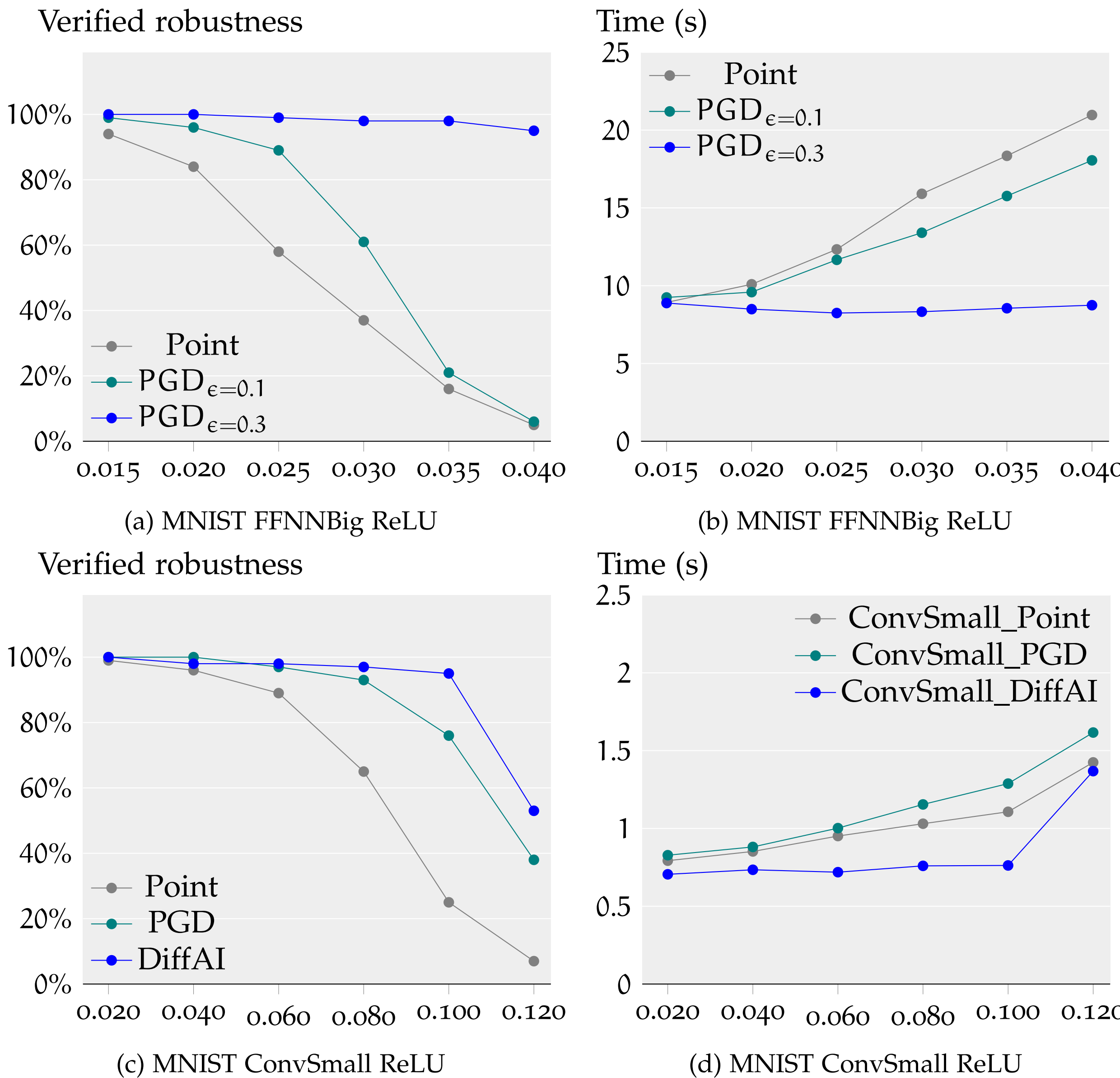
(a) MNIST FFNNSmall ReLU

(b) MNIST FFNNSmall ReLU

Figure 3.3: Comparing the performance and precision of DeepZ with the state of the art.

[170] and AI^2 on the FFNNSmall MNIST network with ReLU activation. We note that both of these certifiers support only a subset of the network architectures that DeepZ can support. Specifically, Fast-Lin only supports FFNNs with ReLU activations whereas AI^2 supports FFNNs and CNNs with ReLu activations. We also note that Fast-Lin is not sound under floating point semantics.

Fig. 3.3 shows the percentage of verified robustness and the average analysis time of all three certifiers. The values of ϵ are shown on the x-axis. DeepZ has the same precision as Fast-Lin but is up to 2.5x times faster. We note that the runtime of DeepZ increases with increasing value of ϵ; this is because the complexity of our analysis is determined by the maximum number of noise symbols in the affine form. Our ReLU transformer creates one noise symbol for any variable that can take both positive and negative values. The number of such cases rises with the increasing value of ϵ. On the other hand, AI^2 is significantly less precise and slower compared to both Fast-Lin and DeepZ. We also compared DeepZ against the duality-based certifier from [44], however it always timed out in our experiments.



(a) MNIST FFNNBig ReLU

(b) MNIST FFNNBig ReLU

(c) MNIST ConvSmall ReLU

(d) MNIST ConvSmall ReLU

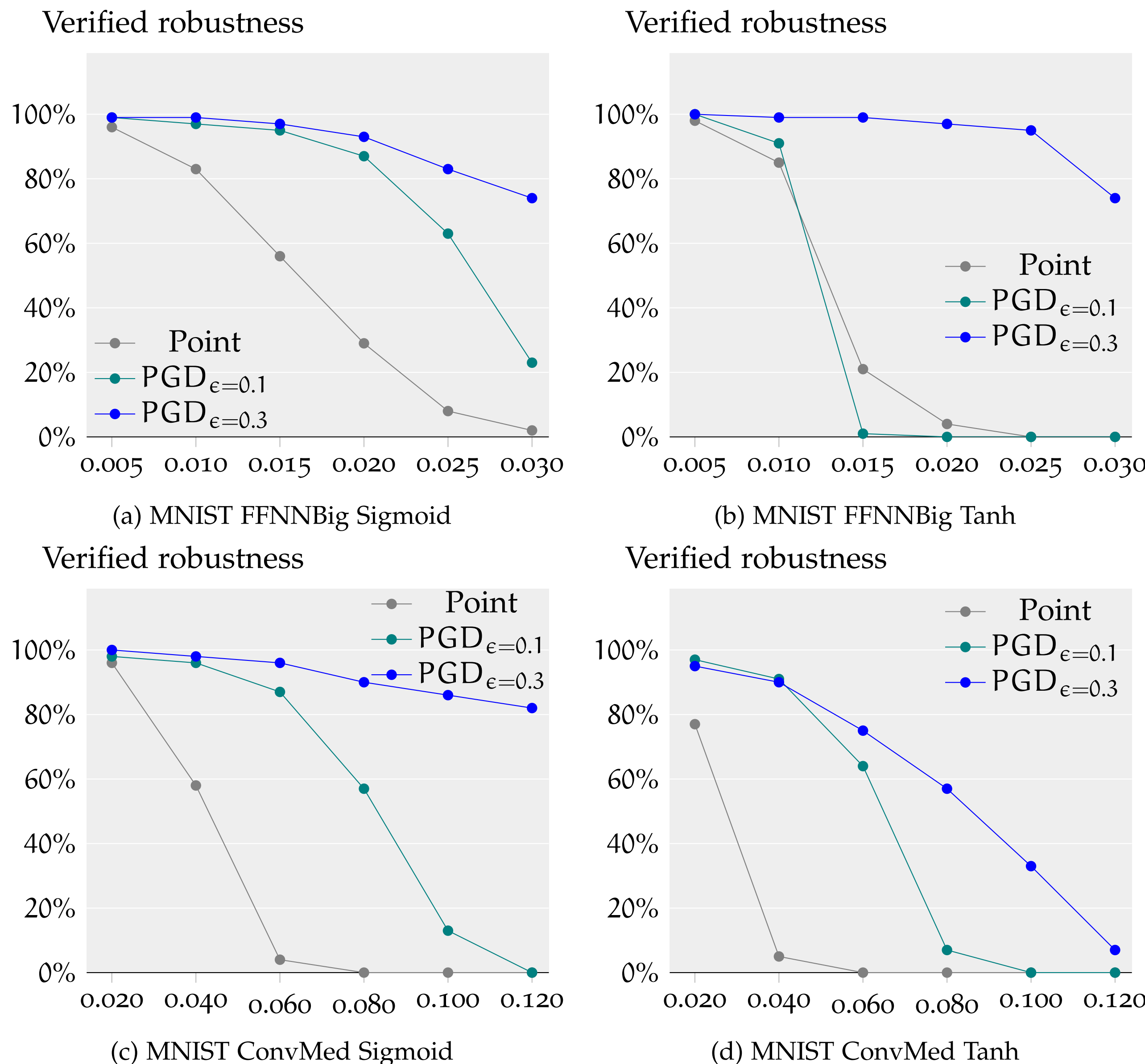
Figure 3.4: Verified robustness by DeepZ on the MNIST networks with ReLU activations.

DETAILED EXPERIMENTS Next, we evaluate DeepZ on the remaining networks using the parallelized version of our Zonotope transformers. Fig. 3.4 shows the percentage of verified robustness and the average analysis time of DeepZ for the MNIST networks with ReLU activations. DeepZ analyzes all FFNNBig networks with average runtime $\leqslant$ 22 seconds and proves 95% of the robustness properties for $\epsilon = 0.04$ for the defended $\text{PGD}_{\epsilon=0.3}$ network. DeepZ is able to analyze all ConvSmall networks with average runtime $\leqslant$ 2 seconds. It proves 95% of the robustness properties for $\epsilon = 0.1$ on the ConvSmall network defended with DiffAI. Table 3.2 shows the precision and the performance of DeepZ on ConvBig and ConvSuper networks trained with DiffAI. DeepZ proves 97% of robustness properties for the ConvSuper network containing $> 88\,000$ hidden units in 133 seconds on average.

Fig. 3.5 shows the precision and the performance of DeepZ on the MNIST FFNNBig and ConvMed networks with Sigmoid and Tanh activations. It can be

Table 3.2: Verified robustness by DeepZ on the ConvBig and ConvSuper networks trained with DiffAI.

Dataset	Model	ϵ	% verified robustness	average runtime(s)
MNIST	ConvBig	0.1	97	5
	ConvBig	0.2	79	7
	ConvBig	0.3	37	17
	ConvSuper	0.1	97	133
CIFAR10	ConvBig	0.006	50	39



(a) MNIST FFNNBig Sigmoid

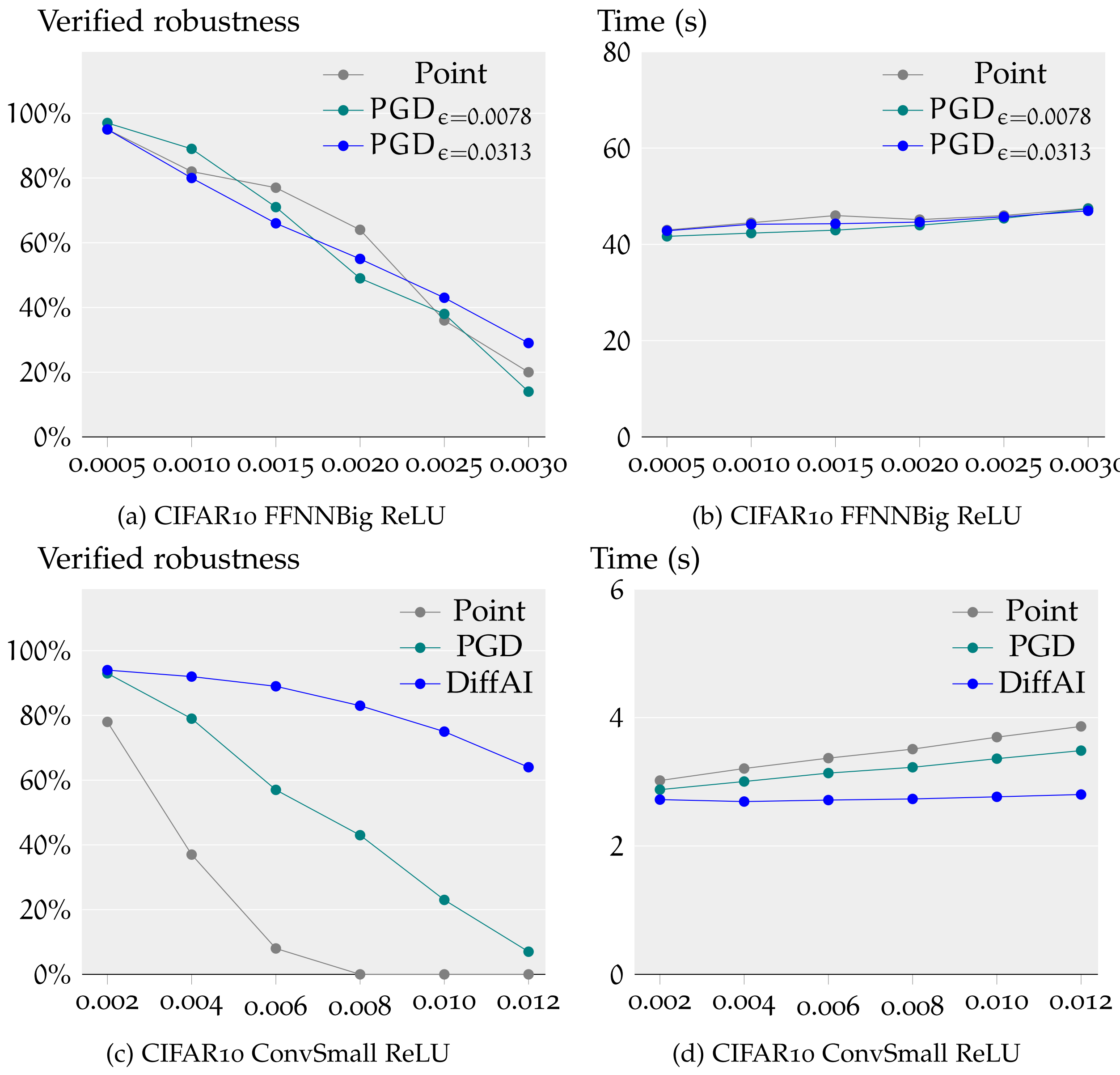
(b) MNIST FFNNBig Tanh

(c) MNIST ConvMed Sigmoid

(d) MNIST ConvMed Tanh

Figure 3.5: Verified robustness by DeepZ on the MNIST networks with Sigmoid and Tanh activations.

seen that DeepZ verifies 74% of the robustness properties on the FFNNBig Sigmoid and Tanh networks trained with PGD$_{\epsilon=0.3}$ for $\epsilon = 0.03$. DeepZ verifies 82% of the robustness properties on the ConvMed Sigmoid network for $\epsilon = 0.1$. The



(a) CIFAR10 FFNNBig ReLU

(b) CIFAR10 FFNNBig ReLU

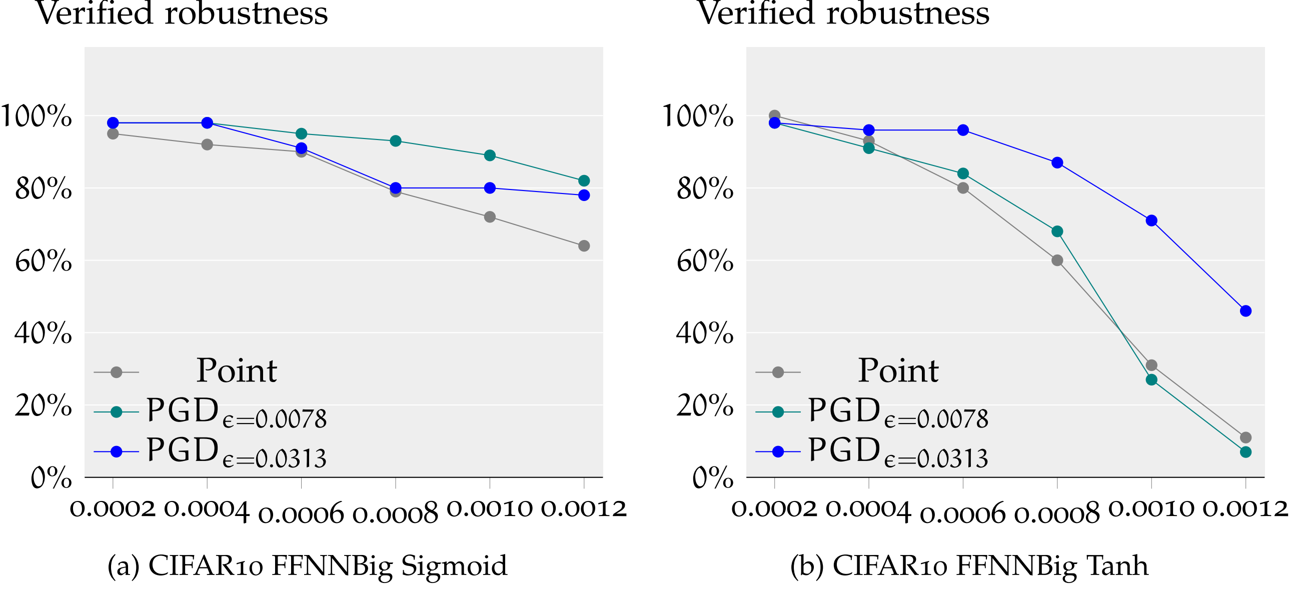
(c) CIFAR10 ConvSmall ReLU

(d) CIFAR10 ConvSmall ReLU

Figure 3.6: Verified robustness by DeepZ on the CIFAR10 networks with ReLU activations.

corresponding number for the Tanh network is 33%. We note that unlike the ReLU transformer, both Sigmoid and Tanh transformers always create a new noise symbol whenever $l_x \neq u_x$. Thus, the runtime does not increase significantly with ϵ and is not plotted. DeepZ has an average runtime of $\leqslant 35$ and $\leqslant 22$ seconds on all FFNNBig and ConvMed networks, respectively.

Fig. 3.6 shows that DeepZ has an average runtime of $\leqslant 50$ seconds for the CIFAR10 FFNNBig ReLU networks. It can be seen that the defended FFNNBig CIFAR10 ReLU networks are not significantly more provable than the undefended network. However, DeepZ verifies more properties on the defended ConvSmall networks than the undefended one and proves 75% of robustness properties on the DiffAI defended network for $\epsilon = 0.01$. DeepZ has an average runtime of $\leqslant 3$ seconds on all ConvSmall networks. DeepZ is able to verify 50% of robustness



(a) CIFAR10 FFNNBig Sigmoid

(b) CIFAR10 FFNNBig Tanh

Figure 3.7: Verified robustness by DeepZ on the CIFAR10 networks with Sigmoid and Tanh activations.

properties for ConvBig network defended with DiffAI with an average runtime of 39 seconds as shown in Table 3.2.

DeepZ verifies 82% of robustness properties on the Sigmoid network defended with PGD$_{\epsilon=0.0078}$ for $\epsilon = 0.012$ in Fig. 3.7. It verifies 46% of the robustness properties on the FFNNBig network with Tanh activation trained using PGD$_{\epsilon=0.0313}$ for the same ϵ. The average runtime of DeepZ on all networks is $\leqslant 90$ seconds.

3.3.3 *Dataset Normalization*

For each dataset we include a normalization layer (which gets applied *after* the ϵ-sized box has been calculated) using an approximated mean μ and standard deviation σ per channel as $\frac{X-\mu}{\sigma}$.

MNIST: $\mu = 0.1307, \ \sigma = 0.3081.$

CIFAR10: $\mu = [0.4914, 0.4822, 0.4465], \ \sigma = [0.2023, 0.1994, 0.2010].$

3.3.4 *Neural Networks Evaluated*

We test with six networks: one feed forward, four convolutional (without maxpool), and one with a residual connection. These are trained in various ways. In the following descriptions, we use $\text{Conv}_s C \times W \times H$ to mean a convolutional layer that outputs C channels, with a kernel width of W pixels and height of H, with a stride of s which then applies ReLU to every output. FC n is a fully connected layer which outputs n neurons without automatically applying ReLU.

For each architecture we test three versions: (i) an undefended network; (ii) a network defended with MI-FGSM (a PGD variant which we refer to as PGD in the graphs) [41] with $\mu = 1$, 22 iterations and two restarts, where the step size is $\epsilon = 5.5^{-1}$ for the ϵ used for training; (iii) a network defended with a system based on DiffAI [114].

FFNN. A 6 layer feed forward net with 500 nodes in each and an activation (ReLU, Sigmoid, Tanh) after each layer except the last.

CONVSMALL. Our smallest convolutional network with no convolutional padding.

$$x \rightarrow \text{Conv}_2 16 \times 4 \times 4 \rightarrow \text{ReLU} \rightarrow \text{Conv}_2 32 \times 4 \times 4 \rightarrow \text{ReLU} \rightarrow \text{FC } 100 \rightarrow z.$$

CONVMED. Similar to ConvSmall, but with a convolutional padding of 1. Here we test with the three activations Act = ReLU, Sigmoid, and Tanh.

$$x \rightarrow \text{Conv}_2 16 \times 4 \times 4 \rightarrow \text{Act} \rightarrow \text{Conv}_2 32 \times 4 \times 4 \rightarrow \text{Act} \rightarrow \text{FC } 1000 \rightarrow z.$$

CONVBIG. A significantly larger convolutional network with a convolutional padding of 1.

$$\begin{aligned} x &\rightarrow \text{Conv}_1 32 \times 3 \times 3 \rightarrow \text{ReLU} \rightarrow \text{Conv}_2 32 \times 4 \times 4 \rightarrow \text{ReLU} \\ &\rightarrow \text{Conv}_1 64 \times 3 \times 3 \rightarrow \text{ReLU} \rightarrow \text{Conv}_2 64 \times 4 \times 4 \rightarrow \text{ReLU} \\ &\rightarrow \text{FC } 512 \rightarrow \text{ReLU} \rightarrow \text{FC } 512 \rightarrow z. \end{aligned}$$

CONVSUPER Our largest convolutional network with no padding.

$$\begin{aligned} x &\rightarrow \text{Conv}_1 32 \times 3 \times 3 \rightarrow \text{Conv}_1 32 \times 4 \times 4 \\ &\rightarrow \text{Conv}_1 64 \times 3 \times 3 \rightarrow \text{Conv}_1 64 \times 4 \times 4 \\ &\rightarrow \text{FC } 512 \rightarrow \text{ReLU} \rightarrow \text{FC } 512 \rightarrow z. \end{aligned}$$

3.4 DISCUSSION

In this chapter, we introduced fast and precise Zonotope abstract transformers for key non-linear activations used in modern neural networks. We used these transformers to build DeepZ, an automated verifier for neural networks. We evaluated the effectiveness of DeepZ on verifying robustness of large feedforward and convolutional networks against challenging L_∞-norm attacks. Our results show that DeepZ is more precise and faster than prior work, while ensuring soundness w.r.t floating point operations.

Both AI^2 and DeepZ have triggered substantial follow-up work. However, they do not address a key aspect of intelligent systems: reasoning about probability distributions. We will consider this problem next, in Part ii of this thesis.

Part II

SYMBOLIC REASONING FOR PROBABILISTIC PROGRAMS

4

PSI: EXACT SYMBOLIC INFERENCE FOR PROBABILISTIC PROGRAMS

Many statistical learning applications make decisions under uncertainty. Probabilistic languages provide a natural way to model uncertainty by representing complex probability distributions as programs. Such programs are formed using standard language primitives from deterministic languages, as well as constructs for drawing random values and constructs for conditioning. Probabilistic programming systems (PPS) [20, 30, 54, 58, 62, 74, 110, 124, 171] provide inference algorithms that operate on expressive models specified as probabilistic programs. The key benefit of PPS is that they typically decouple the task of specifying the (generative) model from the task of constructing inference algorithms.

Exact probabilistic inference for programs with only discrete random variables is already a #P-hard computational problem [31] (although efficient algorithms exist for many interesting special cases, see e.g., Pfeffer [132]). Programs which have both discrete and continuous variables reveal additional challenges, such as representing discrete and continuous components of the joint distribution, computing integrals, and managing a large number of terms in the joint distribution. Because exact probabilistic inference is generally intractable and does not scale to complex models and large datasets, most existing PPS implement inference algorithms that calculate numerical approximations. The general approaches include sampling-based Monte Carlo methods [54, 58, 62, 110, 124, 171] and projections to convenient probability distributions, such as variational inference [20, 113] or discretization [30, 93]. While these methods scale well, they typically come with no accuracy guarantees, since providing such guarantees is NP-hard [39].

At the same time, there has been a renewed research interest in symbolic inference methods. Exact inference is important for several reasons. First, it can often outperform approximate inference for smaller models, ones that otherwise have substantial structure, or on queries with low-probability evidence. Second, it naturally supports symbolic parameters, meaning it solves a possibly infinite number of inference problems at once. Third, exact inference guarantees that no precision is lost. Finally, better support for exact inference in existing PPS will enable more fruitful combinations with approximate methods, for example preserving precision in cases where it is cheap while resorting to approximation in cases where precision is expensive or automatically providing statistical precision guarantees on approximate results.

Existing symbolic inference works fall into different categories:

- *Approximate symbolic inference:* Several analyses of graphical models approximate continuous distribution functions with a mixture of base functions, such as truncated exponentials or polynomials, which can be integrated more easily [27, 118, 142, 146, 150]. For instance, SVE [142] approximates distributions as piecewise low-rank polynomials.
- *Interactive symbolic inference:* A user can write down the inference steps within modern computer algebra systems, such as Mathematica [5] and Maple [3]. These tools can help the user by automating parts of the integration and/or simplification of distribution expressions.
- *Exact symbolic inference:* Bhat et al. [16] presents a type-based analysis translating programs with mixed discrete/continuous variables into symbolic distribution expressions, but does not simplify integral terms symbolically and instead computes them using a numerical integration library. Most recently, Hakaru [23, 122] optimizes probabilistic programs by translating a program into a distribution expression in a DSL within Maple's expression language, and simplifying this expression utilizing Maple's engine, before running (if necessary) the optimized program within a MCMC simulation.

While these works are promising steps, the practical effectiveness of exact symbolic inference in hybrid probabilistic models (with *both* discrete and continuous distributions) remains unknown, dictating the need for further investigation.

THIS CHAPTER We present the PSI (Probabilistic Symbolic Inference) system, a comprehensive approach for automating exact probabilistic inference via program analysis. PSI's analysis performs an *end-to-end symbolic inference* for probabilistic programs with discrete and/or continuous random variables. PSI analyzes a probabilistic program using a symbolic domain which captures the program's probability distribution in a precise manner. PSI comes with its own symbolic optimization engine which generates compact expressions that represent *joint probability density functions* using various optimizations, including algebraic simplification and symbolic integration. The symbolic domain and optimizations are designed to strike a balance between the expressiveness of the probability density expressions and the efficiency of automatically computing integrals and generating compact densities.

Our symbolic analysis (§4.2) generalizes existing analyzers for exact inference on discrete programs (e.g., those that operate at the level of concrete states [30]). Our optimization engine (§4.3) can also automatically simplify many integrals in density expressions and thus directly improve the performance of works that generate unoptimized expressions, such as Bhat et al. [16], without requiring the full complexity of a general computer algebra system, as in Carette and Shan [23], Narayanan et al. [122]. As a result, PSI is able to compute precise and compact inference results even when previous approaches fail (§4.4).

CONTRIBUTIONS Our main contributions are:

- *Symbolic inference for programs with continuous/discrete variables:* A novel approach for fully symbolic probabilistic inference. The algorithm represents the posterior distribution within a symbolic domain.
- *Probabilistic inference system:* PSI, an implementation of our algorithm together with optimizations that simplify the symbolic representation of the posterior distribution. PSI is available at **http://www.psisolver.org**.
- *Evaluation:* We show an experimental evaluation of PSI against state-of-the-art symbolic inference techniques – Hakaru, Maple, Mathematica, and SVE – on a corpus of 21 benchmarks selected from the literature. PSI produced correct and compact distribution expressions for more benchmarks than the alternatives. In addition, we compare PSI to state-of-the-art approximate inference engines, Infer.NET [113] and R2 [124], and show the benefits of exact inference.

Based on our results, we believe that PSI is the most effective exact symbolic inference engine to date and is useful for understanding the potential of exact symbolic inference for probabilistic programs. The work presented in this chapter has been published in Gehr et al. [50].

4.1 OVERVIEW

Fig. 4.1 presents the ClickGraph probabilistic program, adapted from a Fun language program from [113]. It describes an information retrieval model that calculates the posterior distribution of the similarity of two files, conditioned on the users' access patterns to these files.

The program first specifies the prior distribution on the document similarity (Line 2) and the recorded accesses to `A` and `B` for each user (Lines 4-5). It then specifies a trial in which the variable `sim` is the similarity of the documents for an issued query (Line 7). If the documents are similar, the probabilities of accessing them (`p1` and `p2`) are the same, otherwise `p1` and `p2` are independent (Lines 9-14). Finally, the variables `clickA` and `clickB` represent outcomes of the users accessing the documents (Lines 16-17), and each trial produces a specific observation using the `observe` statements (Lines 18-19). The return statement specifies that PSI should compute the posterior distribution of `simAll`.

4.1.1 *Analysis*

To compute the posterior distribution, PSI analyzes the probabilistic program via a symbolic domain that captures probability distributions, and applies optimizations to simplify the resulting expression after each analysis step.

```
1  def ClickGraph(){
2    simAll := Uniform(0,1);
3
4    clicksA := [1, 1, 1, 0, 0];
5    clicksB := [1, 1, 1, 0, 0];
6    for i in [0..5) {
7        sim := Bernoulli(simAll);
8
9        p1:=0; p2:=0;
10       if sim {
11           p1 = Uniform(0,1); p2 = p1;
12       } else {
13           p1 = Uniform(0,1); p2 = Uniform(0,1);
14       }
15
16       clickA := Bernoulli(p1);
17       clickB := Bernoulli(p2);
18       observe(clickA==clicksA[i]);
19       observe(clickB==clicksB[i]);
20   }
21   return simAll;
22 }
```

Figure 4.1: ClickGraph Example

SYMBOLIC ANALYSIS For each statement, the analysis computes a symbolic expression that captures the program's probability distribution at that point. The analysis operates forward, starting form the beginning of the function. As a pre-processing step, the analysis unrolls all loops and lowers the array elements into a sequence of scalars or inlined constants. The state of the analysis at each program point captures (1) the correct execution of the program as a map that relates live program variables $x_1, \ldots, x_n$ to a symbolic expression e representing a *probability density* of the computation at this point in the program, and (2) erroneous executions (e.g., due to an assertion violation) represented by an aggregate error probability expression $\bar{e}$.

The analysis of the first statement (Line 2) identifies that the state consists of the variable `simAll`, which has the $\texttt{Uniform}(0,1)$ distribution. In general, for $x := \texttt{Uniform}(a,b)$, the analysis generates the expression $[a \leqslant x] \cdot [x \leqslant b]/(b-a)$, which denotes the density of this distribution. The factors $[a \leqslant x]$ and $[x \leqslant b]$ are *Iverson brackets*, guard functions that equal 1 if their conditions are true, or equal 0 otherwise. Therefore, this density has a non-zero value only if $x \in [a,b]$. In particular, for `simAll`, the analysis generates $e_{L.2} = [0 \leqslant \texttt{simAll}] \cdot [\texttt{simAll} \leqslant 1]$.

Since the constant arrays are inlined, the analysis next processes the statement on Line 7 (in the first loop iteration). The analysis adds the variable `sim` to the state, and multiplies the expression $e_{L.2}$ with the density function for the distribution `Bernoulli(simAll)`:

$$
\begin{aligned}
e_{L.7} &= [0 \leqslant \texttt{simAll}] \cdot [\texttt{simAll} \leqslant 1] \cdot (\texttt{simAll} \cdot \delta(1-\texttt{sim}) + (1-\texttt{simAll}) \cdot \delta(\texttt{sim})) \\
\bar{e}_{L.7} &= 0
\end{aligned}
$$

The expression $e_{L.7}$ represents a generalized joint probability density over `simAll` and `sim`. To encode discrete distributions like Bernoulli, the analysis uses *Dirac deltas*, $\delta(e)$, which specify a distribution with point masses at the zeros of e.

OPTIMIZATIONS After analyzing each statement, the analysis simplifies the generated distribution expression by applying equivalence-preserving optimizations:

- basic algebraic manipulations (e.g., in the previous expression, an optimization can distribute multiplication over addition);
- removal of factors with trivial or unsatisfiable guards (e.g., in this example the analysis checks whether a product $[0 \leqslant \texttt{simAll}] \cdot [\texttt{simAll} \leqslant 1]$ is always equal to zero, and since it is not, leaves the expression unchanged);
- symbolic integration of the distribution expressions; for instance, at the end of the each loop iteration, the analysis expression $e_{L.19}$ contains several loop-local variables: `sim`, `p1`, `p2`, `clickA`, and `clickB`. The analysis integrates out these local variables because they will not be referenced by the subsequent computation. It first removes the discrete variables `sim`, `clickA`, and `clickB` by exploiting the properties of Dirac deltas. For the continuous variables `p1` and `p2`, it computes the antiderivative (indefinite integral) using PSI's integration engine, finds the integration bounds, and evaluates the antiderivative on these bounds. After the analysis of the first loop iteration, this optimization reduces the size of the distribution expression from 22 to 6 summands.

RESULT OF THE ANALYSIS After analyzing the entire program, the analysis produces the final posterior probability density expression for the variable `simAll`:

$$e_{L.21} \quad = \quad [0 \leqslant \texttt{simAll}] \cdot [\texttt{simAll} \leqslant 1] \cdot \frac{6(\texttt{simAll} + 3)^5}{3367}$$

The analysis also computes that the final error probability $\bar{e}_{L.21}$ is 0. This is the exact posterior distribution for this program. We present this posterior density function graphically in Fig. 4.8.

4.1.2 *Applications of* PSI

PSI's source language (with conditional and bounded loop statements) has the expressive power to represent arbitrary Bayesian networks, which encode many probabilistic models relevant in practice [66]. PSI's analysis is analogous to the variable elimination algorithm for inference in graphical models. We anticipate that PSI can be successfully deployed in several contexts:

PROBABILISTIC INFERENCE PSI allows a developer to specify several classes of queries. For joint posterior distribution, a user may return multiple variables in the `return` statement. The special operators `FromMarginal(e)` and `Expectation(e)` return

$$
\begin{array}{lllll}
n & \in & \mathbb{Z} & bop \in \{+,-,*,/,\hat{}\} \quad lop \in \{\&\&, ||\} \quad cop \in \{==,\neq,<,>,\leqslant,\geqslant\} \\
r & \in & \mathbb{R} & \text{Dist} \in \{\texttt{Bernoulli},\texttt{Gauss},\ldots\}\ \text{SOp} \in \{\texttt{Expectation},\texttt{FromMarginal},\texttt{SampleFrom}\} \\
x & \in & \text{Var} & p \in \text{Prog} \;\rightarrow\; \text{Func}^+ \\
a & \in & \text{ArrVar} & f \in \text{Func} \;\rightarrow\; \texttt{def}\ Id(\text{Var}^*)\ \{\text{Stmt};\ \texttt{return}\ \text{Var}^*\}
\end{array}
$$

$$
\begin{array}{lll}
se \in \text{SEx} & \rightarrow & n \mid r \mid x \mid a[\text{SEx}] \mid \text{SEx}\ bop\ \text{SEx} \mid \text{SEx}\ cop\ \text{SEx} \mid \text{SEx}\ lop\ \text{SEx} \mid \\
 & & \text{Dist}(\text{SEx}^+) \mid \text{SOp}(\text{SEx}) \mid f(\text{SEx}^*) \\
s \in \text{Stmt} & \rightarrow & x := \text{SEx} \mid a := \texttt{array}(\text{SEx}) \mid x = \text{SEx} \mid a[\text{SEx}] = \text{SEx} \mid \\
 & & \texttt{observe}\ \text{SEx} \mid \texttt{assert}\ \text{SEx} \mid \texttt{skip} \mid \text{Stmt};\ \text{Stmt} \mid \\
 & & \texttt{if}\ \text{SEx}\ \{\text{Stmt}\}\ \texttt{else}\ \{\text{Stmt}\} \mid \texttt{for}\ x\ \texttt{in}\ [\text{SEx}..\text{SEx})\ \{\text{Stmt}\}
\end{array}
$$

Figure 4.2: PSI's Source Language Syntax

the marginal distribution and the expectation of an expression `e`, respectively. A developer can also specify assertions using the `assert(e)` statement.

testing and debugging The exact inference results produced by PSI can be used as reference versions for debugging and testing approximate inference engines. It can also be used to test existing computer algebra systems – using PSI, we found errors in Maple's simplifier (see §4.4).

teaching Exact inference results can be used to illustrate the underlying meaning of probabilistic programs, counteracting the wrong notion that concepts from probability and statistics are inherently approximate.

sampling from optimized probabilistic programs Optimized distribution expressions generated by PSI's symbolic optimizer can be used, in principle, for computing proposal distributions in MCMC simulations, as done by [16] and [122].

uncertainty propagation analysis PSI's analysis can serve as a basis for static analyses that propagate uncertainty through computations and determine error bars for the result. This provides a powerful alternative to existing analyses that are primarily sampling-based [21, 139], with at most limited support for simplifying algebraic identities that involve random variables [139].

4.2 symbolic inference

In this section we describe our core analysis: the procedure analyzes each statement in the program and produces a corresponding expression in our symbolic domain which captures probability distributions.

$$
\begin{aligned}
e \in E ::= \;& x \mid n \mid r \mid \log(e) \mid \varphi(e_1,\ldots,e_n) \mid -e \mid e_1+\ldots+e_n \mid e_1\cdot\ldots\cdot e_n \mid e_1^{e_2} \mid \\
& \delta(e) \mid [e_1 = e_2] \mid [e_1 \leqslant e_2] \mid [e_1 \neq e_2] \mid [e_1 < e_2] \mid \\
& \sum_{x\in\mathbb{Z}} e[\![x]\!] \mid \int_{\mathbb{R}} dx\, e[\![x]\!] \mid (d/dx)^{-1}[e^{-x^2}](e)
\end{aligned}
$$

Figure 4.3: Symbolic domain for probability distributions

4.2.1 *Source Language*

Fig. 5.3 presents the syntax of PSI's source language. This is a simple imperative language that operates on real-valued scalar and array data. The language defines probabilistic assignments, which can assign a random value drawn from a distribution Dist, and `observe` statements, which allow constraining the values of probabilistic expressions. The language also supports the standard sequence, conditional statement, and bounded loop statement.

4.2.2 *Symbolic Domain for Probability Distributions*

Fig. 5.5 presents the syntax of our symbolic domain. The domain can succinctly describe joint probability distributions with discrete and continuous components:

- *Basic terms* include variables, numerical constants (such as e and π), logarithms and uninterpreted functions. These terms can form sums, products, or exponents. Division is handled using the rewrite $a/b \to a \cdot b^{-1}$.
- *Dirac deltas* represent distributions that have weight in low-dimensional sets (such as single points). In our analysis, they encode variable definitions and assignments, and linear combinations of Dirac deltas specify discrete distributions.
- *Iverson brackets* represent functions that are 1 if the condition within the brackets is satisfied and 0 otherwise. In our analysis, they encode comparison operators and certain primitive probability distributions (e.g., Uniform).
- *Integrals and infinite sums* are used during the analysis to represent marginalization of variables and UniformInt distributions respectively.
- *Gaussian antiderivative* – $(d/dx)^{-1}[e^{-x^2}](e)$ – used to denote the function $\int_{-\infty}^{e} dx\, e^{-x^2}$, which cannot be decomposed into simpler elementary functions.

We use the notation $e[\![x_1,\ldots,x_n]\!]$ to denote that a symbolic distribution expression e may contain free variables $x_1,\ldots,x_n$ that are bound by an outer operator (such as a sum or integral).

Our design of the symbolic domain aims to strike a balance between *expressiveness* – the kinds of distributions it can represent – and *efficiency* – the ability of the analysis to automatically integrate functions and find simple equivalent expressions.

Name (Params)	**PDF** (x, Params)	**Conditions** (x, Params)
Bernoulli (e_p)	$e_p \cdot \delta(1-x) + (1-e_p) \cdot \delta(x)$	$[0 \leqslant e_p] \cdot [e_p \leqslant 1]$
Poisson (e_λ)	$e^{-e_\lambda} \cdot \sum_{x' \in \mathbb{Z}} [0 \leqslant x'] \cdot \delta(x-x') \cdot e_\lambda^{x'} / \Gamma(x'+1)$	$[0 < e_\lambda]$
UniformInt (e_a, e_b)	$\frac{\sum_{x' \in \mathbb{Z}} \delta(x-x')[e_a \leqslant x'] \cdot [x' \leqslant e_b]}{\sum_{x' \in \mathbb{Z}} [e_a \leqslant x'] \cdot [x' \leqslant e_b]}$	$[\sum_{x' \in \mathbb{Z}} [e_a \leqslant x'] \cdot [x' \leqslant e_b] \neq 0]$
Uniform (e_a, e_b)	$[e_a = e_b] \cdot \delta(x - e_a) + [e_a \neq e_b] \cdot \frac{1}{(e_b - e_a)} \cdot [e_a \leqslant x] \cdot [x \leqslant e_b]$	$[e_a \leqslant e_b]$
Gauss (e_μ, e_ν)	$[e_\nu = 0] \cdot \delta(x - e_\mu) + [e_\nu \neq 0] \cdot \frac{\exp(-(x-e_\mu)^2/(2e_\nu))}{\sqrt{2\pi e_\nu}}$	$[0 \leqslant e_\nu]$
Pareto (e_a, e_b)	$e_a \cdot e_b^{e_a} \cdot x^{-(e_a+1)}$	$[0 \leqslant e_a] \cdot [0 \leqslant e_b]$
Beta (e_α, e_β)	$1/B(e_\alpha, e_\beta) \cdot x^{\alpha-1} \cdot (1-x)^{\beta-1} \cdot [0 \leqslant x] \cdot [x \leqslant 1]$	$[0 < e_\alpha] \cdot [0 < e_\beta]$
Gamma (e_α, e_β)	$\frac{\beta^\alpha}{\Gamma(\alpha)} \cdot x^{\alpha-1} \cdot e^{-\beta \cdot x} \cdot [0 \leqslant x]$ where $\Gamma(t) := \int_0^\infty dx\, x^{t-1} e^{-x}$.	$[0 < \alpha] \cdot [0 < \beta]$

Table 4.1: PDF and Correctness Conditions for Several Primitive Distributions.

In particular, our symbolic domain enables us to define most discrete and continuous distributions from the exponential family and other well-known primitive distributions, such as Student-t and Laplace.

PRIMITIVE DISTRIBUTIONS For each primitive distribution Dist, we define two mappings, $\mathrm{PDF}_{\mathrm{Dist}}$, and $\mathrm{Conditions}_{\mathrm{Dist}}$ to respectively specify the probability density function, and valid parameter and input ranges. For instance, the Bernoulli distribution with a parameter e_p has $\mathrm{PDF}_{\mathrm{Bern}}(x, e_p) = e_p \cdot \delta(1-x) + (1-e_p) \cdot \delta(x)$ and $\mathrm{Conditions}_{\mathrm{Bern}} = [0 \leqslant e_p] \cdot [e_p \leqslant 1]$.

Table 4.1 presents several primitive distributions encoded in PSI's intermediate language. For a distribution Dist, the function $\mathrm{PDF}_{\mathrm{Dist}}$ returns the probability density function and the function $\mathrm{Conditions}_{\mathrm{Dist}}$ returns the conditions that the parameters should satisfy. Both inputs and the parameters can be random quantities. The translation to this intermediate language from the mathematical definition of the functions is typically straightforward.

Currently, PSI supports the following distributions: `Gauss`, `Uniform`, `Exponential`, `Gamma`, `Beta`, `StudentT`, `Weibull`, `Laplace`, `Pareto`, `Rayleigh`, `Bernoulli`, `UniformInt`, and `Categorical`. Adding a new primitive distribution only requires a few lines of code.

Additionally, PSI allows the developer to specify an arbitrary PDF of the resulting distribution in PSI's intermediate language from Fig. 5.5 using the `SampleFrom(pdf_expression,...` primitive.

Additionally, PSI allows the developer to specify an arbitrary density function of the resulting distribution using the `SampleFrom (sym_expr,...)` primitive, which takes as inputs a distribution expression and a set of its parameters.

PROGRAM STATE A symbolic program state σ denotes a probability distribution over the program variables with an additional error state:

$$
\begin{aligned}
A_e :\ & \text{Expr} \to (\Sigma \to \Sigma \times E) \\
A_e(x) :=& \lambda\sigma.\,(\sigma, x) \\
A_e(se_1\ \mathit{bop}\ se_2) :=& \lambda\sigma.\,\textbf{let } (\sigma_1, e_1) = A_e(se_1)(\sigma) \textbf{ and } (\sigma_2, e_2) = A_e(se_2)(\sigma_1) \\
& \quad \textbf{in } (\sigma_2, e_1\ \mathit{SymbolicOp(bop)}\ e_2), \quad \mathit{bop} \in \{+, -, *\} \\
A_e(se_1/se_2) :=& \lambda\sigma.\,\textbf{let } (\sigma_1, e_1) = A_e(se_1)(\sigma) \textbf{ and } (\sigma_2, e_2) = A_e(se_2)(\sigma_1) \\
& \quad \textbf{in } (\text{Assert}([e_2 \neq 0])(\sigma_2), e_1 \cdot {e_2}^{-1}) \\
A_e(se_1\ \mathit{cop}\ se_2) :=& \lambda\sigma.\,\textbf{let } (\sigma_1, e_1) = A_e(se_1)(\sigma) \textbf{ and } (\sigma_2, e_2) = A_e(se_2)(\sigma_1) \textbf{ in} \\
& (\sigma_2, \text{TranslateCop}(\mathit{cop}, e_1, e_2)) \\
& \text{where} \\
& \mathit{cop} \in \{==, \neq, <, >, \leqslant, \geqslant\}, \\
& \text{TranslateCop}(==, e_1, e_2) := [e_1 = e_2], \text{TranslateCop}(\neq, e_1, e_2) := [e_1 \neq e_2], \\
& \text{TranslateCop}(<, e_1, e_2) := [e_1 < e_2], \text{TranslateCop}(>, e_1, e_2) := [e_2 < e_1], \\
& \text{TranslateCop}(\leqslant, e_1, e_2) := [e_1 \leqslant e_2], \text{TranslateCop}(\geqslant, e_1, e_2) := [e_2 \leqslant e_1]. \\
A_e(!se) :=& \lambda\sigma.\,\textbf{let } (\sigma_1, e) = A_e(se) \textbf{ in } (\sigma_1, [e = 0]) \\
A_e(se_1\&\&se_2) :=& \lambda\sigma.\,\textbf{let } (\sigma_1, e_1) = A_e(se_1)(\sigma) \textbf{ and } (\sigma_2, e_2) = A_e(se_2)(\sigma_1) \\
& \quad \textbf{in } (\sigma_2, [e_1 \neq 0] \cdot [e_2 \neq 0]) \\
A_e(se_1 \mid\mid se_2) :=& \lambda\sigma.\,\textbf{let } (\sigma_1, e_1) = A_e(se_1)(\sigma) \textbf{ and } (\sigma_2, e_2) = A_e(se_2)(\sigma_1) \\
& \quad \textbf{in } (\sigma_2, [\,[e_1 \neq 0] + [e_2 \neq 0] \neq 0\,]) \\
A_e(\text{Dist}(se_1, \ldots, se_n)) :=& \lambda\sigma.\,\textbf{let } (\sigma_1, [e_1, ..., e_n]) = A_e^*([se_1, ..., se_n])(\sigma) \textbf{ and } \text{FreshVar}(\tau) \\
& \quad \textbf{and } (P, C) = (\text{PDF}_{\text{Dist}}(\tau, e_1, ..., e_n), \text{Conditions}_{\text{Dist}}(e_1, ..., e_n)) \\
& \quad \textbf{in let } \sigma_2 = (\text{Distribute}(\tau, P) \circ \text{Assert}(C))(\sigma_1) \textbf{ in } (\sigma_2, \tau),
\end{aligned}
$$

$$
\begin{aligned}
& \text{Assert}(e[\![x_1, \ldots, x_n]\!])(\lambda M.\,\textbf{case } M \textbf{ of } (x_1, \ldots, x_n) \Rightarrow e_1[\![x_1, \ldots, x_n]\!], \bot \Rightarrow e_2) := \\
& \quad \lambda M.\,\textbf{case } M \textbf{ of } (x_1, \ldots, x_n) \Rightarrow (e_1 \cdot [e \neq 0])[\![x_1, \ldots, x_n]\!], \\
& \qquad \qquad \bot \Rightarrow e_2 + \text{MarginalizeAll}([e = 0] \cdot e_1) \\
& \text{Distribute}(x, e[\![x_1, ..., x_n, x]\!])(\lambda M.\,\textbf{case } M \textbf{ of } (x_1, ..., x_n) \Rightarrow e_1[\![x_1, ..., x_n]\!], \bot \Rightarrow e_2) := \\
& \quad \lambda M.\,\textbf{case } M \textbf{ of } (x_1, ..., x_n, x) \Rightarrow e_1[\![x_1, ..., x_n]\!] \cdot e[\![x_1, ..., x_n, x]\!], \ \bot \Rightarrow e_2
\end{aligned}
$$

Figure 4.4: Symbolic Analysis of Expressions

$$
\sigma \in \Sigma ::= \lambda M.\,\textbf{case } M \textbf{ of } (x_1, \ldots, x_n) \Rightarrow e_1[\![x_1, \ldots, x_n]\!], \ \bot \Rightarrow e_2 \tag{4.1}
$$

In a regular execution, the state is represented with the variables $x_1, \ldots, x_n$ and the posterior distribution expression e_1. We represent the error state as a symbol $\bot$ and the expression for the probability of error e_2. Conceptually, the map σ associates a probability density with each concrete program state M, which is either a tuple of values of program variables or the error state.

4.2.3 *Analysis of Expressions*

Fig. 4.4 presents the analysis of expressions. The function A_e converts each expression of the source language to a transformer $t \in \Sigma \to \Sigma \times E$ on the symbolic representation. The transformer returns both a new state ($\sigma \in \Sigma$) and a result of expression evaluation ($e \in E$), thus capturing side effects (e.g., sampling values from probability distributions or exhibiting errors such as division by zero).

OPERATIONS The first five rules transform source language variables to distribution expression variables (including operators via the helper function *SymbolicOp*). The rules are standard, with boolean constants `true` and `false` encoded as numbers 1 and 0, respectively. The rules compose the side effects of the operands. The division rule additionally uses the Assert helper function to add the guard $[e_2 \neq 0]$ to the distribution expression and aggregate the probability of $e_2 = 0$ to the overall error probability.

DISTRIBUTION SAMPLING The expression $\mathrm{Dist}(se_1, \dots, se_n)$ accepts distribution parameters $se_1, \dots, se_n$, which can be arbitrary expressions. For a primitive distribution Dist, the analysis obtains expressions from the mappings $\mathrm{PDF}_{\mathrm{Dist}}$ and $\mathrm{Conditions}_{\mathrm{Dist}}$ (§5.3).

The rule first analyzes all of the distribution's parameters (which can represent random quantities). To iterate over the parameters, the rule uses the helper function A_e^*, defined inductively as

$$\begin{aligned} A_e^*([]) &:= \lambda\sigma\,.\,(\sigma, []) \\ A_e^*(se : t) &:= \lambda\sigma\,.\,\textbf{let}\ (\sigma_1, e) = A_e(se)(\sigma)\ \textbf{and}\ (\sigma_2, t') = A_e^*(t)(\sigma_1)\ \textbf{in}\ (\sigma_2, e : t'). \end{aligned}$$

To ensure that distribution parameters have the correct values, the rule invokes a helper function Assert, which adds guards from the $\mathrm{Conditions}_{\mathrm{Dist}}$. Finally, the rule declares a *fresh* temporary variable τ (specified by a predicate FreshVar), which is then distributed according to the density function $\mathrm{PDF}_{\mathrm{Dist}}$, using the helper function Distribute. In the definitions of Assert and Distribute, we specified the states in their expanded forms (Eq. (4.1)).

$\mathrm{Distribute}(x, e[x_1, \dots, x_n, x])$ introduces the new variable x and distributes it according to $e[\![x_1, \dots, x_n, x]\!]$, by adding it as an additional factor to the existing joint distribution. Note that the free variables $(x_1 \dots x_n, x)$ in e are bound by the case statement within the distribution expression.

$$\begin{aligned}
A_s:\ \text{Stmt} &\to (\Sigma \to \Sigma)\\
A_s(\textbf{skip}) &:= \lambda\sigma.\sigma\\
A_s(x := se) &:= \lambda\sigma.\,\textbf{let}\ (\sigma', e) = A_e(se)(\sigma)\ \textbf{in}\ \text{Distribute}(x, \delta(x-e))(\sigma')\\
A_s(x = se) &:= A_s(x := se[\![\tau/x]\!]) \circ \text{Rename}(x,\tau),\ \text{with FreshVar}(\tau)\\
A_s(s_1; s_2) &:= A_s(s_2) \circ A_s(s_1)\\
A_s(\textbf{assert}(se)) &:= \lambda\sigma.\,\textbf{let}\ (\sigma', e) = A_e(se)(\sigma)\ \textbf{in}\ \text{Assert}([e \neq 0])(\sigma')\\
A_s(\textbf{observe}(se)) &:= \lambda\sigma.\,\textbf{let}\ (\sigma', e) = A_e(se)(\sigma)\ \textbf{in}\ \text{Observe}([e \neq 0])(\sigma')\\
A_s(\textbf{if}\ se\ \{s_1\}\ \textbf{else}\ \{s_2\}) &:= \lambda\sigma.\,\textbf{let}\ (\sigma_0, e) = A_e(se)(\sigma)\\
&\qquad \textbf{and}\ \sigma_1 = (A_s(s_1) \circ \text{Observe}([e \neq 0]))(\sigma_0)\\
&\qquad \textbf{and}\ \sigma_2 = (A_s(s_2) \circ \text{Observe}([e = 0]))(\sigma_0)\\
&\qquad \textbf{in}\ \text{Join}(\sigma, \sigma_1, \sigma_2)\\
A_s(\textbf{return}\ (x_1, \dots, x_n)) &:= \text{KeepOnly}(x_1, \dots, x_n)
\end{aligned}$$

Figure 4.5: Symbolic Analysis of Statements

MARGINALIZATION Marginalization aggregates the probability by summing up over the variables in an expression (e.g., local variables at the end of scope or variables in an error expression). To marginalize all variables, we define the function

$$\text{MarginalizeAll}(e[\![x_1, \dots, x_n]\!]) := \int_{\mathbb{R}} dx_1 \cdots \int_{\mathbb{R}} dx_n e[\![x_1, \dots, x_n]\!]\,.$$

The function KeepOnly performs selective marginalization. It takes as input the variables $x'_1 \dots, x'_m$ to keep and the input state σ, and marginalizes out the remaining variables in σ's distribution expressions:

$$\begin{aligned}
&\text{KeepOnly}(x'_1, ..., x'_m)(\lambda M.\,\textbf{case}\ M\ \textbf{of}\ (x_1, ..., x_n) \Rightarrow e_1[\![x_1, ..., x_n]\!]\,, \bot \Rightarrow e_2) =\\
&\quad \textbf{let}\ \{x''_1, ..., x''_l\} = \{x_1, ..., x_n\} \setminus \{x'_1, ..., x'_m\}\\
&\quad \textbf{in}\ \lambda M.\,\textbf{case}\ M\ \textbf{of}\ (x'_1, ..., x'_m) \Rightarrow \int_{\mathbb{R}} dx''_1 \cdots \int_{\mathbb{R}} dx''_l e_1[\![x_1, ..., x_n]\!]\,, \bot \Rightarrow e_2
\end{aligned}$$

4.2.4 *Analysis of Statements*

Fig. 4.5 presents the definition of function A_s: it analyzes each statement and produces a transformer of states: $\Sigma \to \Sigma$. The initial analysis state σ_0 is defined as follows: $\sigma_0 = (\lambda M.\,\textbf{case}\ M\ \textbf{of}\ \vec{x} \Rightarrow \varphi(\vec{x}), \bot \Rightarrow 0)$. Here, the function `F` under analysis has parameters $\vec{x} = (x_1, ...x_n)$ where φ is an uninterpreted function representing the joint probability density of $\vec{x}$. If `F` has no parameters, we replace $\varphi()$ with 1.

$$\begin{aligned}
&\text{Rename}(x, x')(\lambda M.\,\textbf{case}\ M\ \textbf{of}\ (x_1, ..., x, ..., x_n) \Rightarrow e_1[\![x_1, ..., x, ..., x_n]\!], \bot \Rightarrow e_2) := \\
&\quad \lambda M.\,\textbf{case}\ M\ \textbf{of}\ (x_1, ..., x', ..., x_n) \Rightarrow e_1[\![x_1, ..., x', ..., x_n]\!], \bot \Rightarrow e_2 \\
&\text{Observe}(e[\![\vec{x}]\!])(\lambda M.\,\textbf{case}\ M\ \textbf{of}\ (\vec{x}) \Rightarrow e_1[\![\vec{x}]\!], \bot \Rightarrow e_2) := \\
&\quad \lambda M.\,\textbf{case}\ M\ \textbf{of}\ (\vec{x}) \Rightarrow e_1[\![\vec{x}]\!] \cdot e[\![\vec{x}]\!], \bot \Rightarrow e_2 \\
&\text{Join}((\lambda M.\,\textbf{case}\ M\ \textbf{of}\ (\vec{x}) \Rightarrow e_1[\![\vec{x}]\!], \bot \Rightarrow e_2), \sigma_{\text{then}}, \sigma_{\text{else}}) := \\
&\quad \textbf{let}\ (\lambda M.\,\textbf{case}\ M\ \textbf{of}\ (\vec{x}) \Rightarrow e_1'[\![\vec{x}]\!], \bot \Rightarrow e_2') = \text{KeepOnly}(\vec{x})(\sigma_{\text{then}}) \\
&\quad \textbf{and}\ (\lambda M.\,\textbf{case}\ M\ \textbf{of}\ (\vec{x}) \Rightarrow e_1''[\![\vec{x}]\!], \bot \Rightarrow e_2'') = \text{KeepOnly}(\vec{x})(\sigma_{\text{else}}) \\
&\quad \textbf{in}\ \lambda M.\,\textbf{case}\ M\ \textbf{of}\ (\vec{x}) \Rightarrow e_1'[\![\vec{x}]\!] + e_1''[\![\vec{x}]\!], \bot \Rightarrow e_2'[\![\vec{x}]\!] + e_2''[\![\vec{x}]\!] - e_2[\![\vec{x}]\!]
\end{aligned}$$

Figure 4.6: Analysis of Statements - Helper Functions

DEFINITIONS The statement $x :=$ se declares a new variable x and distributes it as a point mass centered at e (the symbolic expression corresponding to se), i.e. the analysis binds x by multiplying the joint probability density by $\delta(x - e)$.

ASSIGNMENTS Analysis of assignments to existing variables ($x =$ se) consistently renames these variable and introduces a new variable with the previous name. The substitution se$[\![\tau/x]\!]$ renames x to τ in the source expression se, since the variable being assigned may itself occur in se. The function Rename(x, τ) alpha-renames all occurrences of the variable x to τ in an existing state (σ) to avoid capture (Fig. 4.6). It is necessary to rename x in se separately, because se is a source program expression, while Rename renames variables in the analysis state.

OBSERVATIONS Observations are handled by a call to the helper function Observe (Fig. 4.6), which conditions the probability distribution on the given expression being true. We do not renormalize the distribution after an observation, but only once, before reporting the final result (§4.2.5). Therefore, observations do not immediately change the error part of the distribution.

CONDITIONALS The analysis of conditionals first analyzes the condition, and then creates two copies of the resulting state σ_0. In one of the copies, the condition is then observed to be true, and in the other copy, the condition is observed to be false. Analysis of the 'then' and 'else' statements s_1 and s_2 in the corresponding states yields σ_1 and σ_2. Finally, σ_1 and σ_2 are joined together by marginalizing all locally scoped variables, including temporaries created during the analysis of the condition, and then adding the distribution and the error terms (Join; Fig. 4.6). We subtract the error probability in the original state to avoid counting it twice.

4.2.5 *Final Result and Renormalization*

We obtain the final result by applying the state transformer obtained from analysis of the function body to the initial state and *renormalizing* it. We define the renormalization function as

$$\begin{aligned}&\text{Renormalize}(\lambda M.\,\textbf{case}\ M\ \textbf{of}\ (x_1,\dots,x_n) \Rightarrow e_1[\![x_1,\dots,x_n]\!], \bot \Rightarrow e_2) :=\\ &\quad \textbf{let}\ e_Z = \text{MarginalizeAll}(e_1) + e_2\\ &\quad \textbf{in}\ \lambda M.\,\textbf{case}\ M\ \textbf{of}\ (x_1,\dots,x_n) \Rightarrow [e_Z \neq 0]\cdot e_1[\![x_1,\dots,x_n]\!]\cdot e_Z^{-1},\\ &\qquad\qquad\qquad\qquad\qquad\qquad \bot \Rightarrow [e_Z \neq 0]\cdot e_2 \cdot e_Z^{-1} + [e_Z = 0]\end{aligned}$$

The function obtains a normalization expression e_Z, such that the renormalized distribution expression of the resulting state integrates to 1. This way, PSI computes a normalized joint probability distribution for the function results that depends symbolically on the initial joint distribution of the function's arguments.

4.2.6 *Discussion*

LOOP ANALYSIS PSI analyzes loops like `for i in [0..N){...}` (as mentioned in §4.1.1) by unrolling the loop body a constant N number of times. This approach also extends to loops where the number of iterations N is a random program variable. If N can be bounded from above by a constant N_{max}, a developer can encode the loop as

```
assert (N <= Nmax);
for i in [0..Nmax) {
   if (i < N) {  /* loop body */  }
}
```

To handle `for`-loops with unbounded random variables and general `while`-loops, a developer can select N_{max} such that the probability of error (i.e., probability that the loop runs for more than N_{max} iterations) is small enough. We anticipate that this approach can be readily automated. Related techniques such as [141] and [48] employ similar approximation techniques.

FUNCTION CALL ANALYSIS PSI can analyze multiple functions, generating for each function $f(x_1, ..., x_n)$ the density expression of its m outputs, parameterized by the unknown distribution of the function's n inputs. The distribution of the function inputs is represented by an uninterpreted function $\varphi(x_1, ..., x_n)$ which appears as a subterm in the output density expression.

We describe the distribution expression transformer corresponding to some source language function f with n arguments and m return values by expressions $e_f[\![x'_1,\dots,x'_m]\!]$ and e'_f each containing φ, which is an uninterpreted function with

n arguments. The expressions describe the distribution of the results and the probability of error respectively. Such expressions can be obtained by running our analysis on the function body in the initial state

$$\begin{aligned}\sigma_0 =(&\lambda M.\, \textbf{case } M \textbf{ of}\\ &(x_1,\dots,x_n) \Rightarrow \varphi(x_1,\dots,x_n)\\ &\bot \Rightarrow 0.\end{aligned}$$

and extracting the two distribution expressions from the final state.

The rule for the analysis of function calls first creates temporary variables $a_1,\dots,a_n$ for each argument of f, and variables $r_1,\dots,r_m$ for each result returned by f. The variables $a_1,\dots,a_n$ are then initialized by the actual parameters $e_1,\dots,e_n$ by multiplying the density of the caller by $\prod_i \delta(a_i - e_i)$. The result variables in f's density expression are renamed to match $r_1,\dots,r_n$, and the uninterpreted function φ within f's density expression is replaced with the new density of the caller (avoiding variable capture). but shouldn't be.

Formally, the rule for translating a function call is given by

$$\begin{aligned}A_e(f(se_1,...,se_n)) :=\lambda\sigma.\, &\textbf{let } \text{FreshVars}(a_1,\dots,a_n) \textbf{ and } \text{FreshVars}(r_1,\dots,r_m)\\ &\textbf{and } e_f''[\![r_1,\dots,r_m]\!] := e_f[\![r_1/x_1']\!]\cdots[\![r_m/x_m']\!]\\ &\textbf{and } \sigma' := (A_s(a_n := se_n)\circ\cdots\circ A_s(a_1 := se_1))(\sigma)\\ &\textbf{and } \sigma'' := \lambda M.\, \textbf{case } M \textbf{ of}\\ &(x_1,\dots,x_k,r_1,\dots,r_m) \Rightarrow\\ &\quad e_f''[\![(\lambda(a_1,\dots,a_n).\, e_c[\![x_1,\dots,x_k,a_1,\dots,a_n]\!])/\varphi]\!],\\ &\bot \Rightarrow\ e_c' + \int_{\mathbb{R}^k} dx_1...dx_k e_f'[\![(\lambda(a_1,...,a_n).\, e_c[\![x_1,...,x_k,a_1,...,a_n]\!])/\varphi]\!]\\ &\textbf{in } (\sigma'',(r_1,\dots,r_m))\\ &\text{where } e_c[\![x_1,\dots,x_k,a_1,\dots,a_n]\!] \text{ and } e_c' \text{ are expressions with}\\ &\sigma' = \lambda M.\, \textbf{case } M \textbf{ of } (x_1,\dots,x_k,a_1,\dots,a_n) \Rightarrow\\ &\qquad e_c[\![x_1,...,x_k,a_1,...,a_n]\!]\\ &\qquad \bot \Rightarrow\ e_c'\end{aligned}$$

FORMAL ARGUMENT A standard approach to prove that the translation from the source language to the target domain (in our case, the symbolic domain) is correct is to show that the transformation preserves semantics, as in [45]. This requires a specification of semantics for both the source language and the symbolic domain language. Below, we outline how one might approach such a formal proof using denotational semantics that map programs and distribution expressions to measure transformers.

Denotational semantics for the source language is easy to define by extending [92], but defining the measure semantics for distribution expressions is more challenging. Defining measure semantics for most expression terms in the symbolic domain is simple (e.g., the measure corresponding to a sum of terms is the sum of the measures of the terms). However, the semantics of expressions containing Dirac deltas is less immediate, since there is no general pointwise product when Dirac delta factors have overlapping sets of free variables.

To assign semantics to a product expression with Dirac delta factors, we therefore (purely formally) integrate the expression against the indicator function of the measured set and simplify it using Dirac delta identities until no Dirac deltas are left. The resulting term can then be easily interpreted as a measure. A formal proof will also need to show that this is a well-formed definition, i.e., that all ways of eliminating Dirac deltas lead to the same measure. Once the semantics for distribution expressions has been defined, the correctness proof proceeds as a straightforward induction over the source language production rules. We consider a complete formalized proof to be an interesting future work item.

4.3 symbolic optimizations

After each step of the analysis from §4.2, PSI's symbolic engine simplifies the joint posterior distribution expressions. The algorithm of this optimization engine is a fixed point computation, which applies various symbolic transformations. We selected these transformations by their ability to optimize expressions that typically arise when analyzing probabilistic programs and that have demonstrated their efficiency for practical programs (as we discuss in §4.4). We next describe three main groups of the transformations.

4.3.1 *Algebraic Optimizations*

These optimizations implement basic algebraic identities. Some examples include removing zero-valued terms in addition expressions, removing one-valued terms in multiplication expressions, distributing exponents over products, or condensing equivalent summands and factors.

4.3.2 *Guard Simplifications*

For each term in an expression with multiple Iverson brackets and/or Dirac deltas, these optimizations analyze the constraints in the bracket factors and delta factors using sound but incomplete heuristics. PSI can then (1) remove the whole term if the constraints are inconsistent and therefore the term is always zero, (2) remove a factor if it is always satisfied, e.g., if both sides of an inequality are constants, or (3) remove a bracket factor if it is implied by other factors.

GUARD NORMALIZATION Internally, all brackets are represented using only the forms $[e=0], [e\neq 0]$ and $[e\leqslant 0]$. If Iverson brackets occur nested within Dirac deltas or other Iverson brackets, these enclosing deltas or brackets are replaced by an exhaustive case distinction on all possible combinations of Iverson bracket conditions, pruning unsatisfiable conditions if possible. For example, the expression $[[x=0]\cdot[y\leqslant 0]=0]$ is transformed into $[x=0]\cdot[y\leqslant 0]\cdot[1\cdot 1=0]+[x\neq 0]\cdot[y\leqslant 0]\cdot[0\cdot 1=0]+[x=0]\cdot[-y\leqslant 0]\cdot[y\neq 0]\cdot[1\cdot 0=0]+[x\neq 0]\cdot[-y\leqslant 0]\cdot[y\neq 0]\cdot[0\cdot 0=0]$ which is in turn simplified to $[x\neq 0]\cdot[y\leqslant 0]+[x=0]\cdot[-y\leqslant 0]\cdot[y\neq 0]+[x\neq 0]\cdot[-y\leqslant 0]\cdot[y\neq 0]$.

GUARD LINEARIZATION Guard linearization analyzes complex Iverson brackets and Dirac deltas with the goal to rewrite expressions in such a way that all included constraints (expressions in Iverson brackets and Dirac deltas) depend on a specified variable x in a linear way. It handles constraints that are easily recognizable as compositions of quadratic polynomials, multiplications with only one factor depending on x and integer and fractional powers (including in particular multiplicative inverses).

One aspect that requires special care is that the integral of a Dirac delta along x depends on the partial derivative of its argument in the direction of x. For example, we have $\delta(2x)=\frac{1}{2}\delta(x)$, and in general we have $\delta(f(x))=\sum_i \frac{\delta(x-x_i)}{|f'(x_i)|}$ for $f(x_i)=0$, whenever $f'(x_i)\neq 0$. We ensure the last constraint by performing a case split on $f'(x)=0$, and substituting the solutions for x into the delta expression in the "equals" case. For example, $\delta(y-x^2)$ is linearized to

$$[-y\leqslant 0]\cdot([x=0]\cdot\delta(y)+[x\neq 0]\cdot\frac{1}{2\sqrt{y}}(\delta(x-\sqrt{y})+\delta(x+\sqrt{y}))).$$

We now present the details of the guard linearization algorithm.

To transform an expression into an equivalent expression that only contains guards depending linearly on a given variable x, all subexpressions that are Iverson brackets or Dirac deltas are rewritten. In fact, our algorithm will isolate the variable x from all other variables in case it succeeds (i.e., the variable is an isolated summand in the constraints it occurs in). Guard linearization is a sound but incomplete heuristic that works for many practically relevant constraints; it does not solve arbitrary non-linear constraints (this is an undecidable problem). We handle constraints that are easily recognizable as compositions of quadratic polynomials, multiplications with only one factor depending on x and integer and fractional powers (including in particular multiplicative inverses).

For the following, we assume that Dirac delta constraint expressions have only countably many zeros. Due to guard normalization, we can assume that all such expressions are differentiable at all points where they are defined.

Note that this discussion is not entirely formal; expressions are sometimes used interchangeably with their meaning. Furthermore, we treat additional assumptions that hold for the duration of a recursive call implicitly.

For a given Iverson bracket or Dirac delta, first, the constraint expression (i.e. e in $\delta(e)$ or $[e \leqslant 0], [e = 0], [e \neq 0]$) is normalized by distributing products over sums containing the variable x. We then use a recursive algorithm to linearize the constraint. The recursion state contains four variables p, l and r; each of those contains an expression from our distribution expression language. The recursive algorithm maintains different invariants for the different kinds of guards:

- Iverson brackets of the form $[e \leqslant 0]$:
  $[e \leqslant 0]$ is equivalent to $[0 \leqslant p] \cdot [l \leqslant r] + [p < 0] \cdot [r \leqslant l]$.
- Iverson brackets of the form $[e = 0]$:
  $[e = 0]$ is equivalent to $[l = r]$.
- Dirac deltas $\delta(e)$:
  $\delta(e)$ is equivalent to $\sum_{x' \in \mathbb{R}: l[\![x'/x]\!] = r, (\frac{\partial}{\partial x} e)[\![x'/x]\!] \neq 0} \frac{\delta(x - x')}{(\frac{\partial}{\partial x} e)[\![x'/x]\!]} + [\frac{\partial}{\partial x} e = 0] \cdot \delta(e)$.

Furthermore, throughout the execution of the recursive algorithm, x does not occur free in r. p is ignored for guards that are not of the shape $[e \leqslant 0]$.

The invariants, together with the information which type of Iverson bracket/Dirac delta is being handled allow computation of a guard which is equivalent to the original guard at any point in the recursive algorithm (except for Dirac deltas, where we additionally need to be able to solve the equation $l = r$). Intuitively, the recursive algorithm recurses, maintaining the invariant, until the original guard can be either easily written in a linear shape given the invariant, or it can be determined that linearization is not supported.

We start the recursion with $p = 1, l = e$ and $r = 0$ such that the invariants are satisfied.

One invocation of the recursive algorithm performs the following case distinction on the shape of l:

- Case $l = l_1 + \cdots + l_n$:
  The summands are partitioned according to whether x occurs free in them. In case only one summand l_i has this property, the algorithm recurses with $p' = p, l' = l_i$ and $r' = r - l_i$. Otherwise, if $l - r$ can be written as a quadratic polynomial $a \cdot x^2 + b \cdot x + c$, the algorithm performs a symbolic case split on $a = 0$: It recurses with $l' = b \cdot x + c, r' = r$ and multiplies the result by $[a = 0]$. Additionally, it creates symbolic expressions for the discriminant $d = b^2 - 4ac$ and the two roots $z_1 = (-b - \sqrt{d})/(2 \cdot a), z_2 = (-b + \sqrt{d})/(2 \cdot a)$. From those expressions, an appropriate expression handling the nondegenerate quadratic can be built using moderately complex expressions with Iverson brackets/Dirac deltas depending at most linearly on x. (The related case

of an arbitrary even power is discussed below in more detail. This should clarify how this expression is built.) This expression is computed (for $[e \neq 0]$, $[e = 0]$ and $\delta(e)$, this involves additional recursive calls with $l' = x, r' = z_{1,2}$), multiplied by $[a = 0]$ and added to the expression handling the degenerate case.

- Case $l = l_1 \cdots l_n$:
  The summands are partitioned according to whether x occurs free in them. In case only one summand l_i has this property, the algorithm performs a symbolic case split on $0 = L := l_1 \cdots l_{i-1} \cdot l_{i+1} \cdots l_n$ (by encoding both branches symbolically and multiplying them with the appropriate Iverson brackets $[L = 0]$ and $[L \neq 0]$ respectively, both not depending on x): For $L = 0$, the condition reduces to an Iverson bracket/Dirac delta of the original kind with an expression of $-r$. As $-r$ does not depend on x, such guards have no non-linear dependencies on x. For $L \neq 0$, the algorithm recurses with $p' = p \cdot L, l' = l_i$ and $r' = r/L$. Linearization is not performed for (non-polynomial) constraints with more than one x-dependent factor.

- Case $l = l_1^{l_2}, 0 > l_2$ constant :
  The algorithm computes the expression $i = l_1^{-l_2}$ and performs a symbolic case split on $r = 0$. For $r \neq 0$, the algorithm recurses with $p' = -p \cdot r \cdot i$, $l' = i$, $r' = r^{-1}$. For $r = 0$, the result depends on the kind of guard under consideration. For $[e \leqslant 0]$, guard linearization is applied recursively to the expression $[p \cdot i \leqslant 0]$. For $[e \neq 0]$, the result is given by 1 and for $[e = 0]$ it is 0. (This exploits the fact that the inverse of a real number is never zero.)

- Case $l = l_1^n, n$ even integer :
  The algorithm first computes symbolic expressions for the two roots $z_2 = r^{1/n}, z_1 = -z_2$. The result depends on the kind of guard under consideration.
  - For $[e \leqslant 0]$: Constraint linearization is performed recursively on $[0 \leqslant p]$. The algorithm performs a symbolic case split on the result. For $0 \leqslant p$, the algorithm recurses with $p' = -1, l' = l_1, r' = z_1$ and $p'' = 1, l'' = l_1, r' = z_2$, multiplies the results (in order to get an appropriate expression describing the constraint that l_1 is between the two roots) and multiplies with a factor $[r \geqslant 0]$ (in order to ensure that the subexpressions z_1 and z_2 are well-formed). For $p < 0$, the algorithm recurses with $p' = 1, l' = l_1, r' = z_1$ and $p'' = -1, l'' = l_1, r'' = z_2$ adds the results (in order to get an appropriate expression describing the constraint that l_1 is not strictly inbetween the two roots), after multiplying the second summand with $[z_2 \neq 0]$ (in order to avoid double-counting a root at 0) and multiplies with a factor $[0 < r]$ (in order to ensure that the subexpressions z_1 and z_2 are well-formed) finally, the expression $[r \leqslant 0]$ is added as a summand (as all even powers are at least 0).

  - For $[e \neq 0]$ and $[e = 0]$, the algorithm recurses with $l' = l_1, r' = z_1$ and $l'' = l_1, r'' = z_2$, multiplies/adds (avoiding double-counting) the results respectively, and multiplies with a factor $[0 \leqslant r]$. For $[e \neq 0]$, the algorithm finally adds a summand $[r < 0]$

- Case $l = l_1^n, n$ odd integer:
  The algorithm performs a symbolic case split on $0 \leqslant r$. For $0 \leqslant r$, the algorithm recurses with $p' = p$, $l' = l_1$ and $r' = r^{1/n}$. For $r < 0$, the algorithm recurses with $p' = p$, $l' = l_1$ and $r' = -(-r)^{1/n}$ (this is necessary because the simplifier never considers power expressions with negative base and fractional exponent well-formed).

- Case $l = l_1^{m/n}, m, n$ integers:
  The algorithm recurses on $p' = p, l' = l_1, r' = r^{n/m}$ and multiplies the result with $[0 \leqslant r]$ (in order to make sure $r^{n/m}$ is well-formed). For $[e \leqslant 0]$, the algorithm additionally adds a summand $p' \cdot [r < 0]$ where p' is the result of running constraint linearization recusively on $[p < 0]$. For $[e < 0]$, the algorithm additionally adds a summand $[r < 0]$.

- Case $l = x$:
  For Iverson brackets, the algorithm can simply return the precise expressions which are known to be equivalent to the original guards due to the invariants, as those already have the required shape.

  For Dirac deltas, the sum simplifies, as the only possible value for x' is r, which does not depend on x, and hence the remaining Dirac delta has a constraint expression of the right shape. The only term in the invariant which is potentially problematic is $[\frac{\partial}{\partial x} e = 0] \cdot \delta(e)$. Intuitively, our approach will be to solve the equation $\frac{\partial}{\partial x} e = 0$ for x. Then we can substitute x for its solution within $\delta(e)$, which makes the Delta completely independent of x. The algorithm simply calls constraint linearization recursively on $[\frac{\partial}{\partial x} e = 0]$ and distributes products over sums and simplifies the result. Then the algorithm iterates over all summands of the resulting expression. For each summand s, the algorithm iterates over all factors and tries to find an Iverson bracket of the shape $[x - e' = 0]$. The algorithm then computes $s \cdot \delta(e[\![e'/x]\!])$. All of these expressions are then added together to form the expression e''. In case the algorithm is not able to determine a suitable e' for some summand, guard linearization fails. The result is given by $\text{linearize}([\frac{\partial}{\partial x} e = 0]) \cdot \frac{\delta(x-r)}{\frac{\partial}{\partial x} e} + e''$. (Where "linearize" performs guard linearization recursively.)

- Otherwise: If l is not given in any of the discussed shapes, guard linearization fails.

4.3.3 *Symbolic Integration*

These optimizations replace the integration terms with equivalent terms that do not contain integration symbols. If the integrated term is a sum, the symbolic engine integrates each summand separately and pulls all successfully integrated summands out of the integral. If the integrated term is a product, the symbolic engine pulls out all factors that do not contain the integration variable before performing the integration.

INTEGRATION OF TERMS WITH DELTAS The integration engine first attempts to eliminate the integration variable with a factor that is a Dirac delta, by applying the rule $f(e) = \int_{\mathbb{R}} dx f(x) \cdot \delta(x - e)$. The engine can often transform deltas that depend on the integration variable x in more complicated ways into equivalent expressions only containing x-dependent deltas of the above form, using guard linearization. This transformation is applied when evaluating the integral.

INTEGRATION OF CONTINUOUS TERMS The symbolic engine integrates continuous terms (without Dirac deltas) in several steps. First, it multiplies out all terms that contain the integration variable and groups together all Iverson bracket terms in a single term. Second, it computes the lower and upper bounds of integration by analyzing the Iverson bracket term. If necessary, it first rewrites the term into an equivalent term within which all Iverson brackets specify the constraints on the integration variable in a direct fashion, using guard linearization. This is necessary as in general, a single condition inside a bracket might not be equivalent to a single lower or upper bound for the integration variable. The integration bounds are then computed as the minimum of all upper bounds and the maximum of all lower bounds. (Using the encoding $\min(a, b) = [a \leqslant b] \cdot a + [b < a] \cdot b$.) Note that the bounds of integration can also be $-\infty$ and ∞ respectively. bounds, where minimum and maximum are again encoded using Iverson brackets.

Once integration bounds have been determined, it suffices to compute an antiderivative for the remaining factors of the integrand, by the fundamental theorem of calculus. The symbolic engine applies a number of standard rules for symbolic integration in order to compute antiderivatives:

- Powers of the integration variable without free integration variable in the exponent (including the special cases $x = x^1$ and $1 = x^0$) are handled by the standard rule

$$\int dx\, x^y = [y + 1 \neq 0] \cdot \frac{x^{y+1}}{y+1} + [y + 1 = 0] \cdot \log|x| + C$$

- Powers that only contain the integration variable in the exponent are handled by rewriting them from x^y to $e^{y \cdot \log|x|}$. If the exponent $z(x) := y \cdot \log|x|$ is a linear function in x, PSI applies the standard rule

$$\int dx e^{z(x)} = \frac{e^{z(x)}}{z'(x)} + C.$$

- The natural logarithm is integrated as

$$\int dx \log(a \cdot x + b) = a^{-1} \cdot (a \cdot x + b) \cdot \log(a \cdot x + b) - x + C.$$

- For $a > 0$, $\int dx\, e^{-a \cdot x^2 + b \cdot x + c} = e^{\frac{b^2}{4a} + c} \frac{1}{\sqrt{a}} \cdot (d/dx)^{-1} [e^{-x^2}] \left(\sqrt{a} \cdot x - \frac{b}{2 \cdot \sqrt{a}}\right) + C$
- The antiderivative of $\log(x)^y / x$ is given by

$$[y + 1 = 0] \cdot \log(|\log(|x|)|) + [y + 1 \neq 0] \cdot \frac{\log(|x|)^{y+1}}{y + 1} + C.$$

- For $\Gamma(a, z) = \int_{\mathbb{R}} dt [z \leqslant t] \cdot t^{a-1} \cdot e^{-t}$ and n a positive integer constant, we have

$$\int dx \log(a \cdot x + b)^n = \frac{(-1)^n}{a} \cdot \Gamma(1 + n, -\log(a \cdot x + b)) + C.$$

- For $a > 0$, $\int dx\, (d/dx)^{-1} [e^{-x^2}](a \cdot x + b) =$
  $\frac{1}{a} (d/dx)^{-1} [e^{-x^2}](a \cdot x + b) \cdot (a \cdot x + b) - e^{-(a \cdot x + b)^2} + C$
- The antiderivative of a Gaussian times its own antiderivative is evaluated via partial integration.
- If the integrand has the form $x^n \cdot f(x)$ for some positive integer n, and the symbolic evaluation engine is able to find an antiderivative $F(x)$ for $f(x)$ as well as for $n \cdot x^{n-1} \cdot F(x)$, the antiderivative for the integrand is computed via partial integration as

$$\int dx\, x^n \cdot f(x) = x^n \cdot F(x) + \int dx\, n \cdot x^{n-1} F(x).$$

  All expressions encountered in this fashion are tracked, and if a linear relation is discovered for an antiderivative, it is computed by solving the corresponding linear equation.

The antiderivatives are then evaluated at the computed bounds. This possibly necessitates evaluating a limit in case one or more of the bounds is infinite. We implemented a number of standard rules to evaluate limits and remove them from the final distribution expression.

EVALUATING LIMITS We implemented a number of simple rules to evaluate limits. For example, for a sum, some summands will have a finite limit, some summands will go to ∞ and some summands will go to $-\infty$. If there are only finite limits of summands or all limits of summands have the same sign, the resulting limit is the sum of the respective limits. To compute a limit, more case splits may become necessary. For example $\lim_{x\to\infty} e^{-a\cdot x^2}$ where a does not depend on x is 1 if a is zero, ∞ if a is negative and 0 if a is positive. The necessary case splits are performed using Iverson brackets.

In case the engine is able to evaluate all necessary limits and they are finite in all cases, the final result of integration is then the product of an Iverson bracket checking that the lower bound is at most the upper bound times the difference of the antiderivative evaluated at the upper and lower bound respectively.

4.4 EVALUATION

This section presents an experimental evaluation of PSI and its effectiveness compared to the state-of-the-art symbolic and approximate inference techniques.

IMPLEMENTATION We implemented PSI using the D programming language. Our representation of distribution expressions relies on hash-consing, such that each unique expression has at most one dynamic representation at a time and expressions can be compared using reference equality checks. PSI can produce resulting query expressions in several formats including Matlab, Maple, and Mathematica. Our system and additional documentation, is available at **http://www.psisolver.org**.

The operands of finite sums and products are represented as sets to avoid redundancy modulo the commutativity and associativity laws of addition and multiplication. We use de Bruin-indices to keep binding expressions unique.

EXECUTION ENVIRONMENT We performed experiments on an 8-core 3.4 GHz Intel I7-6700 CPU with 16 GB RAM.

BENCHMARKS We selected two sets of benchmarks distributed with existing inference engines. Specifically, we used examples from R2 [124] and Fun programs from Infer.NET 2.5 [113]. We use the data sets and queries provided with the original computations. Out of 21 benchmarks, 10 have bounded loops. The loop sizes are usually equal to the sizes of the data sets (up to 784 data points in `DigitRecognition`). Since several benchmarks have data sets that are too large for any of the symbolic tools to successfully analyze, we report the results with truncated data sets. We now briefly describe these benchmarks. R2 benchmarks include:

- **BurglarAlarm**: Finds the probability of burglary, given that the alarm sounded.
- **ClinicalTrial1**: Find if a new medical treatment is effective given the observations of its outcome on the experimental and control groups.

- **CoinBias**: Finds the bias of a biased coin given the toss observations.
- **DigitalRecognition**: Recognizes numerical digits from handwritten notes.
- **Grass**: Finds the probability of raining, given that the grass is wet.
- **HIV**: Estimates the parameters of a linear HIV dynamical model from data.
- **LinearRegression1**: Computes a best fit line given data observations.
- **NoisyOR**: Finds the posterior distribution of a node's value (computed as the noisy-or function of the values of the node's parents) in a directed acyclic graph.
- **SurveyUnbias**: Computes a gender bias for a survey report, models population with a Gaussian distribution.
- **TrueSkill**: Computes the skills of players in a series of games, given the outcomes of these games.
- **TwoCoins**: Finds the marginal distributions, two fair coins, given that not both tosses resulted in heads.

Infer.NET Fun language benchmarks include:

- **AddFun/Max**: Computes a maximum of Gaussian variables.
- **AddFun/Sum**: Computes a sum of Gaussian variables, with a filter.
- **BayesPointMachine**: Training a Bayes point machine.
- **ClickGraph**: Finds the relevance of a web page from the sequence of a user's clicks.
- **ClinicalTrial2**: Find if a new medical treatment is effective given the observations of its outcome on the experimental and control groups. Differs from the R2 version in the parameters and the query.
- **Coins**: Two coins example. The query is the full joint posterior distribution.
- **Evidence/Model1**: Tossing a single coin, with a prior evidence that influences whether the coin is tossed.
- **Evidence/Model2**: Tossing two coins, with a a prior evidence that determines whether two or one coins are tossed.
- **LearningGaussian**: Learning mean and variance of a Gaussian distribution from a set of data points.
- **MurderMystery**: Finds the probability that a person is the murderer given the weapon.

We elided the benchmarks **LDA** and **MixtureOfGaussians**, which use Dirichlet distributions, which we and Hakaru do not currently support.

4.4.1 *Comparison with Exact Symbolic Inference Engines*

EXPERIMENTAL SETUP For comparison with Mathematica 2015 and Maple 2015, we instruct PSI to skip symbolic integration and automatically generate

Table 4.2: Comparison of Exact and Interactive Symbolic Inference Approaches.

Benchmark	Type	Dataset	PSI	Mathem.	Maple	Hakaru
`BurglarAlarm`	D	–	●	●	●	●
`ClinicalTrial1`	DC	100/1000	●	–	–	××
`CoinBias`	DC	5/5	●	t/o	t/o	●^{n}
`DigitRecognition`	D	784/784	●	–	–	×
`Grass`	D	–	●	●	××	●
`HIV`	Co	10/369	◐^{n}	–	–	–
`LinearRegression1`	Co	100/1000	◐$^\int$	–	–	–
`NoisyOr`	D	–	●	●	●	●
`SurveyUnbias`	DC	5/5	●^{n}	t/o	×	●^{n}
`TrueSkill`	C	3/3	◐$^\int$	t/o	t/o	◐^{n}
`TwoCoins`	D	–	●	●	●	–
`AddFun/max`	C	–	●	○	×	○
`AddFun/sum`	C	–	●	●	●	◐^{n}
`BayesPointMachine`	C	6/6	●^{n}	t/o	t/o	◐^{n}
`ClickGraph`	DC	5/5	●	t/o	t/o	●^{n}
`ClinicalTrial2`	DC	5/5	●	t/o	t/o	●^{n}
`Coins`	D	–	●	●	××	●
`Evidence/model1`	D	–	●	○	××	●
`Evidence/model2`	D	–	●	●	××	●
`LearningGaussian`	Co	100/100	◐^{n}	–	–	–
`MurderMystery`	D	–	●	●	●	●

Legend: **Type:** Discrete (D), Continuous (C), Zero-probability observations (o).
Dataset: Full (a/a), no input (–), or the first a out of b inputs (a/b).
Tools: Fully simplified (●) Partially simplified (◐), Not simplified (○),
Not normalized (●^{n}, ◐^{n}), Remaining integrals (◐$^\int$),
Incorrect (××), Crash (×), Timeout (t/o).

distribution expressions in the formats of the two tools. We run Mathematica's `Simplify()` and Maple's `simplify()` commands. For Hakaru [23, 122] (commit `e61cc72009b5cae1dee33bee26daa53c0599f0bc`), we implemented the benchmarks as Hakaru terms in Maple, using the API exposed by the `NewSLO.mpl` simplifier (as recommended by the Hakaru developers). For each benchmark, we set a timeout of 10 minutes and manually compared the results of the tools.

RESULTS Table 4.2 presents the results of symbolic inference. For each benchmark, we present the types of variables it has and whether it has zero-probability observations. We also report the size of the data set provided by the benchmark (if applicable) and the size of the subset we used. For each tool we report the observed

inference result. We mark a result as fully simplified (●) if it does not have any integrals remaining and has a small number of remaining terms. We mark results that have some integral terms remaining ($◐^{\int}$), and partially simplified results (◐). We mark a result as not normalized ($●^n$, $◐^n$) if a tool does not fully simplify the normalization constant. We marked specifically if execution of a tool experienced a crash (×) a timeout (t/o) or a tool produced an incorrect result (××). For five benchmarks, the automatic conversion of PSI's expressions could not produce Mathematica and Maple expressions, because of the complexity of the benchmarks. We marked those entries as '–'. Hakaru's simplifier does not handle zero-probability observations and expectation queries, and therefore we have not encoded these benchmarks (also marked as '–').

PSI PSI was able to fully symbolically evaluate many of the benchmark programs and generate compact symbolic distributions. Running PSI took less than a second for most benchmarks. The most time consuming benchmark was `DigitRecognition`, which PSI analyzed in 37 seconds. For two benchmarks, PSI was not able to remove all integral terms, although it simplified and removed many intermediate integrals.

MATHEMATICA AND MAPLE For several benchmarks, both Mathematica and Maple did not produce a result before the timeout, or returned a non-simplified expression as the result. This indicates that the distribution expressions of these benchmarks are too complex, causing general computer algebra systems to navigate a huge search space. However, we note that these results are obtained for a mechanized translation of programs with the specific encoding we described above. It is possible that a human-driven interactive inference with an alternative encoding may result in more simplified distribution expressions.

Maple crashed for `addFun/max` and `addFun/sum`. We identified that the crashes were caused by an infinite recursion and subsequent stack overflow during simplification. Four benchmarks – `Coins`, `Evidence/model1`, `Evidence/model2`, and `Grass` produce results that are different from those produced by the other tools. For instance, Maple simplifies the density function of `Coins` to 0 (which is incorrect). We attribute this incorrectness to the way Maple integrates Dirac deltas and how it defines Heaviside functions (by default, they are undefined at input 0, but a user can provide a different setting [4]). In our evaluation, none of the alternative settings could yield the correct results. We reported these bugs to the Maple developers. These examples indicate that users should be cautious when using general computer algebra systems to analyze probabilistic programs.

HAKARU For the `ClinicalTrial1` benchmark, Hakaru produced a result differing from PSI's. To get a reference result, we ran R2's simulation to compute an approximate result and found that this result is substantially closer to PSI's. For the `DigitRecognition` benchmark, Hakaru overflowed Maple's stack limit. Hakaru does

Table 4.3: Performance (in s) of Exact and Interactive Symbolic Inference Approaches.

Benchmark	PSI	Mathematica	Maple	Hakaru
BurglarAlarm	0.07	4.57	4.06	0.02
ClinicalTrial1	17.15	–	–	×× 0.54
CoinBias	0.10	t/o	t/o	0.02
DigitRecognition	36.97	–	–	× 6.24
Grass	0.68	7.89	2.00	0.02
HIV	1.14	–	–	–
LinearRegression1	2.93	–	–	–
NoisyOr	0.53	118.89	57.72	0.04
SurveyUnbias	1.22	t/o	× 585.09	0.23
TrueSkill	0.38	t/o	t/o	5.36
TwoCoins	0.04	9.77	0.57	0.01
AddFun/max	0.05	31.66	× 2.02	0.126
AddFun/sum	0.03	9.77	3.89	0.213
BayesPointMachine	1.24	t/o	t/o	41.95
ClickGraph	3.56	t/o	t/o	2.01
ClinicalTrial2	0.87	t/o	t/o	0.72
Coins	0.01	0.58	×× 0.46	0.01
Evidence/model1	0.01	0.55	×× 0.44	0.01
Evidence/model2	0.01	6.40	×× 0.42	0.01
LearningGaussian	15.01	–	–	–
MurderMystery	0.02	0.93	0.51	0.01

not simplify the `AddFun/max` benchmark, but unlike Maple (which it uses), it does not crash.

PERFORMANCE Summed over all examples where Hakaru produced correct but possibly unsimplified results except `BayesPointMachine`, PSI and Hakaru ran for about the same time (8.7s and 8.8s, respectively). `BayesPointMachine` is an outlier, for which Hakaru requires 41.9s, while PSI finds a solution in 1.24s. Mathematica and Maple are 10-300 times slower than PSI.

Table 4.3 presents the detailed timing results for each benchmark and the tool. Columns 2-5 present time in seconds required to compute the output distribution of each benchmark. We specifically marked if a benchmark caused a tool to timeout (t/o), crash (×) or produce a wrong result (××). We ran the timing experiment on an Intel(R) i7 CPU at 3.40GHz, with 16 GB of RAM, running Linux OS.

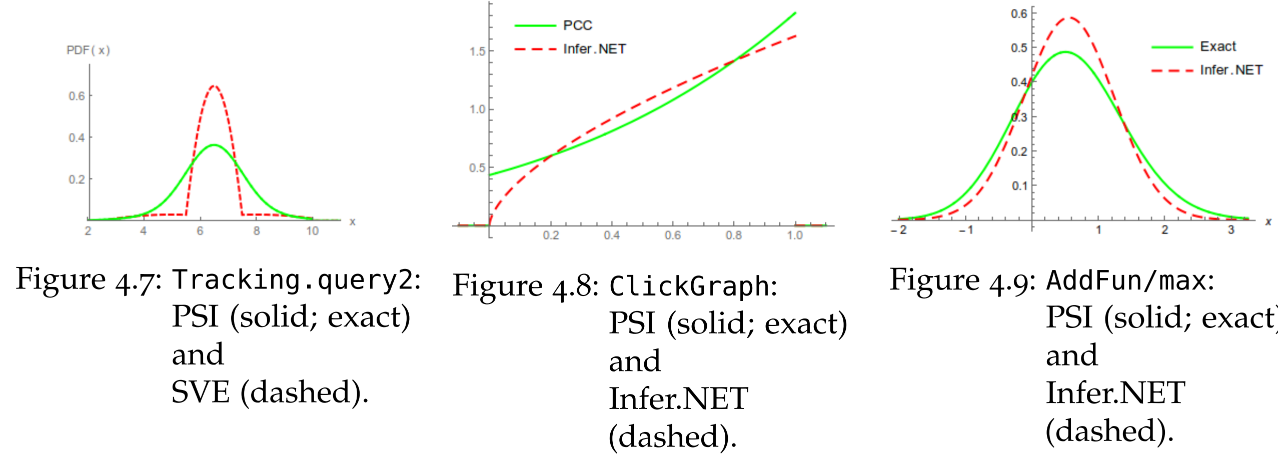


Figure 4.7: `Tracking.query2`: PSI (solid; exact) and SVE (dashed).

Figure 4.8: `ClickGraph`: PSI (solid; exact) and Infer.NET (dashed).

Figure 4.9: `AddFun/max`: PSI (solid; exact) and Infer.NET (dashed).

4.4.2 *Comparison with Approximate Symbolic Inference Engine*

EXPERIMENTAL SETUP We compared PSI with SVE [142] by running posterior distribution queries on the models from the SVE distribution (from the commit `f4cea111f7d489933b36a43c753710bd14ef9f7f`). We included models `tracking` (with 7 provided posterior distribution queries) and `radar` (with 5 posterior distribution queries). We excluded the `competition` model because SVE crashes on it. We did not evaluate SVE on R2 and Infer.NET benchmarks as SVE does not encode some distributions (e.g., Beta or Gamma) and lacks support for Dirac deltas, significantly limiting its ability to represent assignment statements.

RESULTS PSI fully optimized the posterior distributions for all seven queries of the `tracking` model. PSI fully optimized one query from the `radar` benchmark and experienced timeout for the remaining queries. Fig. 4.7 presents the posterior density functions (PDFs) for one of the `tracking` queries. SVE's polynomial approximation yields a less precise shape of the distribution compared to PSI.

4.4.3 *Comparison with Approximate Numeric Inference Engines*

EXPERIMENTAL SETUP We also compared the precision and performance of PSI's exact inference with the approximate inference engines Infer.NET [113] and R2 [124] for a subset of their benchmarks. Specifically, we compared PSI to Infer.NET on `ClickGraph`, `ClinicalTrial`, `AddFun/max`, `AddFun/sum`, and `MurderMystery` and compared PSI to R2 on `BurglarAlarm`, `CoinBias`, `Grass`, `NoisyOR`, and `TwoCoins`. We executed both approximate engines with their default parameters.

RESULTS Infer.NET produces less precise approximate distributions for `ClickGraph` and `AddFun/max` (Figures 4.8 and 4.9), Infer.NET's approximate inference is imprecise in representing the tails of the distributions, although the means of the two distributions are similar (e.g., differing by 0.7% for both benchmarks). PSI and Infer.NET produced identical distributions for the remaining benchmarks. Because of its efficient variational inference algorithms, Infer.NET computed results 5-200

times faster than PSI. The precision loss of R2 on Burglar alarm is 20% (R2's output burglary probability is 0.0036 compared to the exact probability 0.00299). For the other benchmarks, the difference between the results of PSI and R2 is less than 3%. The run times of PSI and R2 were similar, e.g., PSI was two times faster on `TwoCoins`, and R2 was two times faster on `NoisyOR`.

Tables 4.4 and 4.5 present detailed comparison of PSI with R2 and Infer.NET approximate inference engines for a subset of the benchmarks. For each comparison we present columns that represent the name of the benchmark, the results (in the symbolical domain form) that PSI computed for the probabilistic query, the result of the alternative approach (R2 and Infer.NET), and the the inference times of PSI and alternative approaches. We collected inference times of these approaches from tool diagnostics (and removed the time to set up their analysis).

Table 4.4: R2 Benchmark Precision and Analysis Time

Benchmark	Expectation PSI	Expectation R2	Time PSI	Time R2
BurglarAlarm	$2969983/992160802 \approx 0.002993\ldots$	0.0036	70 ms	173 ms
CoinBias	$5/12 = 0.41\bar{6}$	0.417	100 ms	356 ms
Grass	$509/719 \approx 0.7079\ldots$	0.715	680 ms	336 ms
NoisyOR	$130307/160000 \approx 0.8144\ldots$	0.814	530 ms	250 ms
TwoCoins	$(1/3, 1/3)$	$(0.324, 0.336)$	40 ms	108 ms

PRECISION OF EXPECTATION QUERIES For all benchmarks in Table 4.4, PSI produces precise expected values (and does not need to apply approximations). The precision loss of R2 on Burglar alarm is 20%. For the other benchmarks, this difference is smaller than 3%. For TwoCoins, R2 is not able to find equal expectations even though the two outputs have the same marginal distribution. With an exception of NoisyOr, both tools have similar inference times.

PRECISION OF POSTERIOR DISTRIBUTION QUERIES For all benchmarks in Table 4.5, PSI produces precise results PSI was able to compute precise closed-form expressions for all output distributions. For three examples (BurglarAlarm, GaussSum, and MurderMystery) Infer.NET computes the almost the same distributions (with numerical error of distribution parameters less than 0.01%). For ClinicalTrial, Infer.NET produces the distributions of two independent variables. Table 4.5 contains the joint distribution from which these marginals can be immediately derived. Infer.NET produces less precise approximate distributions for ClickGraph and AddFun/max.

Table 4.5: Infer.NET Benchmark Precision and Analysis Time.

Benchmark	Distribution PSI	Distribution Infer.NET
BurglarAlarm	$(2969983 \cdot \delta(1-b) + 989190819 \cdot \delta(b))/992160802$	Bernoulli(0.002995)
ClickGraph	$\frac{6(s+3)^5}{3367} \cdot [0 \leqslant s] \cdot [s \leqslant 1]$	Beta(1.625, 1)
ClinicalTrial	$900/102 \cdot c \cdot (1-c)^4 \cdot t^4 \cdot (1-t) \cdot$ $(77 \cdot \delta(1-e) + 25 \cdot \delta(e))$	(Beta(5, 2), Beta(2, 5)) Bernoulli(0.755))
AddFun/max	$\sqrt{2}/\pi \cdot (d/dx)^{-1}[e^{-x^2}](r/\sqrt{2}) \cdot e^{-r^2/2}$	Gauss(0.56, 0.68)
AddFun/sum	$(2 \cdot \sqrt{\pi})^{-1} \cdot e^{-r^2/4}$	Gauss(0, 2)
MurderMystery	$(9 \cdot \delta(1-p) + 560 \cdot \delta(p))/569$	Bernoulli(0.0158)

Benchmark	Time PSI	Time Infer.NET
BurglarAlarm	70 ms	14 ms
ClickGraph	3.56 s	20 ms
ClinicalTrial	870 ms	21 ms
AddFun/max	50 ms	4 ms
AddFun/sum	30 ms	4 ms
MurderMystery	20 ms	1 ms

The examples in Figures 4.7, 4.8, and 4.9 illustrate that the choice of inference method depends on the context in which the inference results are used. While inferences about expectations in machine learning applications may often tolerate imprecision in return for faster or more scalable computation, many uses of probabilistic inference in domains such as security, privacy, and reliability engineering need to reason about a richer set of queries, while requiring correct and precise inference. We believe that the PSI system is particularly suited for such settings and is an important step forward in making automated exact inference feasible.

4.5 RELATED WORK

This section discusses related work in symbolic inference and probabilistic program analysis.

4.5.1 *Symbolic Inference*

GRAPHICAL MODELS Early research in the machine learning community focused on symbolic inference in Bayesian networks with discrete distributions [146] and combinations of discrete and linearly-dependent Gaussian distributions [27]. For more complex hybrid models, researchers proposed projecting distributions to mixtures of base functions, which can be easily integrated, such as truncated expo-

nentials [118] and piecewise polynomials [142, 150]. In contrast to these approximate approaches, PSI's algorithm performs exact symbolic integration.

PROBABILISTIC PROGRAMS Claret et al. [30] present a data flow analysis for symbolically computing exact posterior distributions for programs with discrete variables. This analysis operates on the program's concrete state, while efficiently storing the states using algebraic decision diagrams (ADDs).

Bhat et al. [15] present a type system for programs with continuous probability distributions. This approach is extended in [16] to programs with discrete and continuous variables (but only discrete observations). Like PSI, the density compiler from [16] computes posterior distribution expressions, but instead of symbolically simplifying and removing integrals, it generates a C program that performs numerical integration (which may, in general, be expensive to run).

We note that for numerical integration, a user needs to provide *all* concrete values of the parameters ahead of time. Moreover, multivariate numerical integration (which is typically based on MC sampling) can be often be prohibitively expensive and produce non-deterministic results.

In contrast, our approach computes all, potentially multivariate, integrals symbolically and provides a symbolic posterior distribution, which contains only elementary functions.

The Hakaru probabilistic language [23, 122] runs inference tasks by combining symbolic and sampling-based methods. To optimize MCMC sampling for probabilistic programs, Hakaru's symbolic optimizer (1) translates the programs to probability density expressions in Maple's language, (2) calls an extended version of Maple's simplifier, (3) uses these results to generate an optimized Hakaru program, and, if necessary, (4) calls a MCMC sampler with the optimized program.

4.5.2 *Probabilistic Program Analysis*

VERIFICATION Researchers presented various static analyses that verify probabilistic properties of programs, including safety, liveness, and/or expectation queries. These verification techniques have been based on abstract interpretation [37, 40, 111, 116], axiomatic reasoning [12, 86, 119], model checking [77], and symbolic execution [48, 141]. Many of the existing approaches compute exact probabilities of failure only for discrete distributions or make approximations when analyzing computations with both discrete and continuous distributions.

Researchers have also formalized fragments of probability theory inside general-purpose theorem provers, including reasoning about discrete [10, 84] and continuous distributions [45, 72]. The focus of these works is on human-guided interactive verification of (possibly recursive) programs. In contrast, PSI performs fully automated inference of hybrid discrete and continuous distributions for programs with bounded loops.

TRANSFORMATION R2 [124] transforms probabilistic programs by moving observe statements next to the sampling statement of the corresponding variable to improve performance of MCMC samplers. Gretz et al. [68] generalize this transformation to move observations arbitrarily through a program. Probabilistic program slicing [83] removes statements that are not necessary for computing a user-provided query. These transformations simplify program structure, while preserving semantics. In comparison, PSI directly transforms and simplifies the probability distribution that underlies a probabilistic program.

4.6 DISCUSSION

In this chapter, we presented PSI, an approach for end-to-end exact symbolic analysis of probabilistic programs with discrete and continuous variables. PSI's symbolic nature provides the necessary flexibility to answer various queries for non-trivial probabilistic programs. More precise and reliable probabilistic inference has the potential to improve the quality of the results in various application domains and help developers when testing and debugging their probabilistic models and inference algorithms. With its rich symbolic domain and optimization engine, we believe that PSI is a useful tool for studying the design of precise and scalable probabilistic inference based on symbolic reasoning.

5

λPSI: EXACT INFERENCE FOR HIGHER-ORDER PROBABILISTIC PROGRAMS

Recent PPS that support exact symbolic reasoning in the presence of continuous distributions (Hakaru [122] and PSI from Chapter 4) lack some useful language features. For example, PSI does not support first-class functions. While Hakaru's implementation supports first-class functions (including distributions over functions), its exact inference operators, namely normalization and disintegration (with respect to the Lebesgue measure) are external programs that manipulate Hakaru terms and are not available as first-class operators within these terms. In contrast, there are *higher-order* PPS which do support first-class inference (e.g., Church [160], WebPPL [63] and Anglican [32]), however, their exact inference algorithms only handle discrete distributions. A key challenge then is to provide support for both first-class inference *and* the ability to compute the exact posterior over discrete, continuous and mixed variables.

THIS CHAPTER We present λPSI, the first PPS which addresses the above challenge.

First, we introduce the statically typed higher-order probabilistic programming language (PPL) λPSI, which is based on the PSI PPL [49] but with additional support for tuples, arrays, higher-order functions, and nested inference. As demonstrated in PPLs such as Church [65], WebPPL [63], Anglican [172] and Venture [109], higher-order constructs are useful for specifying models where inference queries are nested within other inference queries. This enables, for instance, an inference to be made about agents that themselves make use of models and (incomplete) data so to infer knowledge about the state of the world. Unlike other higher-order PPLs (see above), which are dynamically typed, static typing enables easier debugging, better error messages, and avoids expensive dynamic checks during inference.

Second, we introduce an exact inference solver to handle these language features while supporting mixed, discrete, and continuous variables. λPSI's engine further explicitly computes the probability of error, while existing PPS crash stochastically at run time (e.g., randomly indexing an array may or may not cause an out-of-range error during sampling-based inference). We believe λPSI is the first to support exact inference for higher-order probabilistic programs with this level of expressiveness.

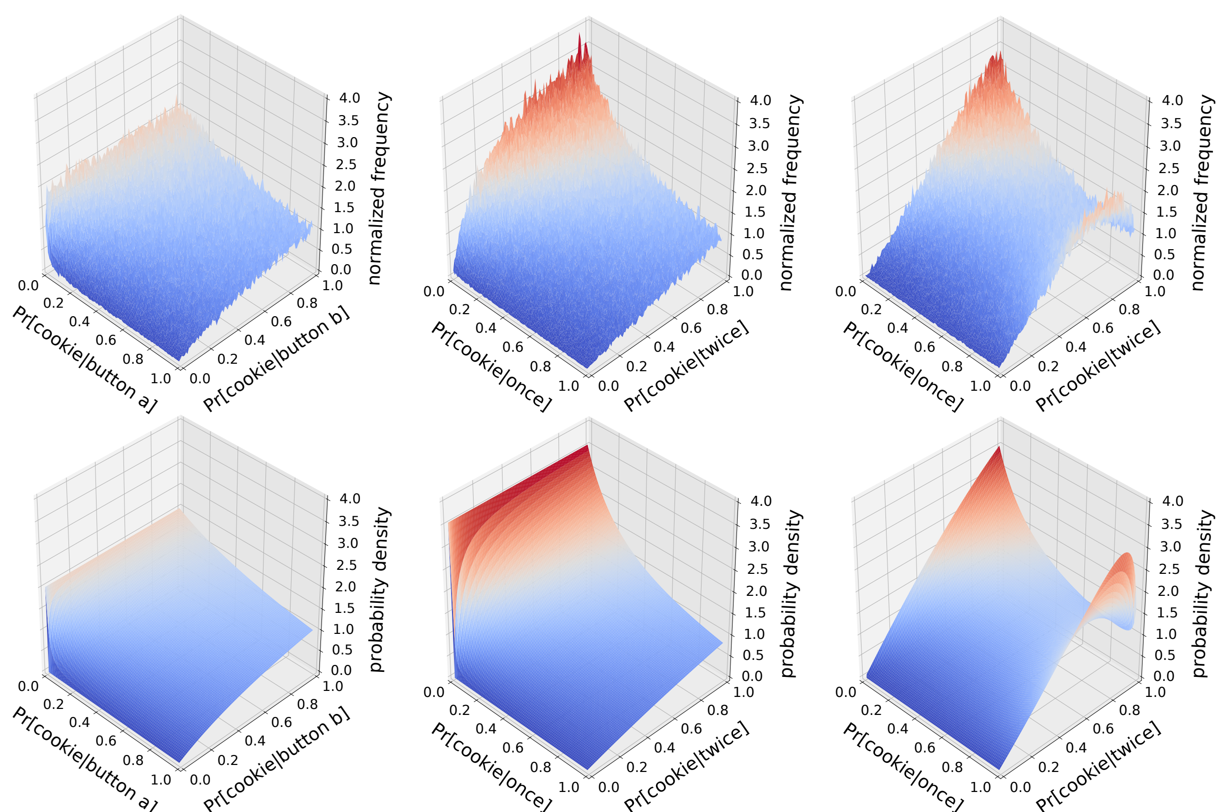


Figure 5.1: Inference on “Epistemic States”: approximate inference in Webchurch (top) vs. exact inference in λPSI (bottom).

Finally, we show λPSI is powerful enough to specify a number of interesting problems ranging from information theory to rational agents, and that its solver can compute, for the first time, the exact posterior for many applications that so far could only be handled approximately.

MAIN CONTRIBUTIONS Our key contributions are:

- The λPSI statically typed higher-order PPL which supports higher-order functions and nested inference (§5.2).
- The λPSI solver which performs exact symbolic inference and computes the posterior distribution over discrete, continuous, and mixed random variables (§5.3–§5.5).
- An extensive evaluation of higher-order exact inference with λPSI across various applications (§5.6).

The work presented in this chapter has been published in Gehr et al. [53].

5.1 MOTIVATION AND OVERVIEW

We now provide a motivating example for nested inference, followed by an overview of λPSI.

NESTED INFERENCE EXAMPLE To reason about rational agent behavior, we can build probabilistic models where multiple rational agents interact and each has a model about how the other agents model the interaction. Such models can be easily built in languages where probabilistic inference is a first-class expression which is allowed to occur inside another probabilistic inference query. For instance, Goodman and Tenenbaum [64] describe a number of (Church/WebPPL) models of this kind in Chapter 15 ("Social cognition"). In order to evaluate our exact inference approach, we have specified all of those models in λPSI (see also §5.6).

To illustrate the results of exact inference, let's consider some examples of section "Epistemic States" of that chapter. These examples model an observer of a rational agent operating a vending machine that probabilistically yields either a cookie or a bagel, depending on which one of two buttons a or b is pressed. In the first model (Fig. 5.1, left), the agent is observed to press button b. The observer assumes a uniform prior over the agent's actions and knows that the agent's goal is to obtain a cookie. The result is the posterior on the probability that the machine yields a cookie when pressing a given button. In the second and third model (Fig. 5.1, middle and right), the vending machine has only one button a, which may be pressed multiple times. The prior belief over the agent's actions is biased towards pressing the button fewer times, and the agent is observed to press button a twice. While the observer knows that the agent's goal is to obtain a cookie in the second model, the goal is unknown in the third model. Such models are interesting because they involve a mixture of continuous and discrete distributions as well as nested inference queries. We show the results comparing approximate vs. exact joint posteriors for these examples in Fig. 5.1. The plots in the top row are normalized histograms of 10^6 samples each with a resolution of 100×100, computed by the Church implementation "Webchurch". The bottom row shows plots of the exact posteriors computed by λPSI. We note that our engine evaluates all posteriors within a few seconds, while random sampling takes up to 10 minutes. To the best of our knowledge, this is the first time that those posterior distributions have been evaluated to this precision. We discuss other interesting applications in §5.6.

λPSI LANGUAGE AND INFERENCE The λPSI program in Fig. 5.2 illustrates some of λPSI's core language features. Fig. 5.2 also visualizes the exact inference result computed by λPSI.

First, the program creates a tuple `a` of two random real numbers. One of them is drawn from a continuous uniform distribution, whereas the other is drawn from a discrete uniform distribution. In addition to tuples, λPSI supports arrays of both fixed and random length.

Next, variable `x` is initialized to a random entry of the tuple. The subsequent assignment stores the result of a nested inference query in the variable `p` of type `Distribution[ℝ]`. The `infer` expression accepts an (anonymous) function representing the query, which uses a uniform prior for the variable `y`. This variable is conditioned on the observed evidence `y <= x` to produce the nested posterior. Note that the

function we pass to `infer` is itself random, as it depends on the external random variable `x`. `infer` itself is a deterministic function without side effects (in particular, the nested inference query does not influence our knowledge of `x`), but because the input is random, the returned distribution `p` is also random.

Finally, we return the expectation of `p` and a value drawn from `p`, instructing λPSI to compute a joint probability distribution for those two values. As `p` is random, so is its expectation. Therefore, the program produces a joint distribution of two dependent real random variables.

The system computes this distribution by combining symbolic expressions for subprograms and then simplifying them. For example, the joint distribution of variables `a` and `x` is represented by the following symbolic expression in λPSI:

$$\int dv \int dz\ \overbrace{\tfrac{1}{2}[0 \leqslant v] \cdot [v \leqslant 2] \cdot \lambda[\![v]\!]}^{(a)} \cdot \overbrace{\left(\tfrac{1}{3} \textstyle\sum_{k=1}^{3} \delta(k/3)[\![z]\!]\right)}^{(b)} \cdot \underbrace{\delta(v,z)[\![a]\!]}_{(c)} \cdot \underbrace{\tfrac{1}{2}\left(\delta(a_0)[\![x]\!] + \delta(a_1)[\![x]\!]\right)}_{(d)}.$$

The expression uses integrals to marginalize the temporary values v and z of the first, resp. second entry of `a`. Part (a) represents the uniform distribution on the interval $[0, 2]$. Here, $\lambda[\![v]\!]$ represents the Lebesgue measure, which is a continuous measure with density 1 at each real number. Part (b) uses Dirac deltas of the form $\delta(e)[\![z]\!]$, which can be interpreted as point-mass distributions on e for z, to represent the discrete uniform distribution on $\{\frac{1}{3}, \frac{2}{3}, 1\}$. Part (c) assigns the tuple (v, z) to a, and part (d) assigns the first or second entry of a to x, each with probability $\frac{1}{2}$. At this point, it is sufficient for the reader to understand the basic ideas. In §5.3, we provide all details required to understand this expression in depth.

The above is an example of an intermediate result computed during the symbolic analysis λPSI performs. A plot of the cumulative distribution function of the final result computed by λPSI is shown in Fig. 5.2. The full symbolic expression for this final result computed by λPSI (ignoring errors) is given by

$$\Lambda x, y.\ [0 \leqslant y] \cdot [y \leqslant 1] \cdot \lambda[\![y]\!] \cdot \Big(\tfrac{1}{2} \cdot [y \leqslant \tfrac{1}{3}] \cdot \delta\big(\tfrac{1}{6}\big)[\![x]\!] + \tfrac{1}{4} \cdot [0 \leqslant x] \cdot [x < \tfrac{1}{2}] \cdot [y \leqslant 2x] \cdot \tfrac{1}{x} \cdot \lambda[\![x]\!] + \tfrac{1}{4} \cdot [y \leqslant \tfrac{2}{3}] \cdot \delta\big(\tfrac{1}{3}\big)[\![x]\!] + \tfrac{5}{12} \cdot \delta\big(\tfrac{1}{2}\big)[\![x]\!]\Big).$$

5.2 THE λPSI PPL

We next describe the higher-order probabilistic programming language λPSI, which extends the PSI language by (i) higher-order functions, (ii) a first-class probabilistic inference operator, (iii) conditioning on probability-zero events, and (iv) a dependent static type system. Fig. 5.3 presents a simplified core syntax of λPSI, which is

```
1  def main(){
2   a := (uniform(0,2),
3     uniformInt(1,3)/3);
4   x := a[flip(1/2)];
5   p := infer((){
6     y := uniform(0,1);
7     observe(y <= x);
8     return y;
9   });
10  return (expectation(p),
11          sample(p));
12 }
```

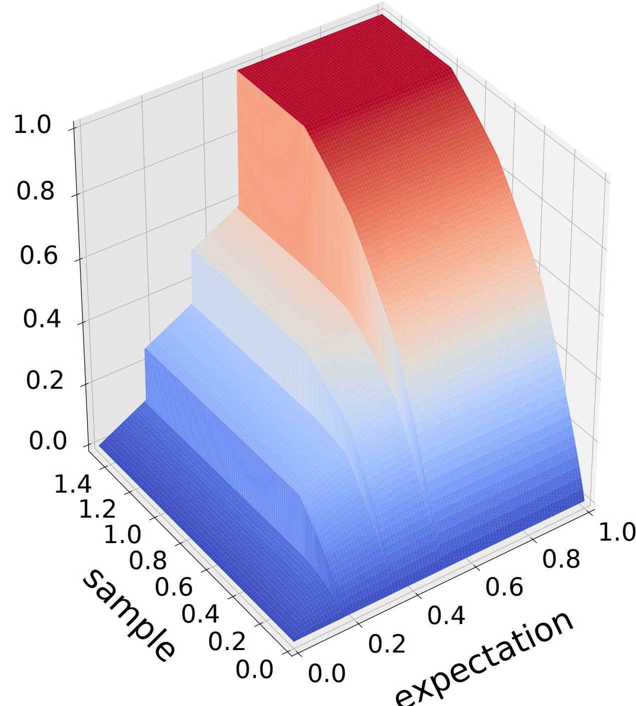


Figure 5.2: A λPSI program (left) and its exact joint probability distribution (right) computed by λPSI.

```
Func   ::=  def x (x: Type, ... ,x: Type) Body
Body   ::=  { Stmt* } | ⇒ Ex;
Ex     ::=  n | x | BuiltIn | uop Ex | Ex bop Ex | (Ex, ... ,Ex) | Ex[Ex] |
            (x: Type, ... ,x: Type) Body | Ex(Ex)
Stmt   ::=  x := Ex; | Ex = Ex; | return Ex; |
            observe(Ex); | cobserve(Ex,Ex); |
            if Ex { Stmt* } else { Stmt* } | for x in [n..n){ Stmt* }
Type   ::=  Type × ··· × Type | Type → Type | Distribution[Type] |
            ℕ | ℤ | ℝ | ···

BuiltIn ∈ {Flip, Gauss, Uniform, UniformInt, Categorical, Exponential, ...}
        ∪ {infer, sample, expectation}
        ∪ {exp, log, ...}
```

Figure 5.3: Core syntax of λPSI (n, x, uop, and bop denote constants, variables, unary, and binary operations, respectively).

sufficiently expressive to highlight the key insights of the work presented in this chapter.

For reference, we present the syntax of the full language in Fig. 5.4.

TYPES Our language features a static type system. In addition to standard numeral, tuple, and function types, λPSI supports dedicated distribution types. For example, `Distribution[ℝ]` describes distributions over a real variable.

Prog	::=	Decl*
Decl	::=	Func | VarDecl
Func	::=	**def** x ParamList* (:Ex)? Body
ParamList	::=	(Param* (,)?) | [Param* (,)?]
Param	::=	x: Ex
Body	::=	{ Stmt* } | ⇒ Ex;
Type	::=	x | * | $\mathbb{B}$ | $\mathbb{N}$ | $\mathbb{Z}$ | $\mathbb{Q}$ | $\mathbb{R}$ | $\mathbb{1}$ | $\mathrm{Ex}^{\mathrm{Ex}}$ | (Ex × ··· × Ex) | Ex[] | (Ex → Ex) | ($\prod_{x:\,\mathrm{Ex}}$ Ex[x])
VarDecl	::=	x := Ex;
Lambda	::=	ParamList* LambdaBody
LambdaBody	::=	{ Stmt* } | ⇒ Ex
Ex	::=	n | x | BuiltIn | (Ex) | (Ex: Ex) | (*uop* Ex) | (Ex *bop* Ex) | () | (Ex,) | (Ex,Ex(,Ex)*(,)?) | [] | [Ex(,Ex)*(,)?] | Ex.length | Ex[Ex] | Ex(Ex) | Lambda | Type
AssgnLhs	::=	x | AssgnLhs[Ex] | (AssgnLhs: Ex) (AssgnLhs, ..., AssgnLhs(,)?)
Stmt	::=	AssgnLhs = Ex; | Decl | **return** Ex; | **observe**(Ex); | **cobserve**(Ex,Ex); | **if** Ex { Stmt* } (**else** { Stmt* })? | **for** x **in** [n..n){ Stmt* } | **assert**(Ex);

n	∈	{0,1,2,3,...}
x	∈	Vars
bop	∈	{+, -, *, %, /, ^, ~, &&, ||, ==, !=, <, >, <=, >=}
uop	∈	{+, -, !}

BuiltIn ∈ {Distribution, infer, sample, expectation, array, π, exp, log, floor, ceil, ...}
∪ {Flip, Gauss, Uniform, UniformInt, Categorical, Exponential, Dirac, ...}

Figure 5.4: Full syntax of the higher-order probabilistic programming language λPSI (extension of Fig. 5.3).

We support dependent types $\mathrm{Ex}^{\mathrm{Ex}}$ for fixed-length arrays, which are compatible to tuple types (e.g., $\mathbb{Z}^2$ is equivalent to $\mathbb{Z} \times \mathbb{Z}$). Subtype relations are standard (e.g., $\mathbb{N} \subseteq \mathbb{Q}$ and $\mathbb{R} \to \mathbb{N} \subseteq \mathbb{Z} \to \mathbb{Q}$). We do not distinguish types and other expressions in our grammar. Types are compared modulo partial evaluation of numerical expressions.

PROGRAMS AND FUNCTIONS A λPSI program consists of a sequence of function declarations, whose bodies can be single expressions, such as in the function

```
def succ(x:ℝ)⇒x+1;
```

Alternatively, function bodies can be sequences of imperative statements, such as in the function

```
def succ(x:ℝ){
    y:=x;
    y=y+1;
    return y;
}
```

The `main` function, which may take parameters, forms the entry-point of a λPSI program.

Any function in λPSI can have multiple parameter lists, which defines a curried function. E.g., `def const(x:ℝ)(_:ℝ) ⇒ x` is shorthand for `def const(x:ℝ) ⇒ (_:ℝ) ⇒ x`. It is possible to optionally specify a return type: `def id(x:ℝ):ℝ ⇒ x`.

Parameter lists can also be declared with square brackets, usually used for dependent types and polymorphic functions: The function `def foo[n:ℕ](x:ℝ^n):ℝ^(2·n) ⇒ x~x;` concatenates a fixed-length array `x` with itself to yield an array of double length. Further, consider the following identity function: `def id[a:*](x:a) ⇒ x;`. The type of this function is $\prod_{x:*}(x \to x)$, which can be read as "for any type x, this provides a function from x to x". Square-bracket parameters can be provided explicitly at the call site, as for example in `id[ℝ[]]([1,2,3])`, or inferred automatically from a regular function call, which will happen twice when type-checking the example expression `id(id)([1,2,3])`.

EXPRESSIONS Our language supports standard unary and binary operations on boolean and numeric types. Also, it supports tuples with the usual syntax. For example, the expression `(3,4)` is a two-element tuple, whose first entry `3` can be accessed by `(3,4)[0]`.

In addition to tuples, λPSI supports arrays. The expression `()` (resp. `[]`) is the empty tuple (resp. array) and `(Ex,)` is a single-element tuple. Subexpressions may be annotated with their types, e.g., `((1:ℕ)+(2:ℕ):ℕ,[-1,2]):ℕ × ℤ[]`.

λPSI supports multiple built-in expressions. These include constructors for built-in distributions, such as `Flip` and `Gauss`, whose lower-case variants (i.e., `flip` and `gauss`) draw a sample from the respective distribution. For example, `Flip(1/2)` is the uniform distribution on $\{0, 1\}$ (whereas `flip(1/2)` is a sample), and `Gauss(0,1)` is the standard normal distribution parameterized by mean and variance. We can sample from an expression `d` representing a distribution via the expression `sample(d)`. For example, the expression `flip(1/2)` is equivalent to `sample(Flip(1/2))`. Similarly, `expectation(d)` computes the expected value of a random variable drawn from `d`.

Finally, λPSI supports lambda expressions denoting anonymous functions, for example `(x:ℝ){ y:=x; y=y+1; return y; }`. The syntax for function application is standard (e.g., `succ(1)`).

$$
\begin{aligned}
e ::= \;& x \mid q \mid e \mid \pi \mid -e \mid e_1 + \ldots + e_n \mid e_1 \cdot \ldots \cdot e_n \mid e_1^{e_2} \mid \log(e) \mid \lfloor e \rfloor \mid \lceil e \rceil \mid \\
& [e_1 = e_2] \mid [e_1 \leqslant e_2] \mid [e_1 \neq e_2] \mid [e_1 < e_2] \mid \textstyle\sum_{x \in \mathbb{Z}} e[x] \mid \int dx\, e[x] \mid (d/dx)^{-1}[e^{-x^2}] \mid \\
& (e_1, \ldots, e_k) \mid [x \mapsto e[x]](e) \mid e_{1e_2} \mid e_{1e_2 \mapsto e_3} \mid \{f_1 \mapsto e_1, \ldots, f_k \mapsto e_k\} \mid e.f \mid e_1\{f \mapsto e_2\} \mid \\
& \lambda x.\, e[x] \mid e_1(e_2) \mid \Lambda x.e[x] \mid e[\![x]\!] \mid \delta(e)[\![x]\!] \mid \lambda[\![x]\!] \mid e_1 /\!\!/ e_2 \mid \bot \mid e_{1?}(e_2)
\end{aligned}
$$

Figure 5.5: Symbolic domain for expressing probability distributions. We write $e[x]$ to denote that x is a free variable in e. The highlighted elements are fundamental to λPSI and new compared to PSI.

STATEMENTS Our language distinguishes variable declarations (e.g., `x:=3`) and assignments (e.g., `x=3`). The `observe` statement conditions the random program state on (positive-probability) observed evidence: all program states that do not satisfy the condition are discarded. The result of inference on the final program is given by renormalizing the resulting subprobability distribution at program exit. For example, the statement `observe(x>=2);` conditions the distribution on program states on the observed fact that `x` is at least `2`.

The `cobserve` ("continuous observation") statement is used to condition on a possible, but probability-zero event. In particular, `cobserve(x,y);` conditions on the probability-zero event that `x` is equal to `y`. We note that conditioning on such events is a delicate matter and hence requires its own statement in the language. See Shan and Ramsey [148] for an in-depth discussion of the involved issues.

Finally, λPSI supports standard `if` statements, and `for` loops with statically-known bounds.

FIRST-CLASS INFERENCE The built-in `infer` function enables us to perform nested inference. It reifies a function `f` with return type a to a `Distribution[a]`, for any type a. If `f` does not execute `observe` statements, `infer` can be thought of as the inverse of sample. This is because `infer(() ⇒ sample(d))` returns the distribution `d` and `() ⇒ sample(infer(f))` yields the function `f`. Otherwise, `infer` is more interesting (see Fig. 5.2): it forms a context within which the observations evaluated by `f` (see Line 7) take effect and returns the normalized posterior of `f` given that evidence and all state outside the context. The computation of `infer` has no side effects, meaning that the observations do not affect the knowledge outside the query (e.g., about `x`).

5.3 A SYMBOLIC DOMAIN FOR DISTRIBUTIONS

We now introduce a symbolic domain for probability distributions. When performing exact inference, λPSI simplifies representations of distributions in this domain. In §5.4, we will see how any λPSI program is translated to this domain.

5.3.1 *Representation*

λPSI's symbolic domain for probability distributions is shown in Fig. 5.5. This grammar extends the symbolic domain used in PSI by representations of data structures (highlighted in second line of Fig. 5.5) as well as higher-order functions and distributions (highlighted in third line).

BASIC ARITHMETIC EXPRESSIONS Basic expressions (first line in Fig. 5.5) are inherited from PSI, including variables (x), rational constants (q), the irrational constants e and π, as well as standard arithmetic expressions including the floor ($\lfloor e \rfloor$) and ceiling ($\lceil e \rceil$) operators. The Iverson bracket $[P]$ is an indicator for the proposition P with the usual convention [89]. Like in PSI, we write divisions a/b as $a \cdot b^{-1}$.

DATA STRUCTURES Our symbolic domain can directly represent data structures of λPSI's programming language (second line in Fig. 5.5). In particular, it supports tuples, arrays, and records (the latter are used for program states, see §5.4). For example, $(1, 2)$ is a tuple and $\{\mathtt{f} \mapsto 1\}$ is a record with one field $\mathtt{f}$, which has value 1. We represent arrays as mappings from indices to values together with their lengths. For example, the identity permutation of length 5 is represented as $[x \mapsto x](5)$. The i-th value in an array or tuple a is denoted by a_i. The expression $a.f$ is the value of field f in record a. The expression $a_{i \mapsto b}$ (resp. $a\{\mathtt{f} \mapsto b\}$) represents a modification of a tuple or array (resp. record) a where the i-th value (resp. the value of field $\mathtt{f}$) is replaced by b.

DISTRIBUTIONS AND HIGHER-ORDER FUNCTIONS The third line in Fig. 5.5 shows the most interesting expressions, which are particular to representing probability distributions and first class functions. The domain contains sums over $\mathbb{Z}$ (we write $e[x]$ to denote that x is a free variable in e) as well as integrals, which will be discussed in detail in §5.3.2. The domain of an integral is implicitly defined by the variable x being integrated over. The expression $(d/dx)^{-1}[e^{-x^2}]$ denotes the antiderivative of the function e^{-x^2}, which does not have a closed-form solution but is useful to for example express the cumulative distribution function of a normal distribution.

To support higher-order functions and nested inference, our domain contains lambdas in two flavours: functions ($\lambda x.e[x]$) and distributions ($\mathbb{\Lambda} x.e[x]$). For example, $\lambda x.f(x)$ is the same as the function f, while $\mathbb{\Lambda} x.p[\![x]\!]$ is the same as the distribution p. Note that unlike for function application $e_1(e_2)$, the argument x for distribution application $e[\![x]\!]$ must be a variable. We discuss distributions in more detail in §5.3.2.

As we will exemplify in §5.3.3, all distributions in λPSI are built from two primitive distributions. The *Dirac delta* $\delta(e)[\![x]\!]$ expresses that variable x is distributed according to the point-mass distribution on e, where x cannot occur freely in e. For

example, $\delta(0)[\![x]\!]$ is the point-mass distribution on 0. Dirac deltas are used to express discrete distributions. The *Lebesgue measure* $\lambda[\![x]\!]$ is used to construct a continuous distribution from a probability density function (discussed in §5.3.3). The operator $/\!\!/$ denotes *disintegration*, which we will discuss in more detail in §5.5.5. It can be loosely thought of as a special kind of division allowing us to eliminate Lebesgue measures by the equivalence $\lambda[\![x]\!] /\!\!/ \lambda[\![x]\!] = 1$.

ERRORS The expression $\perp$ denotes a special error value, and $\delta(\perp)[\![x]\!]$ is used to capture the probability of an error. We use $e_{1?}(e_2)$ to propagate errors upon composition, i.e.$e_{1?}(e_2)$ is equal to $e_1(e_2)$ for $e_2 \neq \perp$, while $e_{1?}(\perp)$ reduces to $\delta(\perp)$.

5.3.2 *Interpretation*

Before continuing our discussion, we provide an interpretation of the more advanced symbolic expressions.

DISTRIBUTIONS A key concept of λPSI are *distributions*, which can be loosely thought of as unnormalized probability densities. Formally, a distribution f over a λPSI type τ (e.g., $\mathbb{R}$) is a bounded measure on τ (it is not necessarily normalized) and we write $D[\tau]$ to denote the set of all distributions over τ. We write $\Lambda x.\ f[\![x]\!]$ to clarify that f is a distribution for the variable x. Consider the expression $\lambda a.\ \Lambda b.\ f(a)[\![b]\!]$, which takes a as input and returns a probability distribution for b. Here, f can be thought of as taking a as input and returning a probabilistic value for b.

INTEGRALS A distribution $\Lambda x. f[\![x]\!]$ over τ can be formally interpreted as a random variable: for any $S \subseteq \tau$, it is

$$\Pr[f \in S] \propto f(S) = \int_\tau \mathbf{1}_S \, df$$

where the integral is the Lebesgue integral (recall that f is a measure). The probability is proportional due to the missing normalization.

λPSI's symbolic domain (see Fig. 5.5) uses a convenient (non-standard) notation for integrals: for any types τ, τ', distribution $f \in D[\tau]$, and function $g \in \tau \to \tau'$, it is

$$\int dx\ g(x) f[\![x]\!] \quad \text{defined as} \quad \int_\tau g\ df.$$

The domain of the integral is determined by the type τ of x. Note how the Riemann-style notation (dx) makes dependencies explicit: The "output" of f is used as an input to g. Notwithstanding the above definition, it often suffices to think about integrals in the common Riemann sense.

INTEGRATING HIGHER-ORDER DISTRIBUTIONS Our notation allows conveniently expressing integrals involving higher-order distributions. For $f \in D[D[\tau]]$ being a distribution over distributions over some type τ, we can e.g. write

$$\Lambda r.\ h[\![r]\!] = \Lambda r. \int dx\ x[\![r]\!] \cdot f[\![x]\!]\,.$$

Here, the integration variable x and the result h are first-order distributions. The interpretation is that for any $S \subseteq \tau$: $\Pr[h \in S] \propto \int dx\ x(S) \cdot f[\![x]\!]$.

DIRAC DELTA For any value v, the Dirac delta $\delta(v)$ is a measure capturing the point mass on v. Formally, for any type τ, $\delta\colon \tau \to D[\tau]$ is defined as

$$\forall v \in \tau,\ S \subseteq \tau. \quad \delta(v)(S) = [v \in S].$$

The expression $\Lambda x.\ \delta(v)[\![x]\!]$ denotes that x is distributed according to a point mass on v. We can loosely think of $\delta(v)[\![x]\!]$ being 0 for all $x \neq v$ and ∞ for $x = v$. The Dirac delta is normalized: $\int dx\ \delta(v)[\![x]\!] = 1$.

5.3.3 *Examples*

The Dirac delta is used to represent discrete probability distributions. For example, the Bernoulli distribution with success probability $\frac{1}{3}$ (i.e., `flip(1/3)`) can be written as

$$\Lambda x.\ \text{Bernoulli}(\tfrac{1}{3})[\![x]\!] := \Lambda x.\ \tfrac{2}{3}\delta(0)[\![x]\!] + \tfrac{1}{3}\delta(1)[\![x]\!]\,.$$

Note that as expected, the probability of value 1 is

$$\int dx\ [x = 1] \cdot \text{Bernoulli}(\tfrac{1}{3})[\![x]\!] = \tfrac{1}{3}.$$

Discrete distributions with infinite support can be represented using expressions of the form $\sum_{x \in \mathbb{Z}} e[x]$. For instance, `geometric(1/4)`, the geometric distribution with success probability $\frac{1}{4}$, can be written as

$$\Lambda x.\ \textstyle\sum_{i \in \mathbb{Z}} [i \geqslant 0] \cdot \left(\tfrac{3}{4}\right)^{i} \cdot \tfrac{1}{4} \cdot \delta(i)[\![x]\!]\,. \tag{5.1}$$

Intuitively, the Lebesgue measure $\lambda[\![x]\!]$ assigns uniform weight to all values. It can be used to define continuous distributions: the expression $\Lambda x.\ p(x) \cdot \lambda[\![x]\!]$ denotes the distribution of a continuous random variable with probability density function p. For example, the exponential distribution with rate 2 (i.e., `exponential(2)`) can be written as

$$\Lambda x.\ [0 \leqslant x] \cdot 2e^{-2x} \cdot \lambda[\![x]\!]\,.$$

Variable read	`x`	=	$\lambda\sigma.\ \Lambda r.\ \delta(\sigma.x)\llbracket r\rrbracket$	(5.5)
Binary operation	`a + b`	=	$\lambda\sigma.\ \Lambda x.\ \int dy \int dz\ \boxed{\texttt{a}}(\sigma)\llbracket y\rrbracket \cdot \boxed{\texttt{b}}(\sigma)\llbracket z\rrbracket \cdot \delta(y+z)\llbracket x\rrbracket$	(5.6)
Assignment	`x = e;`	=	$\lambda\sigma.\ \Lambda\sigma'.\ \int dx'\ \boxed{\texttt{e}}(\sigma)\llbracket x'\rrbracket \cdot \delta(\sigma\{x \mapsto x'\})\llbracket\sigma'\rrbracket$	(5.7)
Seq. composition	`A; B;`	=	$\lambda\sigma.\ \Lambda\sigma''.\ \int d\sigma'\ \boxed{\texttt{A}}(\sigma)\llbracket\sigma'\rrbracket \cdot \boxed{\texttt{B}}(\sigma')\llbracket\sigma''\rrbracket$	(5.8)
Observation	`observe(e);`	=	$\lambda\sigma.\ \Lambda\sigma'.\ \delta(\sigma)\llbracket\sigma'\rrbracket \cdot p(\sigma) \quad \text{where } p(\sigma) := \int dx\ \boxed{\texttt{e}}(\sigma)\llbracket x\rrbracket \cdot [x \neq 0]$	(5.9)
Continuous obs.	`A; cobserve(b,c);`	=	$\lambda\sigma.\ \Lambda\sigma'.\ \int dy\ \left(\left(\boxed{\texttt{A}}(\sigma)\llbracket\sigma'\rrbracket \cdot \boxed{\texttt{b}}(\sigma')\llbracket y\rrbracket\right) /\!\!/ \lambda\llbracket y\rrbracket\right) \cdot \boxed{\texttt{c}}(\sigma')\llbracket y\rrbracket$	(5.10)
Control flow	`if e {A} else {B}`	=	$\lambda\sigma.\ \Lambda\sigma'.\ \int dx\ \boxed{\texttt{e}}(\sigma)\llbracket x\rrbracket \cdot \Big([x \neq 0] \cdot \boxed{\texttt{\{A\}}}(\sigma)\llbracket\sigma'\rrbracket\ +\ [x = 0] \cdot \boxed{\texttt{\{B\}}}(\sigma)\llbracket\sigma'\rrbracket\Big)$	(5.11)
Scoping	`{A}`	=	$\lambda\sigma.\ \Lambda\sigma'.\ \int d\sigma''\ \boxed{\texttt{A}}(\sigma')\llbracket\sigma''\rrbracket \cdot \delta(\sigma'' \setminus \{x : \text{variable } x \text{ introduced in } \texttt{A}\})\llbracket\sigma'\rrbracket$	(5.12)
Inference	`infer(f)`	=	$\lambda\sigma.\ \Lambda x.\ \delta\Big(\Lambda y.\ \boxed{\texttt{f}}()\llbracket y\rrbracket \cdot Z^{-1}\Big)\llbracket x\rrbracket \quad \text{where } Z := \int dz\ \boxed{\texttt{f}}()\llbracket z\rrbracket$	(5.13)
Sample	`sample(d)`	=	$\lambda\sigma.\ \Lambda z.\ \int dx\ \boxed{\texttt{d}}(\sigma)\llbracket x\rrbracket \cdot x\llbracket z\rrbracket$	(5.14)
Expectation	`expectation(d)`	=	$\lambda\sigma.\ \Lambda z.\ \int dx\ \boxed{\texttt{d}}(\sigma)\llbracket x\rrbracket \cdot \delta\left(\int dy\ x\llbracket y\rrbracket \cdot y\right)\llbracket z\rrbracket$	(5.15)
Function	`(){ A; return e; }`	=	$\lambda\sigma.\ \Lambda z. \int d\sigma'\ \boxed{\texttt{A}}(\sigma)\llbracket\sigma'\rrbracket \cdot \boxed{\texttt{e}}(\sigma')\llbracket z\rrbracket$	(5.16)

Figure 5.6: Key translation rules, ignoring error states. The rules are recursive and we write $\boxed{\texttt{a}}$ to denote the translation of a.

A key property of λPSI's symbolic domain is the fact that it can represent distributions which are only partially continuous. For example, the uniform distribution over an interval $[a, b]$ (i.e., `uniform(a,b)`) is represented by

$$\lambda a, b.\ \Lambda x.\ [a < b] \cdot \frac{1}{b-a} \cdot [a \leqslant x] \cdot [x \leqslant b] \cdot \lambda\llbracket x\rrbracket \tag{5.2}$$

$$+ [a = b] \cdot \delta(a)\llbracket x\rrbracket \tag{5.3}$$

$$+ [b < a] \cdot \delta(\bot)\llbracket x\rrbracket\,. \tag{5.4}$$

This distribution is parametric in a and b, and it consists of three parts. For $a < b$, the part (5.2) defines a continuous uniform distribution between a and b. In part (5.3), the interval only includes a single point and we hence place a point mass on a. The case $b < a$ is treated as an error and we put all the probability mass on the error value $\bot$ in part (5.4).

5.3.4 *Comparison to PSI*

As a core difference to PSI from Chapter 4, λPSI's symbolic domain closely follows the measure-theoretic interpretation of its terms (see §5.3.2). In particular, it introduces explicit Lebesgue measures ($\lambda\llbracket x\rrbracket$) for continuous distributions and explicitly specifies the random output variable of a Dirac delta. While the expression $\delta(x)$ in PSI is equivalent to $\delta(0)\llbracket x\rrbracket$ in λPSI, the formal interpretation of the PSI expression $\delta(x - y)$ is unclear. In λPSI, this is equivalent to either $\delta(x)\llbracket y\rrbracket$ or $\delta(y)\llbracket x\rrbracket$.

Note that in λPSI, the error state ($\bot$) is integrated in the symbolic domain instead of being treated separately in the program state. Also, data structures (tuples, arrays, and records) are directly modeled by λPSI's representation.

5.4 FROM PROGRAMS TO SYMBOLIC REPRESENTATIONS

We now show how λPSI translates programs to the symbolic domain of §5.3. This is the first step of performing exact inference for higher-order probabilistic programs.

5.4.1 *Translating Programs to the Symbolic Domain*

A λPSI program is translated recursively. For each statement Stmt we represent the posterior distribution over values of all program variables *given* their previous values and any observations made within Stmt. More specifically, a statement Stmt is translated to an expression of the form $\lambda\sigma.\ \Lambda\sigma'.\ f(\sigma)\,[\![\sigma']\!]$, which takes as input a *state* σ before executing Stmt, and returns the distribution over the state σ' after executing Stmt in σ. A state is a record containing values for all accessible variables. Similarly, an expression Ex is translated to a distribution of the form $\lambda\sigma.\ \Lambda x.\ f(\sigma)[\![x]\!]$. This symbolic expression takes as input a state σ and returns the distribution over the value of Ex in the state σ.

Fig. 5.6 shows selected key rules of the translation, which is defined recursively. To reduce clutter, the presented rules ignore error handling and polymorphic types. We will discuss incorporating error states in §5.4.2.

BASIC EXPRESSIONS Variables are translated to the point mass distribution on the value of the variable according to the state (analogously for constants), see rule (5.5).

The rule for binary operations (5.6) is instantiated for addition, but works analogously for other deterministic expressions. The probability that `a+b` evaluates to a value x is computed by integrating over all possible values y and z for `a` and `b`, respectively, such that their sum $y + z$ equals x. In rule (5.6), this is expressed by recursively translating `a` and `b`, and introducing a Dirac delta. Note that because λPSI expressions do not have side-effects, the probabilities for the values of `a` and `b` are independent given the current state σ.

DISTRIBUTIONS Sampling from built-in distributions (using, for example, the functions `flip` or `exponential`) is directly translated to a symbolic representation, as exemplified in §5.3.3.

BASIC STATEMENTS For assignments `x = e`, we translate `e` to obtain the distribution over all possible right-hand sides x' in state σ. The new state σ' is equal to σ except that the value of variable `x` may be any such x' with the according probability. This is expressed using an integral over x' in (5.7).

The rule for sequential composition (5.8) is based on the standard chain rule for probabilities. In particular, the rule integrates over all possible intermediate states σ'.

```
1   infer(() {
2       x := 2 + uniform(0,3);
3       observe(x >= 3);
4       return x;
5   })
```

Figure 5.7: Nested inference example.

OBSERVATIONS An observation `observe(e)` restricts the possible output states according to the boolean expression `e`. Intuitively, this amounts to setting the probabilities of all states violating `e` to zero and re-normalizing the resulting distribution. Rule (5.9) first computes the probability $p(\sigma)$ of `e` evaluating to a non-zero number (meaning, true) in the current state σ. If `e` is deterministic, $p(\sigma)$ is either 0 or 1. However, note that `e` may involve random choices such as in `observe(uniform(0,2) < 1)`, where $p(\sigma)$ is $\frac{1}{2}$. Next, the rule rescales the probability of the current state σ using $\delta(\sigma)\llbracket\sigma'\rrbracket \cdot p(\sigma)$. The resulting distribution over σ' may not be normalized any more, but will be re-normalized later.

The effect of `observe` can be better understood under sequential composition. Consider the code in Fig. 5.7. After Line 2, `x` is uniformly distributed between 2 and 5. For Line 3, the probability $p(x)$ that `x >= 3` evaluates to true is $[x \geqslant 3]$. According to the rule for sequential composition (5.8), the distribution over `x` after Line 3 is obtained by integrating over all intermediate values of `x` after Line 2. The factor $p(x)$ "cuts off" the distribution below 3 and we obtain the (unnormalized) uniform distribution between 3 and 5, as expected. In §5.5, we will see how λPSI formally derives this in a sequence of translation and simplification steps.

CONTINUOUS OBSERVATIONS Rule (5.10) translates continuous observations of the shape `cobserve(b,c)`. For this, it incorporates all statements `A` preceding the observation in the current statement block (if there are none, `A` can be treated as an empty statement). This rule takes precedence over rule (5.8).

First, we recursively translate `A` and `b` to obtain a distribution for σ' and the value y of `b`. For `cobserve` to be defined, it must be possible to rewrite this distribution such that it involves a Lebesgue measure factor $\lambda\llbracket y\rrbracket$. Next, we eliminate this Lebesgue measure using disintegration ($/\!\!/$) and replace it by the distribution of `c`. This will become more clear once we discuss rules for disintegration in §5.5.5.

CONTROL FLOW AND SCOPING For if-then-else statements, we first translate both branches. A branch may introduce local variables in its scope, which must not occur in the output distribution. Hence, we use rule (5.12) to marginalize all variables introduced in a branch and obtain a distribution over all variables in the outer scope. Next, rule (5.11) translates the condition `e` and integrates over all possible values of `e`, always selecting the appropriate branch. Loops in λPSI are bounded and are unrolled during translation.

NESTED INFERENCE A key insight of λPSI is that the process of inference itself is directly expressible in λPSI's symbolic domain. The result is a distribution over the inferred distribution. In order to translate `infer(f)`, rule (5.13) first recursively translates the zero-argument function `f` and computes the normalization constant Z. Next, the rule normalizes the distribution represented by `f` and returns the Dirac delta at that position. Note that inference is deterministic and hence translated to a point mass.

Given a distribution `d`, the expression `sample(d)` draws a sample from `d`. Assume `d` is computed as follows (note that `Flip` is a distribution, while `flip` is a sample):

```
d := Flip(1/2);
if flip(1/4) { d = Flip(1/3); }
```

To compute the probability of a sample z from `d` we need to sum (i) the probability that `flip(1/4)` is false and z is generated by `Flip(1/2)`, and (ii) the probability that `flip(1/4)` is true and z is generated by `Flip(1/3)`. In general, we need to integrate over all possible distributions x represented by `d` and compute the probability of z according to x, see rule (5.14).

To translate the expression `expectation(d)`, we also integrate over all possible distributions x. For each such distribution, we compute the expectation by the standard definition (i.e., $\int dy\, x[\![y]\!] \cdot y$) and construct the point mass on that value. The result is a distribution over the expected value of d.

FUNCTIONS For simplicity, consider a function containing only one return statement at the end, i.e.the body has the form `A; return e;` (the general case is similar). In rule (5.16), we first translate `A` to obtain a distribution over the state σ', which comprises all variables in the function's scope. Then, we translate `e` to obtain a distribution over the return value in the state σ'. Finally, we integrate over all possible states σ'. Note that in general, the resulting distribution may be parameterized by the function's arguments (not shown).

RENORMALIZATION The entry point for a λPSI program is its `main` function, which may accept parameters. This function is translated just as any other function according to rule (5.16), but λPSI renormalizes the distribution before returning the result (similarly as in rule (5.13) for `infer`). Note that the normalization constant may depend on the parameters of `main`.

5.4.2 *Accounting for Error States*

Statements and expressions in λPSI may lead to errors under some states. Examples include divisions by zero and passing non-conforming parameters to distributions such as $a > b$ in `uniform(a,b)`. λPSI incorporates the probability of an error in the computed posterior distributions: the representation $\lambda\sigma.\, \Lambda\sigma'.\, f(\sigma)[\![\sigma']\!]$ of a statement assigns to each *non-error* starting state σ the distribution over the output state σ',

`2 + uniform(0,3)`

$$\overset{\S5.4}{=} \lambda\sigma.\ \Lambda x.\ \int dy \int dz\ \boxed{2}(\sigma)[\![y]\!] \cdot \boxed{\texttt{uniform(0,3)}}(\sigma)[\![z]\!] \cdot \delta(y+z)[\![x]\!] \quad (5.17)$$

$$\overset{\star}{=} \lambda\sigma.\ \Lambda x. \int dy \int dz\ \delta(2)[\![y]\!] \cdot \tfrac{1}{3} \cdot [0 \leqslant z] \cdot [z \leqslant 3] \cdot \lambda[\![z]\!] \cdot \delta\Big(y + z\Big)[\![x]\!] \quad (5.18)$$

$$\overset{\S5.5.1}{=} \lambda\sigma.\ \Lambda x. \int dz\ \tfrac{1}{3} \cdot [0 \leqslant z] \cdot [z \leqslant 3] \cdot \lambda[\![z]\!] \cdot \delta(2+z)[\![x]\!] \quad (5.19)$$

$$\overset{\S5.5.2}{=} \lambda\sigma.\ \Lambda x. \int dz\ \tfrac{1}{3} \cdot [0 \leqslant z] \cdot [z \leqslant 3] \cdot \lambda[\![x]\!] \cdot \delta(x-2)[\![z]\!] \quad (5.20)$$

$$\overset{\S5.5.1}{=} \lambda\sigma.\ \Lambda x.\ \tfrac{1}{3} \cdot [2 \leqslant x] \cdot [x \leqslant 5] \cdot \lambda[\![x]\!] \quad (5.21)$$

```
x := 2 + uniform(0,3);
observe(x >= 3);
```

$$\overset{\S5.4}{=} \lambda\sigma.\ \Lambda\sigma''.\ \int d\sigma'\ \boxed{\texttt{x := 2 + uniform(0,3)}}(\sigma)[\![\sigma']\!] \cdot \boxed{\texttt{observe(x >= 3)}}(\sigma')[\![\sigma'']\!] \quad (5.22)$$

$$\overset{\star}{=} \lambda\sigma.\ \Lambda\sigma''. \int d\sigma' \int dx'\ \tfrac{1}{3} \cdot [2 \leqslant x'] \cdot [x' \leqslant 5] \cdot \lambda[\![x']\!] \cdot \delta(\sigma\{x \mapsto x'\})[\![\sigma']\!] \cdot \delta\Big(\sigma'\Big)[\![\sigma'']\!] \cdot [\sigma'.x \geqslant 3] \quad (5.23)$$

$$\overset{\S5.5.1}{=} \lambda\sigma.\ \Lambda\sigma''.\ \int dx'\ \tfrac{1}{3} \cdot [2 \leqslant x'] \cdot [x' \geqslant 3] \cdot [x' \leqslant 5] \cdot \lambda[\![x']\!] \cdot \delta(\sigma\{x \mapsto x'\})[\![\sigma'']\!] \quad (5.24)$$

$$\overset{\S5.5.3}{=} \lambda\sigma.\ \Lambda\sigma''.\ \int dx'\ \tfrac{1}{3} \cdot [3 \leqslant x'] \cdot [x' \leqslant 5] \cdot \lambda[\![x']\!] \cdot \delta(\sigma\{x \mapsto x'\})[\![\sigma'']\!] \quad (5.25)$$

Fig. 5.7

$$\overset{\S5.4}{=} \lambda\sigma.\ \Lambda x.\ \delta\Big(\Lambda y.\ \boxed{\text{Lines 2–4 in Fig. 5.7}}()[\![y]\!] \cdot \Big(\int dz\ \boxed{\text{Lines 2–4 in Fig. 5.7}}()[\![z]\!]\Big)^{-1}\Big)[\![x]\!] \quad (5.26)$$

$$\overset{\star}{=} \lambda\sigma.\ \Lambda x.\ \delta\Big(\Lambda y.\ \tfrac{1}{3} \cdot [3 \leqslant y] \cdot [y \leqslant 5] \cdot \lambda[\![y]\!] \cdot \Big(\int dz\ \tfrac{1}{3} \cdot [3 \leqslant z] \cdot [z \leqslant 5] \cdot \lambda[\![z]\!]\Big)^{-1}\Big)[\![x]\!] \quad (5.27)$$

$$\overset{\S5.5.4}{=} \lambda\sigma.\ \Lambda x.\ \delta(\Lambda y.\ \tfrac{1}{2} \cdot [3 \leqslant y] \cdot [y \leqslant 5] \cdot \lambda[\![y]\!])[\![x]\!] \quad (5.28)$$

Figure 5.8: Selected steps of deriving a simplified representation of the code in Fig. 5.7. The terms affected by substitution (§5.5.1), linearization (§5.5.2), guard simplification (§5.5.3), and symbolic integration (§5.5.4) are highlighted. Equalities annotated with ⋆ denote recursive translation and simplification.

which may be the error state ⊥ (similarly for expressions). Symbolic distributions make use of Dirac deltas $\delta(\perp)[\![x]\!]$ to capture the probability of an error (see for example Eq. (5.4)).

The presence of errors slightly complicates the translation rules of Fig. 5.6. In particular, for all integrals of the form $\int d\sigma\, f[\![\sigma]\!]$, the integration domain also includes ⊥ (as f may cause an error) and we hence must analyze the case $\sigma = \perp$ separately. For instance, the rule for sequential composition needs to propagate errors caused in `A` through `B` using an expression of the form $e_{1?}(e_2)$:

$$\lambda\sigma.\ \Lambda\sigma''.\ \int d\sigma'\ \boxed{\texttt{A}}(\sigma)\,[\![\sigma']\!] \cdot \boxed{\texttt{B}}_?(\sigma')\,[\![\sigma'']\!]\ .$$

We do not further discuss error states.

5.5 INFERENCE BY SYMBOLIC SIMPLIFICATION

We now present how the symbolic representation of a translated program is simplified to a compact representation. This constitutes the second step of λPSI's inference procedure.

As we discuss in §5.5.6, the presented simplifications are an extension of the symbolic optimizations used by PSI. In particular, we (i) generalize PSI's rules to λPSI's more powerful symbolic domain, and (ii) improve the former's efficiency using various (low-level) optimizations.

Substitution

$$\int dx\ f(x) \cdot \delta(v)[\![x]\!] \ = \ f(v) \tag{5.29}$$

Linearization

$$\delta(f(x))[\![y]\!] \cdot \lambda[\![x]\!] = [f'(x) = 0] \cdot \delta(f(x))[\![y]\!] \cdot \lambda[\![x]\!] + [f'(x) \neq 0] \cdot \overbrace{\textstyle\sum_{z:f(z)=y} \underbrace{\delta(z)[\![x]\!] \,/|f'(z)| \cdot \lambda[\![y]\!]}_{\text{part 1}}}^{\text{part 2}} \tag{5.30}$$

Disintegration

$$(e \cdot \lambda[\![x]\!]) /\!\!/ \lambda[\![x]\!] \ = \ e \tag{5.31}$$

Figure 5.9: Simplifying Dirac deltas and Lebesgue measures. Here, f' is the derivative of f.

BASIC ALGEBRAIC SIMPLIFICATIONS λPSI applies various basic algebraic rules, such as removing multiplications by 1 and additions with 0, and simplifying terms multiplied by 0 to 0. It further leverages commutative, associative, and distributive laws where applicable. In general, integrals over sums are simplified to sums of integrals, and constant factors within integrals are moved out of the integrals.

RUNNING EXAMPLE We next describe the most important simplification rules on a running example. Concretely, we translate and simplify the λPSI expression in Fig. 5.7, while discussing a selection of interesting simplification steps (Fig. 5.8).

We start by translating and simplifying the expression `2 + uniform(0,3)` (Line 2 in Fig. 5.7). We apply the rule for binary operations (5.6) to obtain (5.17), see Fig. 5.8. Next, the constant `2` is translated to a point mass, and `uniform(0,3)` is translated according to (5.2).

5.5.1 *Dirac Delta Substitution*

Expression (5.18) contains an integral over y, which occurs as an "output" of a Dirac delta (see highlighted)—a common structure. Intuitively, we know that $\delta(2)[\![y]\!]$ is zero for all $y \neq 2$. Hence, we can simplify the expression by removing the integral and Dirac delta, and substituting all occurrences of y by 2 to obtain (5.19). In general, integrals over the output variable of a Dirac delta result in substituting the variable. This key rule is shown in (5.29) of Fig. 5.9.

5.5.2 *Dirac Delta Linearization*

The structure of (5.19) is similar as before, but this time the integration variable z occurs in the first argument to δ. In general, λPSI often encounters expressions of the form $\int dx\ g[\![x]\!] \cdot \delta(f(x))[\![y]\!]$, which can be interpreted as y depending deterministically on x by $y = f(x)$. If g is a Dirac delta, we can apply (5.29) to substitute x in $f(x)$. Otherwise, we would like to express $\delta(f(x))[\![y]\!]$ in terms of $\delta(h(y))[\![x]\!]$ for some

h such that we can later apply substitution (5.29). This is achieved by *linearization*, which rewrites the original Dirac delta over y as a linear combination of Dirac deltas over x.

LINEARIZATION IN A SIMPLE EXAMPLE Let us have a look at our running example (5.19), where $g[\![z]\!] = \frac{1}{3} \cdot [0 \leqslant z] \cdot [z \leqslant 3]$ is the uniform distribution on $[0, 3]$. Instead of first selecting z uniformly between 0 and 3 (by g) and then setting x to $2 + z$ (by the Dirac delta), we can just as well directly select x uniformly between 2 and 5.

Intuitively, $\delta(2+z)[\![x]\!]$ is only non-zero at locations where $x = 2 + z$, or equivalently $z = x - 2$. Hence, we can linearize this Dirac delta (see highlighted in (5.19)) by expressing z in terms of x and moving z to the second argument of δ, see (5.20). Note how thereby, $\lambda[\![z]\!]$ changes to $\lambda[\![x]\!]$. Then, we can apply substitution (5.29) to obtain the desired result in (5.21).

THE GENERAL CASE Rewriting $\delta(f(x))[\![y]\!]$ for general f requires more care. We now explain the general rule as presented in Fig. 5.9, Eq. (5.30).

To highlight a first issue with our previous attempt, inspect the following normalized distribution over y:

$$\Lambda y.\ \int dx\ [0 \leqslant x] \cdot [x \leqslant 1] \cdot \lambda[\![x]\!]\, \delta(2x)[\![y]\!]\,.$$

Incorrectly linearizing $\lambda[\![x]\!]\, \delta(2x)[\![y]\!]$ to $\lambda[\![y]\!]\, \delta(y/2)[\![x]\!]$ gives

$$\int dx\ [0 \leqslant x] \cdot [x \leqslant 1] \cdot \lambda[\![y]\!]\, \delta(y/2)[\![x]\!] \overset{(5.29)}{=} [0 \leqslant y] \cdot [y \leqslant 2] \cdot \lambda[\![y]\!]$$

which is not normalized anymore. In fact, we would need to introduce a factor $\frac{1}{2}$. As can be shown by the substitution rule of Lebesgue integration, in general one needs to divide by the absolute value of the derivative f' of f (part 1 in Fig. 5.9).

Second, there may be more than one value x for which $f(x) = y$ (i.e., f may not be invertible). For example, for $y > 0$ the Dirac delta $\delta(x^2)[\![y]\!]$ is non-zero for both $x = \sqrt{y}$ and $x = -\sqrt{y}$, so the linearized expression is a sum of two Dirac deltas at these positions (see part 2 in Fig. 5.9).

Because part 2 is not defined for locations where the derivative of f is zero, we need to treat such locations separately. For this reason, (5.30) distinguishes $f'(x) = 0$ and $f'(x) \neq 0$.

For the first case, we can often find all solutions x of $f'(x) = 0$ and substitute these in $f(x)$. For example, consider the function f given in Fig. 5.10 whose derivative is zero at $x = 0$, everywhere below -1, and everywhere above 1. We can hence rewrite $[f'(x) = 0]$ to $[x \leqslant -1] + [x \geqslant 1] + [x = 0]$ and distribute $\delta(f(x))[\![y]\!]$ over these three summands. Because we know that $f(0) = 2$, we can rewrite $[x = 0] \cdot \delta(f(x))[\![y]\!] = [x = 0] \cdot \delta(2)[\![y]\!]$. Further, because the derivative is zero, we know that $f(x)$ must have

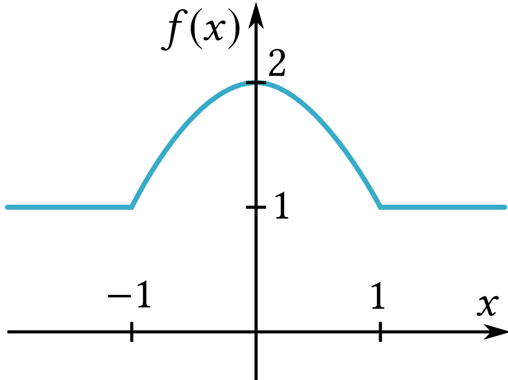


Figure 5.10: The function $f(x) = 1 + [-1 \leqslant x] \cdot [x \leqslant 1] \cdot (1 - x^2)$.

the same value (namely, 1) for all $x \leqslant -1$. Hence, we can rewrite $[x \leqslant -1] \cdot \delta(f(x))[\![y]\!]$ to $[x \leqslant -1] \cdot \delta(1)[\![y]\!]$ (similarly for $x \geqslant 1$).

5.5.3 *Guard Simplifications*

We continue our running example by translating and simplifying Line 2 and Line 3 of Fig. 5.7. These lines are translated to (5.22) using the rule for sequential composition (5.8), and instantiating the simplified expressions (steps not shown in Fig. 5.8) gives (5.23). Because the integration variable σ' is the output of a Dirac delta (see highlighted), we can again apply substitution (5.29). Note how the access of field $\mathtt{x}$ in $\sigma'.\mathtt{x}$ is simplified to x', because σ' is substituted by $\sigma\{\mathtt{x} \mapsto x'\}$.

In the resulting expression (5.24), there are multiple Iverson bracket factors imposing constraints on x' (called *guards*). In particular, x' is bounded from below by both 2 and 3 due to the highlighted factors. As the constraint $2 \leqslant x'$ is implied by $x' \geqslant 3$, we simplify the two factors to $[3 \leqslant x']$ in (5.25).

In addition to eliminating redundant guards, λPSI supports many more *guard simplifications* (mostly inherited from PSI). For example, it simplifies whole terms to 0 if the therein contained guards are unsatisfiable (e.g., as in $[x = 0] \cdot [x \neq 0]$). Also, λPSI analyzes complex guard constraints (such as quadratic polynomials) to rewrite them as a combination of simpler, linear guard constraints (e.g., we rewrite $[x^2 \geqslant 4]$ as $[x \geqslant 2] + [x \leqslant -2]$). Guard simplifications are also used to simplify $[f'(x) = 0]$ during linearization (5.30), see our previous example in §5.5.2.

5.5.4 *Symbolic Integration*

We continue translating and simplifying Fig. 5.7, which performs nested inference at the top level. Recall that nested inference can be directly represented in λPSI's symbolic domain: using rule (5.13), we translate `infer` to (5.26). Recursively translating and simplifying Lines 2–4 gives (5.27).

The normalization constant (highlighted) is an integral to be simplified. This time, the integrand does not contain any Dirac deltas, so the simplification rules of §5.5.1 and §5.5.2 do not apply. However, the integrand is simply a constant function

between 3 and 5, hence we can simplify the integral to $\frac{5-3}{3} = \frac{2}{3}$. The resulting expression (5.28) is fully simplified and represents the posterior distribution over the value of the expression from Fig. 5.7.

SIMPLIFYING INTEGRALS λPSI extends PSI's powerful engine for symbolic integration of a wide class of functions not involving Dirac deltas. To simplify such integrals, λPSI first applies guard simplifications (§5.5.3) in order to determine the integration bounds. Note that a single guard constraint may be simplified to a sum of guards, hence this step may split an integral into a sum of integrals. Next, λPSI leverages antiderivatives of known function classes (e.g., polynomials and logarithms) and standard integration rules (e.g., integration by parts) to find the integrand's antiderivative. If this succeeds, the latter is evaluated at the bounds to give the final result.

SIMPLIFYING SUMS λPSI applies similar techniques to simplify absolutely convergent series, which may for example occur when computing expectations of discrete distributions. For example, while simplifying `expectation(Geometric(`$\frac{1}{4}$`))` λPSI encounters the following expression (cp. (5.1)):

$$\begin{aligned} &\int dx \ \textstyle\sum_{i \in \mathbb{Z}} [i \geqslant 0] \cdot (\frac{3}{4})^i \cdot \frac{1}{4} \cdot \delta(i)[\![x]\!] \cdot x. \\ &\overset{(5.29)}{=} \frac{1}{4} \cdot \textstyle\sum_{i \in \mathbb{Z}} [i \geqslant 0] \cdot (\frac{3}{4})^i \cdot i. \end{aligned} \tag{5.32}$$

λPSI identifies several convergent series with known values and heavily makes use of Abel's lemma (summation by parts) [6] to simplify such expressions. Using this, it can for instance simplify (5.32) to the value 3.

5.5.5 *Symbolic Disintegration*

Consider the following code snippet:

```
x := gauss(μ,ν); cobserve(2·x, y);
```

The `cobserve` statement conditions on the possible but probability zero event that `2·x` equals an observed value `y`. Intuitively, this has two effects: (i) the current program path is reweighted by $\frac{1}{2}f(y/2; \mu, \nu)$, where f is the Gaussian density, and (ii) the value of `2·x` is fixed to the observed value `y`. We now derive these effects from our translation and simplification rules. After translation and some simplification steps (σ and σ' omitted for brevity), the distribution of `x` is

$$\begin{aligned} &\int dz \, ((f(x;\mu,\nu) \cdot \lambda[\![x]\!] \cdot \delta(2 \cdot x)[\![z]\!]) /\!\!/ \lambda[\![z]\!]) \cdot \delta(y)[\![z]\!] \\ &\overset{(5.30)}{=} \int dz \, \Big(\big(\tfrac{1}{2} f(x;\mu,\nu) \cdot \delta(z/2)[\![x]\!] \cdot \lambda[\![z]\!] \big) /\!\!/ \lambda[\![z]\!] \Big) \cdot \delta(y)[\![z]\!] \, . \end{aligned}$$

We purposefully used Dirac delta linearization to write the joint prior distribution of x and $z = 2 \cdot x$ with an explicit factor $\lambda[\![z]\!]$. Now, we use the disintegration rule (5.31) to eliminate the highlighted $\lambda[\![z]\!]$, obtaining the desired weighted Dirac delta:

$$\int dz\ \tfrac{1}{2} f(x;\mu,\nu) \cdot \delta\left(\tfrac{z}{2}\right)[\![x]\!] \cdot \delta(y)[\![z]\!] = \tfrac{1}{2} f\left(\tfrac{y}{2};\mu,\nu\right) \cdot \delta\left(\tfrac{y}{2}\right)[\![x]\!]$$

In general, rule (5.31) transforms the density of the first argument of `cobserve` to a weight for the remaining distribution. Note that due to its powerful Dirac delta linearizer, λPSI can symbolically disintegrate some programs that are not handled by Shan and Ramsey [148].

5.5.6 *Comparison to PSI*

The main differences to PSI from chapter 4 are related to adding support for the new terms of the symbolic domain (see Fig. 5.5) and are hence purely additive. Still, λPSI introduces major design and implementation improvements. We list the most important differences to PSI's symbolic optimizations below.

While basic arithmetic simplifications and guard simplifications (§5.5.3) are mostly inherited from PSI, some low-level improvements were added (e.g., PSI can not simplify guards involving reciprocals of polynomials). Unlike in PSI, the Dirac delta of λPSI has an explicit "output" argument, but the rules for Dirac delta substitution (§5.5.1) are analogous. Linearization (§5.5.2) closely follows the rules already present in PSI. However, the rewrites allowed in λPSI are more restricted because Lebesgue measures are no longer implicit and must be present. While simplification rules for integrals (§5.5.4) are mainly inherited from PSI, λPSI introduces many non-trivial simplifications of sums (e.g., to simplify expectations). Disintegration (§5.5.5) is new, as PSI does not support `cobserve`.

5.5.7 *Limitations*

λPSI's simplifications are only best-effort, i.e., sound but not complete. Like virtually all existing exact inference and incomplete computer algebra systems, its limitations (what can and can not be simplified) are hard to characterize. Generally speaking, the limitations of λPSI are related to inference being intractable in general. In particular, not all programs have closed-form representations in the symbolic domain of Fig. 5.5, and no algorithm (efficient or not) will always be able to decide if such representations exist.

However, in §5.6 we show that λPSI's simplification rules work well for a set of benchmark programs. We also show an example which can not be simplified by λPSI.

Table 5.1: Probabilistic programs used in evaluation (31 in total). For each program, we indicate if it involves higher-order functions ($\rightarrow$), nested inference ($\rightsquigarrow$), first-class expectations ($\mathbb{E}$), continuous distributions ($\wedge$), continuous observations ($\curlywedge$), or symbolic parameters ($\boxed{a}$). SocialCognition and TotalVarDist have multiple variants. Some programs are not expressible in Hakaru (–), while others lead to errors (×), unsimplified (■) or incorrect (× ×) results. [a]Not directly expressible; rewritten as first-order programs without function calls, multiple manual steps. [b]For concrete instantiation of symbolic parameters.

Program(s)	Description	Features						Runtime	
		$\rightarrow$	$\rightsquigarrow$	$\mathbb{E}$	$\wedge$	$\curlywedge$	$\boxed{a}$	Hakaru [148]	λPSI
SocialCognition (12) [64]	Multiple rational agent models (see §5.1)		•					× / × ×	<5.5s
CondProb (Fig. 5.13)	Compute conditional probability using expectation operator	•	•	•	•		•	×[ab]	3s
Overview (Fig. 5.2)	Example involving multiple language features		•		•			×	0.3s
ChannelCap	Mutual information between input and output of noisy channel	•	•	•			•	-	2s
Entropy	Entropy of randomly generated sequence		•	•			•	-	3s
GenCap [22]	Generalization capacity of sorting algorithms (see Fig. 5.14)	•	•	•				-	16s
AIDE [38]	KL-divergence between particle filter and exact inference on HMM	•	•	•				-	42s
BivariateIndep	Verify that bivariate distribution has independent components	•	•	•				-	0.1s
SecretSanta	Five people guess secret santa in turn, based on uniform prior		•	•				×	0.5s
TotalVarDist (2) [13]	Total variation distance for random walk and Dynkin process (20 steps)		•	•				×	< 5s
MontyHall	Monty hall problem variants modeled using nested inference		•	•				×	0.1s
Variance	Compute variance of given distribution		•	•	•			-	0.5s
CDF	Compute CDF of Gaussian distribution at a point drawn from it		•	•	•			×	0.1s
GANLoss	GAN loss for simple probabilistic model against optimal discriminator	•	•	•	•		•	-	0.7s
FairSVM [164]	Infer weights for fair SVM classifier	•	•	•	•			×	■
BayesLinReg [75]	Bayesian linear regression from 5 data points with Gaussian noise	•			•	•	•	2s[ab]	0.2s
BayesPiecewiseLR [76]	Bayesian piecewise linear regression from 7 data points	•			•	•		×	33s
DisintegrateLinear [148]	Motivating example from [148], disintegrate linear function two ways		•	•	•	•		4s[a]	0.2s
DisintegrateQuadratic	Disintegrate quadratic function (involves `cobserve((x-1)^2,y))`		•	•	•	•		■[a]	0.1s

5.5.8 *Correctness*

We do not provide an explicit embedding of λPSI's symbolic representation into a system widely accepted to be consistent, such as set theory. However, we note that the correctness of λPSI is nonetheless falsifiable. For example, we can write a program that computes a known real number or real function. The expression produced by λPSI will often be interpretable as a standard mathematical expression, which can be compared to the known result. There are also less explicit ways to falsify the correctness of λPSI: For example, if it were to compute a negative probability or probability density, we would know that it was incorrect.

5.6 EVALUATION

We implemented λPSI by extending the publicly available PSI PPL (`https://psisolver.org`) with the features from §5.2 and the exact inference capabilities from §5.3–§5.5.

We assembled a collection of 31 programs with higher-order constructs such as distributions over functions and nested inference, summarized in Table 5.1. The collection comprises examples from the literature (including the applications discussed in §5.1 and §5.6.2) and custom programs.

All our experiments were performed on a commodity laptop with 32 GB of RAM and 4 CPU cores at 2.60 GHz.

expressiveness and performance of λpsi We can express all programs succinctly in λPSI, as all required language features are supported as first-class citizens. For the FairSVM [164] example, there is no closed-form representation of the posterior in λPSI's symbolic domain as it depends non-trivially on properties of products of Gaussians (this is inherited from PSI). The following simple example can not be simplified by λPSI for the same reason:

```
def main() ⇒ gauss(0, 1)*gauss(0, 1) < 1;
```

For the remaining 30 examples, λPSI successfully infers a closed-form exact result (no integrals left) within at most 42 seconds. We conclude that λPSI is powerful enough to express interesting applications and that its simplification engine is effective.

5.6.1 *Comparison to Previous Work*

We compare λPSI to Hakaru [122], the only other system we are aware of that can perform exact inference for probabilistic programs with continuous distributions. Our goal is to validate that λPSI is the first tool that can perform exact symbolic inference on *higher-order* probabilistic programs with continuous distributions. As Hakaru terms support first-class functions, this is not immediately obvious.

Hakaru provides external transformations "normalize" (for inference with positive-probability evidence), "disintegrate" (for inference with continuous evidence), and "simplify" (to transform terms produced by the other transformations into closed-form representations). In λPSI, `infer` (normalize) and `cobserve` (disintegrate) are first-class operators and can therefore be used for higher-order inference.

without continuous observations Hakaru's normalize transformation can in principle be expressed as a Hakaru term using the "expect" operator to compute the total weight of a measure. We use this strategy to encode most of our examples without continuous observations in Hakaru (see Table 5.1). Unfortunately, this leads to an error relating to the "expect" operator in Hakaru's simplification engine. Hakaru's "expect" operator can only be used on functions bounded between 0 and 1, hence examples including Entropy and Variance are not encodable in Hakaru. Furthermore, as Hakaru can not simplify function terms, some programs can not be directly expressed in Hakaru, particularly those involving symbolic parameters. However, we manually rewrite some programs for which inlining functions is possible. For a fully inlined version of CondProb with concretized symbolic parameters and where we use an unnormalized observation instead of Bayes' rule, simplification leads to a stack overflow in Maple (used by Hakaru). For some examples without nested evidence (SocialCognition), simplify returns the zero

```
def sampleFunction(){
    (s,b):=(gauss(0,3^2), gauss(0,3^2));
    return (x) ⇒ s*x+b;
}

def main(X){
    f := sampleFunction();

    observations := [(1,2.5), (2,3.8), (3,4.5), (4,6.2), (5,8.0)];
    for i in [0..5){
        (x,y) := observations[i];
        cobserve(f(x)+gauss(0,(1/2)^2),y);
    }
}
```

Figure 5.11: Bayesian linear regression.

measure instead of the correct answer and disintegrate terminates with an error unless we inline all function definitions.

WITH CONTINUOUS OBSERVATIONS While Hakaru does not support first-class disintegration, this can sometimes be simulated by chained calls to Hakaru's disintegration, normalization and simplification engines. In cases where manual inlining of higher-order functions is easy (such as for BayesLinReg, see Table 5.1), we can use Hakaru to compute a result. Otherwise, Hakaru cannot easily be used to perform inference. For example, we cannot express BayesPiecewiseLR as an inlined Hakaru term without significant manual effort. The DisintegrateQuadratic example can be disintegrated by λPSI, but not Hakaru.

We suspect that one could automate our manual steps by directly using Hakaru's Monad within a Haskell program. Unfortunately, this mode of using Hakaru is not documented and does not seem to be encouraged. It is also important to note that this does not allow Hakaru to perform (non-trivial) nested inference, as it cannot simplify function terms.

5.6.2 *Case Studies*

We now elaborate on a number of applications presented in Table 5.1.

BAYESIAN REGRESSION Heunen et al. [75] motivate higher-order probabilistic programming by expressing linear regression as a prior over first-class functions f together with observed I/O examples. We use λPSI to compute the posterior density $p(y)$ of $y = f(x)$ in terms of x. Fig. 5.11 shows how to encode this inference problem in λPSI. We show a plot of the posterior after conditioning on 5 samples in Fig. 5.12.

We also encoded an example with a piecewise linear function prior [76], deriving the posterior for y at a specific x.

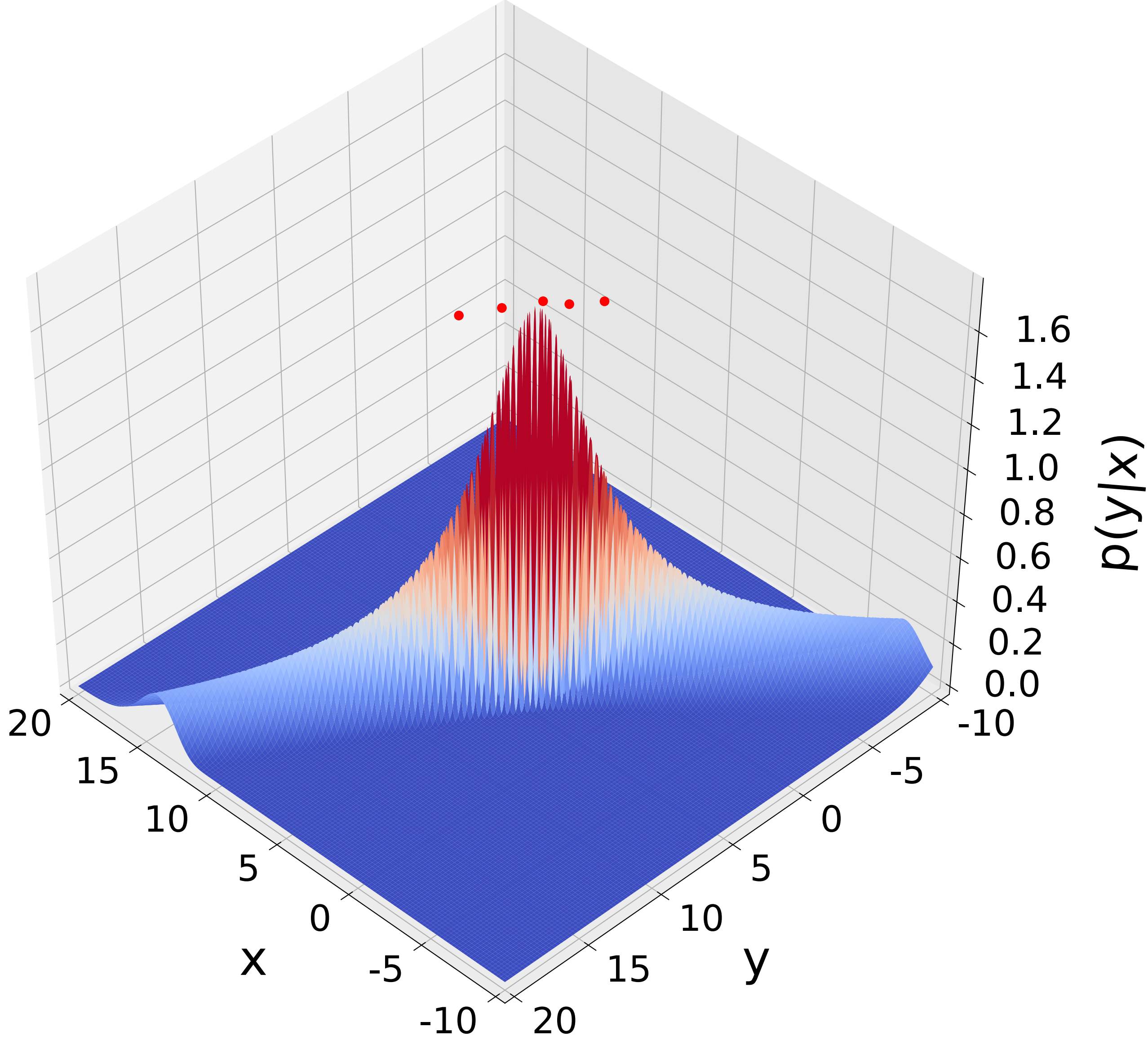
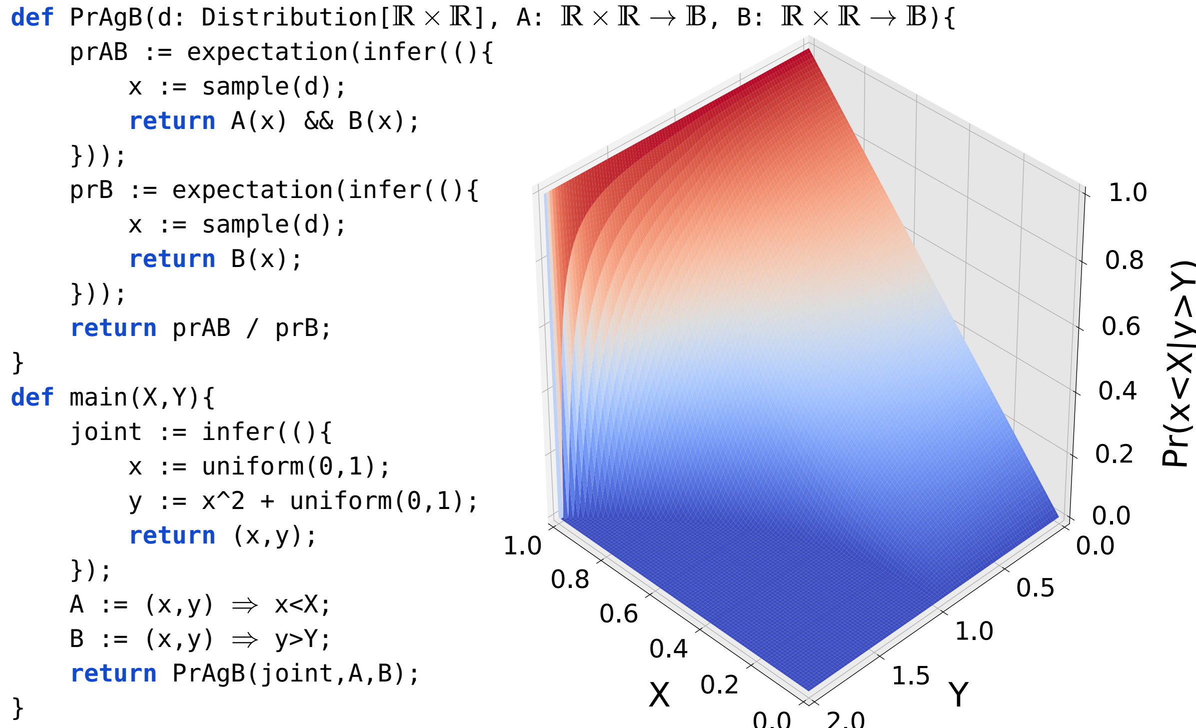


Figure 5.12: Bayesian linear regression: posterior of $y = f(x)$ after 5 samples.

Conditional with symbolic parameters Given a probability distribution Pr, an event A and observed evidence B, we want to compute $\Pr[A \mid B]$ (shown in Fig. 5.13). We use Bayes' rule directly (instead of `observe`). λPSI evaluates the resulting probability for all valid values for parameters `x` and `y` *simultaneously*; the result is shown in Fig. 5.13, right.

Entropy We can also naturally express information theoretical concepts such as *entropy*, *KL-divergence* and *mutual information*, shown in Fig. 5.14 (left).

For instance, the function `S` quantifies surprise caused by particular outcomes of a random experiment, and entropy `H` is the expected surprise with respect to the true distribution p. Cross-entropy is the expected surprise with respect to a wrong model q of the data. KL-divergence is the difference between the two. Given a joint distribution, mutual information measures the dependency of the two projections. The mutual information is zero if and only if the two random variables are independent. Note that there exists no (purely sampling-



```
def PrAgB(d: Distribution[ℝ×ℝ], A: ℝ×ℝ→𝔹, B: ℝ×ℝ→𝔹){
    prAB := expectation(infer((){
        x := sample(d);
        return A(x) && B(x);
    }));
    prB := expectation(infer((){
        x := sample(d);
        return B(x);
    }));
    return prAB / prB;
}
def main(X,Y){
    joint := infer((){
        x := uniform(0,1);
        y := x^2 + uniform(0,1);
        return (x,y);
    });
    A := (x,y) ⇒ x<X;
    B := (x,y) ⇒ y>Y;
    return PrAgB(joint,A,B);
}
```


Figure 5.13: Conditional probability Pr[x < X | y > Y] depending on X and Y.

based) unbiased estimator for entropy or mutual information [126]. Busse et al. [22] introduce *generalization capacity*, quantifying how much algorithms depend on noise in noisy input data. Fig. 5.14 (right) shows a λPSI encoding of this task.

The `genCap` function accepts an algorithm `f`, an input distribution `p`, and a probabilistic `noise` model. It computes the mutual information between the outputs of the algorithm on two different realizations of noise on the input. Intuitively, algorithms that are more vulnerable to noisy input data will get lower values. We compare the generalization capacity of three sorting algorithms on sequences of length 3. AIDE [38] infers an approximate upper bound on the expected (symmetrized) KL-divergence between the results of two inference approaches. Our AIDE benchmark exactly computes the expected KL-divergence between the results of a particle filter and exact inference, and shows that the particle filter gains precision as more particles are added.

As input, the algorithms obtain a matrix of pairwise element comparisons. The output is a permutation. The element comparisons are corrupted by randomly flipping each bit with probability 0.1. We also show how we use the `genCap` function. The three sorting algorithms are: (i) "optimal" sorting: it loops over all permutations of the input array and returns one that minimizes the number of inconsistencies with the comparison matrix (a matrix of uniform random permutation), (ii) bubble sort, and (iii) insertion sort. λPSI successfully determined that the optimal sorting algorithm has the highest generalization capacity and that bubble sort has a higher generalization capacity than insertion sort.

Reasoning exactly about information theoretic concepts also enables other applications, such as quantitative information flow.

```
def log₂(x) ⇒ log(x)/log(2); // unit: bits

// surprise
def S[a](x: a, q: Distribution[a]) ⇒
    -log₂(expectation(infer(() ⇒ x == sample(q))));

// entropy and cross-entropy:
def H[a](p: Distribution[a]) ⇒
    expectation(infer(()⇒S(sample(p), p)));
def Hcross[a](p: Distribution[a], q: Distribution[a]) ⇒
    expectation(infer(()⇒S(sample(p), q)));

// projections to marginals:
def π₁[a,b](p: Distribution[a×b])⇒infer(() ⇒ sample(d)[0]);
def π₂[a,b](p: Distribution[a×b])⇒infer(() ⇒ sample(d)[1]);

// KL-divergence and mutual information:
def KL[a](p: Distribution[a], q: Distribution[a]) ⇒ Hcross(p,q) - H(p);
def I[a,b](p: Distribution[a×b]) ⇒ H(π₁(p)) + H(π₂(p)) - H(p);

// generalization capacity:
def genCap[a,b](f: a→b, p: Distribution[a], noise: a→a) ⇒
    I(infer((){
        x := sample(p);
        return (f(noise(x)), f(noise(x)));
    }));

// generalization capacity of sorting algorithms:
def sortCap[n:ℕ](sort: 𝔹^(n×n)→ ℕ^n){
    input := RandomComparisonMatrix(n);
    error := noise(0.1);
    def evalSort(sort: ℝ[][] → ℝ[]){
        return genCap(sort, input, error);
    }
    return (evalSort(optimalSort),
            evalSort(bubbleSort),
            evalSort(selectionSort));
}
```

Figure 5.14: Information-theoretic quantities associated with discrete distributions and an application: generalization capacity (of sorting algorithms).

5.7 RELATED WORK

The semantics of higher-order probabilistic programs has been studied extensively [157, 158], resulting in the definition of the category of quasi-Borel spaces [75]. Ścibior et al. [145] formulate a framework for denotational verification of inference transformations, supporting higher-order probabilistic programs with continuous as well as discrete distributions. Based on this, Sato et al. [143] present a program logic.

While Hakaru [122, 148] does not currently provide exact inference support for higher-order constructs, the system has made other important advances, such as disintegrating programs with symbolic arrays [121], as well as exact reasoning about symbolic arrays to automatically and efficiently derive closed-form conditional distributions [168].

Tavares et al. [164] propose a new kind of higher-order inference operator that allows certain models with nested inference to be specified more concisely.

5.8 DISCUSSION

In this chapter, we presented λPSI, the first higher-order statically typed probabilistic programming language equipped with a solver that computes exact (symbolic) probability distributions of programs. We showed how to express several interesting applications (e.g., information theory, rational agents) in λPSI and demonstrated that our solver was able to compute their exact distributions.

This is the first time one is able to exactly analyze probabilistic programs at this level of expressiveness. In the future, we plan to investigate ways to further scale the exact inference algorithm as well as explore combinations with approximate inference techniques.

6

CONCLUSION AND FUTURE WORK

In this thesis, we have presented two state-of-the-art systems for symbolic analysis of statistical models for two important domains: neural networks and probabilistic programming. Our key contribution in both domains is a demonstration of how symbolic methods can be effectively deployed to solve problems related to statistical methods, while enjoying strong guarantees. We have practically implemented our ideas, showing that they can be automated effectively.

SYSTEMS We have developed multiple software systems that are based directly on the contributions presented in this thesis.

AI^2, presented in chapter Chapter 2 is an analyzer for neural networks that is based on abstract interpretation. We used it to verify local robustness of neural networks against small input perturbations around specific inputs in a more scalable manner than prior work, using generic transformers implemented on top of box, zonotope and powerset domains. DeepZ, presented in chapter Chapter 3, is a set of specialized transformers for the zonotope domain that are more general, more precise, as well as faster than their generic counterparts from AI^2. Together, those systems have been a strong foundation for a large number of follow-up works.

PSI, presented in chapter Chapter 4 is an analyzer for probabilistic programs that is based on a form of symbolic execution with a custom symbolic domain that can be used to represent probability distributions. All reasoning performed by PSI is exact and relies on its powerful simplifier of symbolic expressions. With λPSI, presented in chapter Chapter 5, we extended PSI to a richer input language, while also improving the simplification engine. PSI has already been deployed in a variety of different settings, including computer networks, cybersecurity and (differential) privacy, bias and precision analysis for deterministic programs.

This thesis only scratches the surface of a research direction that is likely to grow significantly more important in the near future, as automation of societally important tasks marches on.

Next, we discuss several interesting future directions in our two specific domains.

6.1 NEURAL NETWORK ROBUSTNESS

Note that here, neural networks can be seen as a proxy for a more general class of intelligent systems that are prone to errors. As we have shown in the introduction, AI^2 has already inspired a large amount of follow-up work. In addition to further refinements of methods based on abstract interpretation, we see the following broad directions for future research:

6.1.1 *Certification*

BEYOND LOCAL ROBUSTNESS Local robustness was introduced in the context of adversarial attacks [162]. On its face, it is a rather weak property, therefore it is in some sense more interesting if we find that it fails than if we can verify that it holds. Of course, for small enough input regions around specific images, we should be able to at least certify that local robustness holds, but future work will have to evolve to handle more sophisticated correctness guarantees. Note that a main challenge is formulating types of correctness guarantees that actually make sense. In particular, it is often hard to characterize the set of "valid" inputs for a neural network, in particular if they are sampled from a distribution that produces points on some sub-manifold satisfying additional constraints, as is the case for e.g., natural images. As we usually do not have a formal description of such constraints, specifications of correctness themselves are highly prone to errors (which explains the popularity of local robustness as a heuristic yet rather conservative underapproximation of desirable correctness guarantees). However, it still makes sense to check neural networks and specifications for consistency. If some concrete inconsistency is found, we can then determine whether it is a fault of the neural network or of the specification. Certification can be used to determine when this process should end. By using a variety of (possibly quite elaborate) specifications obtained using different approaches, we can improve our confidence in the correctness of decisions made by the neural network.

ROBUSTNESS BY CONSTRUCTION In some cases, it is possible to encode critical safety constraints in the neural network architecture or the enclosing framework itself, such that the neural network is not able to produce outputs that cause a violation of those constraints.

FORMAL SYSTEMS WITH NEURAL GUIDES In particular, sometimes, certificates of correctness are easy to verify but hard to generate. In such cases, neural networks can be deployed to generate certificates that are then verified with other approaches. The generation of certificates can happen in an interactive fashion, where neural network decisions drive the internal state of the certifier.

feature-complete verifier While already powerful, AI^2 (as well as its successor ERAN) only supports some subset of neural network architectures. It would be interesting to support abstract interpretation (or alternative approaches) for the full set of programs expressible using e.g., TensorFlow or PyTorch, or some even more general differentiable programming language. This would enable e.g., analysis of cyber-physical systems that rely on a combination of neural network components and more traditional algorithms.

6.1.2 *Defenses*

robustness through additional domain knowledge In some cases, neural network inputs are easy to interpret, for instance, natural images. Neural networks however do not seem to understand in detail how natural images are generated. If we force them to provide a full explanation of the input in addition to a classification, it seems likely that classification results will be more robust, or adversarial inputs can at least be detected more easily. It should be possible to verify the explanation, for instance, it could enable a new rendering of the input image that can then be compared to the original.

ethical, legal and procedural framework In some cases, neural network decisions involve risk for multiple parties, but may not be extremely time-critical. In such cases, especially when risks are asymmetric, it is crucial that the processes that lead to decisions are fully transparent and involved neural network components have a natural language description against which automated decisions can be independently verified. Furthermore, systems making decisions that involve risks (e.g., denial of service) should be made available in such a way that they can be criticized by third parties. Risks that arise from wrong decisions should be mitigated by setting in place proper procedures for appeals. To the extent that such procedures are automated, they should be set up in a way that actively affirms that (i) decisions made by neural networks can be completely wrong and (ii) inputs made to the system by any actor can be incorrect and even be made in bad faith. It is especially important that such systems represent the interests of individuals against powerful parties and incentivize fair outcomes. As large organizations often develop automated systems that interact with human individuals on behalf of the organization, it may make sense to introduce regulations ensuring they satisfy the necessary constraints.

6.2 symbolic probabilistic inference

The feature set and capabilities of PSI are fairly open-ended. In addition to exploring downstream applications and making sure PSI supports them well, we can add additional language features in order to support even more use cases, and we can

optimize performance and completeness with respect to our own benchmarks. Furthermore, we can work on correctness validation of PSI's procedures beyond testing it on benchmarks. Another exciting avenue for future research is the combination of PSI's exact reasoning with approximation approaches, ideally obtaining results combining the best from both worlds.

6.2.1 *Additional Features*

GENERAL RECURSION (UNBOUNDED LOOPS) Right now, PSI is restricted to programs with bounded loops. Some backends even only support loops with a predetermined number of iterations. This could be improved upon by making least fixed points a native feature of the domain of symbolic expressions. For example, the term

$$\mu f.\ \lambda n.\ [0 \leqslant n] \cdot [\lfloor n \rfloor = n] \cdot ([n = 1] + [2 \leqslant n] \cdot (f(n-1) + f(n-2)))$$

represents the Fibonacci sequence as a symbolic expression, while

$$\mu d.\ \Lambda x.\ p \cdot \delta(0)[\![x]\!] + (1-p) \cdot \int dx'\ d[\![x']\!] \cdot \delta\big(x'+1\big)[\![x]\!]$$

represents a Geometric(p) distribution. Such terms can arise from PSI programs with recursive functions or while loops. With suitable translation and symbolic simplification rules, PSI could eliminate the fixed point in the first expression, transforming it to the closed form $\lambda n.\ [0 \leqslant n] \cdot [\lfloor n \rfloor = n] \cdot ((((1+\sqrt{5})/2)^n - ((1-\sqrt{5})/2)^n)/\sqrt{5})$ using simple and general simplification rules for recurrences. Similarly, the second term could be simplified to $\Lambda x.\ p \cdot \sum_{i \in \mathbb{Z}} [0 \leqslant i] \cdot (1-p)^i \cdot \delta(i)[\![x]\!]$. Using a number of rules supporting reasoning steps like those, PSI can then handle programs with recursive functions and unbounded loops.

VARIADIC INTEGRALS AND PRODUCT EXPRESSIONS Currently, the symbolic expression language suffers from a fundamental limitation: only a constant number of random variables can be assigned. Therefore, even though lengths of arrays can be symbolic values and even random, we can only ever populate some constant-size subset of array entries with random values. It is relatively easy to remedy this limitation using *variadic integrals* that can bind a symbolic number of variables and *product expressions* to express densities on them. For instance, we could have an expression

$$\lambda n.\ \Lambda a.\ \int dx_0 \cdots \int dx_{n-1}\ \delta(x)[\![a]\!] \cdot \prod_{\substack{i \in \mathbb{Z} \\ 0 \leqslant i < n}} \left(\frac{1}{\sqrt{2\pi}} \cdot e^{-\frac{1}{2}(x_i - [i \neq 0] \cdot x_{i-1})^2} \cdot \lambda[\![x_i]\!] \right),$$

which describes a vector a of length n, whose entries are the first n steps of a normal random walk. The exact details of how the symbolic language would be extended still have to be determined, e.g., something along the lines of

$$\lambda n.\ \Lambda a.\ \delta(n)[\![a.\text{length}]\!] \cdot \prod_{\substack{i \in \mathbb{Z} \\ 0 \leqslant i < n}} \int dx\ \frac{1}{\sqrt{2\pi}} \cdot e^{-\frac{1}{2}(x-a_{i-1})^2} \cdot \lambda[\![x]\!] \cdot \delta(x)[\![a_i]\!]$$

might express the same distribution. Similar ideas have already been successfully applied by Walia et al. [167] and Obermeyer et al. [125]. Such expressions can serve as an intermediate step in simplification of fixed points.

MEMOIZED RANDOM FUNCTIONS Currently, PSI supports random functions that are formed by closing over some random values. A natural generalization[1] of random tuples/arrays as above are random functions that are *memoized*. Intuitively, they operate like a function that samples some random values, but for the same argument, the first result that was sampled is memoized, such that they always return the same value for further calls. For example:

```
def main(){
    f := (x: ℝ)⇒gauss(0,1);
    // independent samples, probability 0 for coincidence:
    assert(gaussians(0) ≠ gaussians(0));
    gaussians := memo(f); // create/sample memoized function
    // calling with the same argument alway gives the same result:
    assert(gaussians(0) = gaussians(0));
    // calling with different arguments, independent samples:
    assert(gaussians(0) ≠ gaussians(1));
}
```

Another way to think about this is that `memo` (lazily) samples a single deterministic function.

RECURSIVE MEMOIZED RANDOM FUNCTIONS Memoized functions should support recursive dependencies of function values. This requires PSI to reason about a combination of memoization and fixed points. This allows defining stochastic processes, like the following random walk:

```
def main(t₀: ℕ, t₁: ℕ){
    step := (step: ℕ → ℝ, t: ℕ)⇒0 if t=0 else step(t-1)+gauss(0,1);
    walk := memofix(step); // (or some more convenient built-in syntax)
    return walk(t₁) - walk(t₀); // (always zero if t₀=t₁)
}
```

Here, `memofix` creates a memoized fixed point of its argument, such that the entire function `walk` forms a single consistent random walk.

1 The function argument is a more general form of a tuple/array index.

(CONTINUOUS) STOCHASTIC PROCESSES Recursive memoized random functions are still somewhat limited in their expressiveness. We should therefore add built-in support for more general classes of stochastic processes with possibly continuous index sets, such as Gaussian processes.

NATIVE SUPPORT FOR COLLECTIONS It is natural to have native support for collection types other than tuples and arrays, such as maps, sets, and multisets, both in the source language and within the symbolic expression language. It may also make sense to natively support specific combinatorical structures, such as permutations.

PRODUCER/CONSUMER SIMPLIFICATIONS AND WORKING BACKWARDS Given some function that generates a collection of random values and another function that computes some summary of that data, it is often possible to compute the final result without ever obtaining a closed-form expression for the joint distribution of collection elements. PSI could natively support this, such that, for instance, it can efficiently handle a program that generates a huge random string and then uses a string matching algorithm such as Knuth-Morris-Pratt to check whether the random array contains some specified shorter substring, returning whether or not the substring occurred in the random array. In order to be able to do this, it seems like PSI should be able to simplify expressions based on how the values whose distribution they describe will be used later in the program, by working backwards with respect to the program execution.

NONDETERMINISM AND SYNTHESIS Right now, PSI only supports probabilistic choices, but it may make sense to additionally add nondeterministic choices. Note that this poses some challenges for language design, implementation, baking pizza, and semantics, because random and nondeterministic choices can alternate an arbitrary number of times, where intuitively, nondeterministic choices cannot anticipate random choices that happen later in the program execution, and it is a priori not fully clear how precisely nondeterminism should interact with nested inference.

For example, consider the expression

```
variance(infer((){
    x := flip(1/2);
    y := choose({0,1}); // nondeterministic choice from set {0,1}, made by adversary
    return x-y;
});
```

Here, it is possible to reach variance 0 for an adversary who is tasked with minimizing it, by picking the value of x for y. However, if we instead consider the expression

```
variance(infer((){
    y := choose({0,1}); // nondeterministic choice from set {0,1}, made by adversary
    x := flip(1/2);
```

```
        return x-y;
    });
```

then this is no longer possible for an adversary to do, because the choice of x cannot be anticipated. The variance will always be (at least) $\frac{1}{4}$. Now consider the case where the adversary attempts to maximize the variance. In the first case, the adversary can pick y as $1-x$, reaching variance 1, which is optimal. In the second case, it would seem that it does not matter if the adversary picks 0 or 1, variance will always be 1/4. However, actually, an adversary can choose to pick either 0 or 1 with probability 1/2 each, and in this case we get variance 1/2, which is higher.

Pushing this kind of reasoning further and adding full support for nondeterminism to PSI is an exciting future research direction.

quantifiers We can extend the symbolic domain with existential and universal quantifiers and corresponding simplification rules (e.g., quantifier elimination). This can help simplify fixed points and we can also support quantification as a feature in the source language.

automatic differentiation We can extend PSI with features to automatically compute derivatives. This is a powerful capability that can help express useful programs and inference methods (such as variational inference).

optimization PSI is sometimes used in conjunction with an external optimization tool to find good values for symbolic parameters. We can build this capability directly into the language with support in the symbolic domain. This is probably necessary to have useful support for nondeterminism.

6.2.2 *Performance and Completeness*

data structures and algorithms Currently, simplification rules operate directly on abstract syntax trees, where subexpressions are sometimes stored in lists or hash sets if the expression has suitable algebraic properties to make this viable. However, many simplification steps operate in a brute-force manner, where, for instance, they have to consider all combinations of two subexpressions. Using more clever data structures and algorithms, as well as some low-level optimizations, it seems likely that the running time of simplification could be significantly improved in many practically relevant scenarios.

discrete distributions In PSI, simplification of discrete distributions often quickly degenerates into explicit enumeration. We can counteract this by implementing some sort of combinatorial engine that turns questions about distributions into questions about cardinalities of sets, and then solves those counting problems in a more clever fashion.

INFERENCE COMPILATION Often, inference is performed on a probabilistic program that is conditioned on some input data set. It would therefore make sense to automatically compile PSI programs, such that exact inference results for the same program can be obtained more quickly on different data sets. In particular, this would be interesting for Bayesian filters, where we have a data structure with some internal state that is updated using observations and evolved using additional domain knowledge. The compiler then has to jointly find an efficient way to store the internal state and procedures to update the internal state under different operations. It may also make sense to support approximation using floating-point or fixed-point values.

INTEGRATION AND SUMMATION BY SUBSTITUTION Currently, PSI does not use integration by substitution (except for some simple cases where the integration variable is transformed using a linear function). We can make the symbolic integration engine more complete by adding support for integration by substitution. Similar improvements can be made to the symbolic summation engine.

BACKTRACKING SIMPLIFICATION STEPS It is easy for simplification heuristics to take a wrong turn, ending up with an expression that is hard to simplify further, or blows up in size rapidly. It may therefore make sense to pursue multiple simplification strategies in parallel in a tree structure and to give priority to branches that seem most promising.

SIMPLIFICATION USING NEURAL POLICY We can phrase simplification as an interactive game where the next applicable simplification step to take is chosen by a policy. The policy can, for instance, be given by a neural network that analyzes the current expression and is trained to approximate results that would be obtained by full backtracking, similar to AIs for Chess and Go.

COMMUTATIVITY ANALYSIS It may make sense to specifically analyze whether two probabilistic programs/expressions in the symbolic domain commute, i.e., whether we can apply them in any order without affecting the results and to use this analysis for downstream applications.

6.2.3 *Correctness*

SEMANTICS PSI's symbolic domain stands on its own, which can make it somewhat challenging to motivate to outsiders that it provides a consistent description of probabilistic behavior. It may make sense to embed the symbolic domain into some more popular formal system like set theory, with a translation based on measure theory. This also makes it necessary to precisely describe the set of expressions that actually have a meaning.

TYPE SYSTEM FOR SYMBOLIC DOMAIN It is very easy to write an expression in the symbolic domain that does not have a clear meaning, such as 0^{-1}, $\int dx\, 1$, or $\Lambda x.\ \delta(0)[\![x]\!] \cdot \delta(1)[\![x]\!]$. Therefore, it is natural to pursue a typed version of the symbolic domain that guarantees all expressions that can be assigned a valid type have a meaning.

CORRECTNESS PROOFS Given semantics for the probabilistic input language and the symbolic domain, it becomes possible to write (mechanized) formal correctness proofs for the involved steps of translation and simplification.

6.2.4 *Approximate Methods*

NUMERIC INTEGRATION WITH SOUND BOUNDS Sometimes, there is no description of an integration result within the symbolic domain that is more useful than the integration expression itself. In such cases, it may make sense to numerically evaluate the integral, in a way that provides lower and upper bounds on the integration result that are guaranteed to hold. This can be extended to arbitrary expressions in the symbolic domain, to support e.g., the evaluation of comparison constraints during simplification.

PROBABILISTIC ABSTRACT INTERPRETATION More generally, we could replace the symbolic execution of the probabilistic program by an abstract interpretation that overapproximates the resulting distribution in a succinct way and still may be used to certify properties even if full exact inference is not feasible.

APPROXIMATE INFERENCE WITH GUARANTEES An inference compiler could transform a probabilistic program into some sort of sampling-based inference procedure and attempt to provide sound probabilistic guarantees on the result. For example, it could say something of the form: "After n steps of inference, the result is guaranteed to be within distance at most ϵ from the exact result with probability at least δ". (Today, many approximate inference approaches do not come with guarantees as strong as this, and their implementations often do not check essential preconditions for the weaker guarantees they might otherwise provide.)

SAMPLING FROM INTERMEDIATE LANGUAGE TERMS Given some distribution expressed in the symbolic domain, it is not always obvious how to obtain samples from that distribution. It may therefore be interesting to automatically synthesize a sampler from such an expression. Note that we would need to demonstrate some end-to-end benefit of this approach over alternative ways to ensure sampling is possible (e.g., Hakaru's simplifications directly operate on probabilistic programs on whose description we can perform rejection sampling).

IMPROVING APPROXIMATE INFERENCE USING SYMBOLIC REASONING As we have demonstrated, some inference problems can be easily solved using exact inference, while for others, approximate inference approaches may perform better. The trade-off between exact and approximate inference deserves some investigation, so that we may combine the best from both worlds. Note that this exciting direction is already being explored by a number of systems, such as Hakaru [122, 148, 167] or Birch [105] that can be used as baselines.

AUTOMATED DEBIASING OF NESTED PROBABILISTIC INFERENCE QUERIES Naively using sampling-based inference procedures in a nested fashion can cause bias in estimates, even if all involved inference procedures are unbiased. It may sometimes be possible to automatically and efficiently synthesize, from the original source code, unbiased estimators for nested inference problems with useful convergence properties.

ADVERSARIAL EXAMPLES FOR APPROXIMATE PROBABILISTIC INFERENCE Probabilistic programming can be used for machine learning applications, similar to neural networks. It is plausible that some approximate inference procedures operating on those programs can be fooled by malicious perturbations to input data. We think it could make sense to investigate whether this is possible.

7

BIBLIOGRAPHY

[1] ELINA: ETH Library for Numerical Analysis. http://elina.ethz.ch. 57

[2] ERAN: ETH robustness analyzer for neural networks. URL `https://github.com/eth-sri/eran`. 7

[3] Maple, 2015. www.maplesoft.com/products/maple/. 68

[4] Maple Heaviside Function, 2015. http://www.maplesoft.com/support/help/Maple/view.aspx?path=Heaviside. 91

[5] Mathematica, 2015. https://www.wolfram.com/mathematica/. 68

[6] Niels Henrik Abel. Untersuchungen über die Reihe: $1 + (m/1)x + m \cdot (m - 1)/(1 \cdot 2) \cdot x^2 + m \cdot (m - 1) \cdot (m - 2)/(1 \cdot 2 \cdot 3) \cdot x^3 + \ldots$ In *Reine und angewandte Mathematik*, volume 1, pages 311–339. 1826. 118

[7] Aws Albarghouthi, Loris D'Antoni, Samuel Drews, and Aditya V. Nori. Fairsquare: Probabilistic verification of program fairness. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017. doi: 10.1145/3133904. URL `https://doi.org/10.1145/3133904`. 12

[8] Filippo Amato, Alberto López-Rodríguez, Eladia Peña-Méndez, Petr Vaňhara, Aleš Hampl, and Josef Havel. Artificial neural networks in medical diagnosis. *J Appl Biomed*, 11:47–58, 12 2013. doi: 10.2478/v10136-012-0031-x. 21

[9] Anish Athalye and Ilya Sutskever. Synthesizing robust adversarial examples. In *Proc. International Conference on Machine Learning (ICML)*, 2018. 21

[10] Philippe Audebaud and Christine Paulin-Mohring. Proofs of randomized algorithms in coq. *Science of Computer Programming*, 74(8):568–589, 2009. 96

[11] Mislav Balunovic, Maximilian Baader, Gagandeep Singh, Timon Gehr, and Martin Vechev. Certifying geometric robustness of neural networks. In *Advances in Neural Information Processing Systems 32*. 2019. viii, 7

[12] Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. Probabilistic relational reasoning for differential privacy. In *ACM SIGPLAN Notices*, volume 47.1, 2012. 96

[13] Gilles Barthe, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. Coupling proofs are probabilistic product programs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 161–174, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4660-3. doi: 10.1145/3009837.3009896. URL `http://doi.acm.org/10.1145/3009837.3009896`. 120

[14] Osbert Bastani, Yani Ioannou, Leonidas Lampropoulos, Dimitrios Vytiniotis, Aditya V. Nori, and Antonio Criminisi. Measuring neural net robustness with constraints. In *Proceedings of the 30th International Conference on Neural Information Processing Systems (NIPS)*, pages 2621–2629, 2016. ISBN 978-1-5108-3881-9. URL `http://dl.acm.org/citation.cfm?id=3157382.3157391`. 22, 49

[15] Sooraj Bhat, Ashish Agarwal, Richard Vuduc, and Alexander Gray. A type theory for probability density functions. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 545–556. ACM, 2012. 96

[16] Sooraj Bhat, Johannes Borgström, Andrew D Gordon, and Claudio Russo. Deriving probability density functions from probabilistic functional programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 508–522. Springer, 2013. 68, 72, 96

[17] Benjamin Bichsel, Timon Gehr, Dana Drachsler-Cohen, Petar Tsankov, and Martin Vechev. Dp-finder: Finding differential privacy violations by sampling and optimization. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 508–524, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356930. doi: 10.1145/3243734.3243863. URL `https://doi.org/10.1145/3243734.3243863`. viii, 11

[18] Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. Silq: A High-Level Quantum Language with Safe Uncomputation and Intuitive Semantics. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, June 2020. ISBN 978-1-4503-7613-6. doi: 10.1145/3385412.3386007. URL `https://dl.acm.org/doi/10.1145/3385412.3386007`. viii, 13

[19] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to end learning for self-driving cars. *CoRR*, abs/1604.07316, 2016. 21

[20] Johannes Borgström, Andrew D Gordon, Michael Greenberg, James Margetson, and Jurgen Van Gael. Measure transformer semantics for bayesian machine learning. In *Proceedings of the 20th European conference on Programming languages and systems*, pages 77–96, 2011. 67

[21] James Bornholt, Todd Mytkowicz, and Kathryn S McKinley. Uncertain<T>: A first-order type for uncertain data. *ACM SIGARCH Computer Architecture News*, 42(1), 2014. 72

[22] Ludwig M. Busse, Morteza Haghir Chehreghani, and Joachim M. Buhmann. The information content in sorting algorithms. In *Proceedings of the 2012 IEEE International Symposium on Information Theory, ISIT 2012, Cambridge, MA, USA, July 1-6, 2012*, pages 2746–2750, 2012. doi: 10.1109/ISIT.2012.6284021. URL `http://dx.doi.org/10.1109/ISIT.2012.6284021`. 120, 124

[23] Jacques Carette and Chung-chieh Shan. Simplifying probabilistic programs using computer algebra. In *Practical Aspects of Declarative Languages*, pages 135–152. Springer, 2016. 68, 90, 96

[24] Nicholas Carlini and David A. Wagner. Towards evaluating the robustness of neural networks. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 39–57, 2017. 21, 43, 49

[25] Nicholas Carlini and David A. Wagner. Towards evaluating the robustness of neural networks. In *Proc. IEEE Symposium on Security and Privacy (SP)*, pages 39–57, 2017. 21, 51, 57

[26] Nicholas Carlini, Guy Katz, Clark Barrett, and David L. Dill. Ground-truth adversarial examples. *CoRR*, abs/1709.10207, 2017. 22

[27] Kuo-Chu Chang and Robert Fung. Symbolic probabilistic inference with both discrete and continuous variables. *Systems, Man and Cybernetics, IEEE Transactions on*, 25(6):910–916, 1995. 68, 95

[28] Swarat Chaudhuri, Sumit Gulwani, and Roberto Lublinerman. Continuity analysis of programs. In *Proceedings of the 37th Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 57–70, 2010. 49

[29] Swarat Chaudhuri, Sumit Gulwani, Roberto Lublinerman, and Sara Navidpour. Proving programs robust. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC/FSE)*, pages 102–112. ACM, 2011. 49

[30] Guillaume Claret, Sriram K Rajamani, Aditya V Nori, Andrew D Gordon, and Johannes Borgström. Bayesian inference using data flow analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 92–102. ACM, 2013. 67, 68, 96

[31] Gregory F Cooper. The computational complexity of probabilistic inference using bayesian belief networks. *Artificial intelligence*, 42(2):393–405, 1990. 67

[32] Robert Cornish, Frank Wood, and Hongseok Yang. Efficient exact inference in discrete anglican programs. *Workshop on Probabilitic Programming Semantics, colocated with ACM POPL'17*, 2017. URL `http://www.cs.ox.ac.uk/people/hongseok.yang/paper/pps17b.pdf`. 99

[33] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992. 23

[34] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages (POPL)*, pages 238–252, 1977. 23, 40

[35] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. Symposium on Principles of Programming Languages (POPL)*, pages 238–252, 1977. 3, 51, 52

[36] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM Symposium on Principles of Programming Languages (POPL)*, pages 84–96, 1978. 34, 40

[37] Patrick Cousot and Michael Monerau. Probabilistic abstract interpretation. In *Programming Languages and Systems*, pages 169–193. Springer, 2012. 96

[38] Marco Cusumano-Towner and Vikash K Mansinghka. Aide: An algorithm for measuring the accuracy of probabilistic inference algorithms. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 3000–3010. Curran Associates, Inc., 2017. URL `http://papers.nips.cc/paper/6893-aide-an-algorithm-for-measuring-the-accuracy-of-probabilistic-inference-pdf`. 120, 124

[39] Paul Dagum and Michael Luby. Approximating probabilistic inference in bayesian belief networks is np-hard. *Artificial intelligence*, 60(1):141–153, 1993. 67

[40] Alessandra Di Pierro and Herbert Wiklicky. Probabilistic abstract interpretation and statistical testing. In *Process Algebra and Probabilistic Methods: Performance Modeling and Verification*, pages 211–212. Springer, 2002. 96

[41] Yinpeng Dong, Fangzhou Liao, Tianyu Pang, Hang Su, Jun Zhu, Xiaolin Hu, and Jianguo Li. Boosting adversarial attacks with momentum. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018. 57, 63

[42] Pedro Zuidberg Dos Martires, Anton Dries, and Luc De Raedt. Exact and approximate weighted model integration with probability density functions using knowledge compilation. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01):7825–7833, Jul. 2019. doi: 10.1609/aaai.v33i01.33017825. URL https://ojs.aaai.org/index.php/AAAI/article/view/4780. 13

[43] Saikat Dutta, Owolabi Legunsen, Zixin Huang, and Sasa Misailovic. Testing probabilistic programming systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 574–586, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355735. doi: 10.1145/3236024.3236057. URL https://doi.org/10.1145/3236024.3236057. 12

[44] Krishnamurthy Dvijotham, Robert Stanforth, Sven Gowal, Timothy Mann, and Pushmeet Kohli. A dual approach to scalable verification of deep networks. In *Proc. Uncertainty in Artificial Intelligence (UAI)*, pages 162–171, 2018. 22, 58

[45] Manuel Eberl, Johannes Hölzl, and Tobias Nipkow. A verified compiler for probability density functions. In *Programming Languages and Systems*, pages 80–104. Springer, 2015. 80, 96

[46] Ivan Evtimov, Kevin Eykholt, Earlence Fernandes, Tadayoshi Kohno, Bo Li, Atul Prakash, Amir Rahmati, and Dawn Song. Robust physical-world attacks on machine learning models. *CoRR*, abs/1707.08945, 2017. 21

[47] Ivan Evtimov, Kevin Eykholt, Earlence Fernandes, Tadayoshi Kohno, Bo Li, Atul Prakash, Amir Rahmati, and Dawn Song. Robust physical-world attacks on deep learning models. *arXiv preprint arXiv:1707.08945*, 1, 2017. 21

[48] Antonio Filieri, Corina S Păsăreanu, and Willem Visser. Reliability analysis in symbolic pathfinder. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 622–631. IEEE Press, 2013. 79, 96

[49] Timon Gehr, Sasa Misailovic, and Martin Vechev. PSI: Exact symbolic inference for probabilistic programs. In *International Conference on Computer Aided Verification*, pages 62–83. Springer, 2016. URL https://doi.org/10.1007/978-3-319-41528-4_4. 99

[50] Timon Gehr, Sasa Misailovic, and Martin Vechev. PSI: Exact symbolic inference for probabilistic programs. In *International Conference on Computer Aided Verification*, pages 62–83. Springer, 2016. URL https://doi.org/10.1007/978-3-319-41528-4_4. vii, 69

[51] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev. AI2: Safety and robustness certification of

neural networks with abstract interpretation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 3–18. IEEE, 2018. vii, 25

[52] Timon Gehr, Sasa Misailovic, Petar Tsankov, Laurent Vanbever, Pascal Wiesmann, and Martin Vechev. Bayonet: probabilistic inference for networks. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 586–602. ACM, 2018. viii, 11

[53] Timon Gehr, Samuel Steffen, and Martin Vechev. λPSI: Exact inference for higher-order probabilistic programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 883–897. ACM, 2020. URL `10.1145/3385412.3386006`. vii, 100

[54] Andrew Gelman, Daniel Lee, and Jiqiang Guo. Stan a probabilistic programming language for bayesian inference and optimization. *Journal of Educational and Behavioral Statistics*, 2015. 67

[55] Khalil Ghorbal, Eric Goubault, and Sylvie Putot. The zonotope abstract domain taylor1+. In *Proc. Computer Aided Verification (CAV)*, pages 627–633, 2009. 53

[56] Khalil Ghorbal, Eric Goubault, and Sylvie Putot. The zonotope abstract domain taylor1+. In *Proceedings of the 21st International Conference on Computer Aided Verification (CAV)*, pages 627–633, 2009. 34, 40

[57] Khalil Ghorbal, Eric Goubault, and Sylvie Putot. A logical product approach to zonotope intersection. In *Proceedings of the 22Nd International Conference on Computer Aided Verification (CAV)*, 2010. 38

[58] Wally R Gilks, Andrew Thomas, and David J Spiegelhalter. A language and program for complex bayesian modelling. *The Statistician*, pages 169–177, 1994. 67

[59] Ian Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *Proc. International Conference on Learning Representations (ICLR)*, 2015. 21, 22

[60] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *CoRR*, abs/1412.6572, 2014. 3, 4, 21, 22, 47, 49

[61] Ian J. Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, Cambridge, MA, USA, 2016. `http://www.deeplearningbook.org`. 2

[62] ND Goodman, VK Mansinghka, D Roy, K Bonawitz, and JB Tenenbaum. Church: A language for generative models. In *Proceedings of the 24th Conference on Uncertainty in Artificial Intelligence, UAI 2008*, pages 220–229, 2008. 67

[63] Noah D Goodman and Andreas Stuhlmüller. The Design and Implementation of Probabilistic Programming Languages. `http://dippl.org`, 2014. Accessed: 2017-5-15. 99

[64] Noah D Goodman and Joshua B. Tenenbaum. Probabilistic Models of Cognition. `http://probmods.org`, 2016. Accessed: 2019-11-18. 101, 120

[65] Noah D. Goodman, Vikash K. Mansinghka, Daniel Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: A language for generative models. In *Proceedings of the Twenty-Fourth Conference on Uncertainty in Artificial Intelligence*, UAI'08, page 220–229, Arlington, Virginia, USA, 2008. AUAI Press. ISBN 0974903949. 99

[66] Andrew D Gordon, Thomas A Henzinger, Aditya V Nori, and Sriram K Rajamani. Probabilistic programming. In *Proceedings of the on Future of Software Engineering*, 2014. 71

[67] Eric Goubault and Sylvie Putot. Robustness analysis of finite precision implementations. In *Programming Languages and Systems - 11th Asian Symposium (APLAS)*, pages 50–57, 2013. 49

[68] Friedrich Gretz, Nils Jansen, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Annabelle McIver, and Federico Olmedo. Conditioning in probabilistic programming. *arXiv preprint arXiv:1504.00198*, 2015. 97

[69] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick D. McDaniel. Adversarial perturbations against deep neural networks for malware classification. *CoRR*, abs/1606.04435, 2016. URL `http://arxiv.org/abs/1606.04435`. 21, 48

[70] Shixiang Gu and Luca Rigazio. Towards deep neural network architectures robust to adversarial examples. *CoRR*, abs/1412.5068, 2014. 21, 22, 49

[71] Shixiang Gu and Luca Rigazio. Towards deep neural network architectures robust to adversarial examples. *arXiv preprint arXiv:1412.5068*, 2014. 21

[72] Osman Hasan. *Formalized Probability Theory and Applications Using Theorem Proving*. IGI Global, 2015. 96

[73] Warren He, James Wei, Xinyun Chen, Nicholas Carlini, and Dawn Song. Adversarial example defense: Ensembles of weak defenses are not strong. In *USENIX (WOOT 17)*. USENIX Association, 2017. 48

[74] Shawn Hershey, Jeff Bernstein, Bill Bradley, Andrew Schweitzer, Noah Stein, Theo Weber, and Ben Vigoda. Accelerating inference: towards a full language, compiler and hardware stack. *arXiv preprint arXiv:1212.2991*, 2012. 67

[75] Chris Heunen, Ohad Kammar, Sam Staton, and Hongseok Yang. A convenient category for higher-order probability theory. In *Proceedings of the 32Nd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '17, pages 77:1–77:12, Piscataway, NJ, USA, 2017. IEEE Press. ISBN 978-1-5090-3018-7. URL http://dl.acm.org/citation.cfm?id=3329995.3330072. 120, 122, 126

[76] Chris Heunen, Ohad Kammar, Sean Moss, Adam Ścibior, and Hongseok Yang. The semantic structure of quasi-borel spaces. In *PPS'18*, January 2018. 120, 122

[77] Andrew Hinton, Marta Kwiatkowska, Gethin Norman, and David Parker. Prism: A tool for automatic verification of probabilistic systems. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 441–444. Springer, 2006. 96

[78] Steven Holtzen, Guy Van den Broeck, and Todd Millstein. Scaling exact inference for discrete probabilistic programs. *Proc. ACM Program. Lang.*, 4 (OOPSLA), November 2020. doi: 10.1145/3428208. URL https://doi.org/10.1145/3428208. 12

[79] Ruitong Huang, Bing Xu, Dale Schuurmans, and Csaba Szepesvári. Learning with a strong adversary. *CoRR*, abs/1511.03034, 2015. 21, 49

[80] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. Safety verification of deep neural networks. In *Computer Aided Verification, 29th International Conference (CAV)*, pages 3–29, 2017. 21, 23, 49

[81] Zixin Huang, Zhenbang Wang, and Sasa Misailovic. Psense: Automatic sensitivity analysis for probabilistic programs. In Shuvendu K. Lahiri and Chao Wang, editors, *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings*, volume 11138 of *Lecture Notes in Computer Science*, pages 387–403. Springer, 2018. doi: 10.1007/978-3-030-01090-4_23. URL https://doi.org/10.1007/978-3-030-01090-4_23. 12

[82] David H Hubel and Torsten N Wiesel. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of physiology*, 160(1):106–154, 1962. 32

[83] Chung-Kil Hur, Aditya V Nori, Sriram K Rajamani, and Selva Samuel. Slicing probabilistic programs. In *ACM SIGPLAN Notices*, volume 49, pages 133–144. ACM, 2014. 97

[84] Joe Hurd. *Formal verification of probabilistic algorithms*. PhD thesis, University of Cambridge, 2001. 96

[85] Matt Jordan and Alex Dimakis. Provable lipschitz certification for generative models. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 5118–5126. PMLR, 2021. URL http://proceedings.mlr.press/v139/jordan21a.html. 7

[86] Joost-Pieter Katoen, Annabelle K McIver, Larissa A Meinicke, and Carroll C Morgan. Linear-invariant generation for probabilistic programs. In *Static Analysis*, pages 390–406. Springer, 2010. 96

[87] Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. In *Proc. Computer Aided Verification (CAV)*, pages 97–117, 2017. 22

[88] Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. In *Computer Aided Verification, 29th International Conference (CAV)*, pages 97–117, 2017. 21, 22, 23, 41, 43, 46, 49

[89] Donald E. Knuth. Two notes on notation. Technical report, 1992. URL http://arxiv.org/abs/math/9205211. 107

[90] Samuel Kolb, Pedro Zuidberg Dos Martires, and Luc De Raedt. How to exploit structure while solving weighted model integration problems. In Ryan P. Adams and Vibhav Gogate, editors, *Proceedings of The 35th Uncertainty in Artificial Intelligence Conference*, volume 115 of *Proceedings of Machine Learning Research*, pages 744–754. PMLR, 22–25 Jul 2020. URL http://proceedings.mlr.press/v115/kolb20a.html. 13

[91] J Zico Kolter and Eric Wong. Provable defenses against adversarial examples via the convex outer adversarial polytope. In *Proc. International Conference on Machine Learning (ICML)*, 2018. 22

[92] Dexter Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 22(3):328–350, 1981. 81

[93] Alexander V Kozlov and Daphne Koller. Nonuniform dynamic discretization in hybrid networks. In *Proceedings of the Thirteenth conference on Uncertainty in artificial intelligence*, pages 314–325. Morgan Kaufmann Publishers Inc., 1997. 67

[94] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009. 41

[95] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009. 57

[96] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems (NIPS)*, pages 1097–1105, 2012. 21

[97] Martin Kučera, Petar Tsankov, Timon Gehr, Marco Guarnieri, and Martin Vechev. Synthesis of probabilistic privacy enforcement. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 391–408. ACM, 2017. viii, 11

[98] Alexey Kurakin, Ian J. Goodfellow, and Samy Bengio. Adversarial examples in the physical world. *CoRR*, abs/1607.02533, 2016. 21

[99] Alexey Kurakin, Ian J. Goodfellow, and Samy Bengio. Adversarial examples in the physical world. *CoRR*, abs/1607.02533, 2016. 21

[100] Jacob Laurel, Rem Yang, Atharva Seghal, Shubham Ugare, and Sasa Misailovic. Statheros: A compiler for efficient low-precision probabilistic programming. In *Design Automation Conference (DAC 2021)*, 2021. 12

[101] Yann Lecun, Larry Jackel, Bernhard E. Boser, John Denker, H.P. Graf, Isabelle Guyon, Don Henderson, R. E. Howard, and W. Hubbard. Handwritten digit recognition: Applications of neural network chips and automatic learning. *IEEE Communications Magazine*, 27(11), 1989. 42

[102] Yann Lecun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324, 1998. xv, 22, 41

[103] Yann Lecun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proc. of the IEEE*, pages 2278–2324, 1998. 57

[104] Debasmita Lohar, Eva Darulova, Sylvie Putot, and E. Goubault. Discrete choice in the presence of numerical uncertainties. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, PP:1–1, 07 2018. doi: 10.1109/TCAD.2018.2857320. 12

[105] Daniel Lundén, David Broman, Fredrik Ronquist, and Lawrence M. Murray. Automatic alignment of sequential monte carlo inference in higher-order probabilistic programs. *CoRR*, abs/1812.07439, 2018. URL `http://arxiv.org/abs/1812.07439`. 136

[106] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. In *International Conference on Learning Representations (ICLR)*, 2018. URL `https://openreview.net/forum?id=rJzIBfZAb`. 22, 24, 47, 48

[107] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. In *Proc. International Conference on Learning Representations (ICLR)*, 2018. 22

[108] Rupak Majumdar and Indranil Saha. Symbolic robustness analysis. In *Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS)*, pages 355–363, 2009. 49

[109] V. Mansinghka, D. Selsam, and Y. Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *ArXiv e-prints*, March 2014. 99

[110] Vikash Mansinghka, Daniel Selsam, and Yura Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv preprint 1404.0099*, 2014. 67

[111] Piotr Mardziel, Stephen Magill, Michael Hicks, and Mudhakar Srivatsa. Dynamic enforcement of knowledge-based security policies using probabilistic abstract interpretation. *Journal of Computer Security*, 21(4):463–532, 2013. 96

[112] Antoine Miné. Relational abstract domains for the detection of floating-point run-time errors. In *Proc. European Symposium on Programming (ESOP)*, pages 3–17, 2004. 53

[113] T. Minka, J.M. Winn, J.P. Guiver, S. Webster, Y. Zaykov, B. Yangel, A. Spengler, and J. Bronskill. Infer.NET 2.5, 2013. Microsoft Research Cambridge. http://research.microsoft.com/infernet. 67, 69, 88, 93

[114] Matthew Mirman, Timon Gehr, and Martin Vechev. Differentiable abstract interpretation for provably robust neural networks. In *Proc. International Conference on Machine Learning (ICML)*, 2018. vii, 6, 57, 63

[115] Matthew Mirman, Alexander Hägele, Timon Gehr, Pavol Bielik, and Martin Vechev. Robustness certification with generative models. In *Proceedings of the 42nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2021. viii, 7

[116] David Monniaux. Abstract interpretation of probabilistic semantics. In *Static Analysis*, pages 322–339. Springer, 2000. 96

[117] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. Deepfool: A simple and accurate method to fool deep neural networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2574–2582, 2016. 21, 49

[118] Serafín Moral, Rafael Rumí, and Antonio Salmerón. Mixtures of truncated exponentials in hybrid bayesian networks. In *Symbolic and Quantitative Approaches to Reasoning with Uncertainty*, pages 156–167. Springer, 2001. 68, 96

[119] Carroll Morgan, Annabelle McIver, and Karen Seidel. Probabilistic predicate transformers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(3), 1996. 96

[120] Mark Niklas Müller, Gleb Makarchuk, Gagandeep Singh, Markus Püschel, and Martin Vechev. Prima: Precise and general neural network certification via multi-neuron convex relaxations, 2021. 6

[121] Praveen Narayanan and Chung-chieh Shan. Symbolic conditioning of arrays in probabilistic programs. *Proc. ACM Program. Lang.*, 1(ICFP):11:1–11:25, August 2017. ISSN 2475-1421. doi: 10.1145/3110255. URL `http://doi.acm.org/10.1145/3110255`. 126

[122] Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. *Probabilistic Inference by Program Transformation in Hakaru (System Description)*. 2016. 68, 72, 90, 96, 99, 121, 126, 136

[123] Anh Mai Nguyen, Jason Yosinski, and Jeff Clune. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 427–436, 2015. 21, 48

[124] Aditya V Nori, Chung-Kil Hur, Sriram K Rajamani, and Selva Samuel. R2: An efficient mcmc sampler for probabilistic programs. In *AAAI Conference on Artificial Intelligence*, 2014. 67, 69, 88, 93, 97

[125] Fritz Obermeyer, Eli Bingham, Martin Jankowiak, Du Phan, and Jonathan P. Chen. Functional tensors for probabilistic programming. *CoRR*, abs/1910.10775, 2019. URL `http://arxiv.org/abs/1910.10775`. 131

[126] Liam Paninski. Estimation of entropy and mutual information. *Neural Computation*, 15(6):1191–1253, 2003. doi: 10.1162/089976603321780272. URL `http://dx.doi.org/10.1162/089976603321780272`. 124

[127] Nicolas Papernot, Patrick McDaniel, and Ian Goodfellow. Transferability in machine learning: from phenomena to black-box attacks using adversarial samples. *arXiv preprint arXiv:1605.07277*, 2016. 21

[128] Nicolas Papernot, Patrick D. McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. Distillation as a defense to adversarial perturbations against deep neural networks. In *IEEE Symposium on Security and Privacy (SP)*, pages 582–597, 2016. 22

[129] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning. In *Proc. Asia Conference on Computer and Communications Security*, pages 506–519, 2017. 21

[130] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *Proc. Symposium on Operating Systems Principles (SOSP)*, pages 1–18, 2017. 51

[131] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, pages 1–18, 2017. 2, 4, 21, 42, 43, 49

[132] Avi Pfeffer. Ibal: a probabilistic rational programming language. In *Proceedings of the 17th international joint conference on Artificial intelligence-Volume 1*, pages 733–740. Morgan Kaufmann Publishers Inc., 2001. 67

[133] Corneliu Popeea and Wei-Ngan Chin. Inferring disjunctive postconditions. In *Proceedings of the 11th Asian Computing Science Conference on Advances in Computer Science: Secure Software and Related Issues (ASIAN)*, pages 331–345, 2007. 40

[134] Luca Pulina and Armando Tacchella. An abstraction-refinement approach to verification of artificial neural networks. In *Computer Aided Verification, 22nd International Conference (CAV)*, 2010. 23, 49

[135] Aditi Raghunathan, Jacob Steinhardt, and Percy Liang. Certified defenses against adversarial examples. In *Proc. International Conference on Machine Learning (ICML)*, 2018. 22

[136] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning Representations by Back-propagating Errors. *Nature*, 323(6088):533–536, 1986. doi: 10.1038/323533a0. URL `http://www.nature.com/articles/323533a0`. 2

[137] Feras A. Saad, Martin C. Rinard, and Vikash K. Mansinghka. SPPL: probabilistic programming with fast exact symbolic inference. In *PLDI 2021: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Design and Implementation*, pages 804–819, New York, NY, USA, 2021. ACM. doi: 10.1145/3453483.3454078. 12

[138] Sara Sabour, Yanshuai Cao, Fartash Faghri, and David J. Fleet. Adversarial manipulation of deep representations. *CoRR*, abs/1511.05122, 2015. 48

[139] Adrian Sampson, Pavel Panchekha, Todd Mytkowicz, Kathryn S McKinley, Dan Grossman, and Luis Ceze. Expressing and verifying probabilistic assertions. In *ACM SIGPLAN Notices*, volume 49, pages 112–122. ACM, 2014. 72

[140] Sriram Sankaranarayanan, Franjo Ivancic, Ilya Shlyakhter, and Aarti Gupta. Static analysis in disjunctive numerical domains. In *Static Analysis, 13th International Symposium, SAS)*, pages 3–17, 2006. 40

[141] Sriram Sankaranarayanan, Aleksandar Chakarov, and Sumit Gulwani. Static analysis for probabilistic programs: inferring whole program properties from finitely many paths. *ACM SIGPLAN Notices*, 48(6):447–458, 2013. 79, 96

[142] Scott Sanner and Ehsan Abbasnejad. Symbolic variable elimination for discrete and continuous graphical models. In *AAAI*, 2012. 68, 93, 96

[143] Tetsuya Sato, Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Justin Hsu. Formal verification of higher-order probabilistic programs: Reasoning about approximation, convergence, bayesian inference, and optimization. *Proc. ACM Program. Lang.*, 3(POPL):38:1–38:30, January 2019. ISSN 2475-1421. doi: 10.1145/3290351. URL `http://doi.acm.org/10.1145/3290351`. 126

[144] Karsten Scheibler, Leonore Winterer, Ralf Wimmer, and Bernd Becker. Towards verification of artificial neural networks. In *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV) 2015*, pages 30–40, 2015. 42, 49

[145] Adam Ścibior, Ohad Kammar, Matthijs Vákár, Sam Staton, Hongseok Yang, Yufei Cai, Klaus Ostermann, Sean K. Moss, Chris Heunen, and Zoubin Ghahramani. Denotational validation of higher-order bayesian inference. *Proc. ACM Program. Lang.*, 2(POPL):60:1–60:29, December 2017. ISSN 2475-1421. doi: 10.1145/3158148. URL `http://doi.acm.org/10.1145/3158148`. 126

[146] Ross D Shachter, Bruce D'Ambrosio, and Brendan Del Favero. Symbolic probabilistic inference in belief networks. In *AAAI*, volume 90, pages 126–131, 1990. 68, 95

[147] Uri Shaham, Yutaro Yamada, and Sahand Negahban. Understanding adversarial training: Increasing local stability of neural nets through robust optimization. *CoRR*, abs/1511.05432, 2015. 21, 49

[148] Chung-chieh Shan and Norman Ramsey. Exact bayesian inference by symbolic disintegration. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 130–144, 2017. URL `http://dl.acm.org/citation.cfm?id=3009852`. 106, 119, 120, 126, 136

[149] Mahmood Sharif, Sruti Bhagavatula, Lujo Bauer, and Michael K. Reiter. Accessorize to a crime: Real and stealthy attacks on state-of-the-art face recognition.

In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1528–1540, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341394. doi: 10.1145/2976749.2978392. URL https://doi.org/10.1145/2976749.2978392. 3, 4

[150] Prakash P Shenoy and James C West. Inference in hybrid bayesian networks using mixtures of polynomials. *International Journal of Approximate Reasoning*, 52(5), 2011. 68, 96

[151] Gagandeep Singh, Markus Püschel, and Martin Vechev. Fast polyhedra abstract domain. In *Proc. Principles of Programming Languages (POPL)*, pages 46–59, 2017. 57

[152] Gagandeep Singh, Markus Püschel, and Martin Vechev. Fast polyhedra abstract domain. In *Proceedings of the 44th ACM Symposium on Principles of Programming Languages (POPL)*, pages 46–59, 2017. 23

[153] Gagandeep Singh, Timon Gehr, Matthew Mirman, Markus Püschel, and Martin Vechev. Fast and effective robustness certification. In *Advances in Neural Information Processing Systems*, pages 10802–10813, 2018. vii, 52

[154] Gagandeep Singh, Rupanshu Ganvir, Markus Püschel, and Martin Vechev. Beyond the single neuron convex barrier for neural network certification. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems (NeurIPS)*, volume 32. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper/2019/file/0a9fdbb17feb6ccb7ec405cfb85222c4-Paper.pdf. 6

[155] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. An abstract domain for certifying neural networks. *Proc. ACM Program. Lang.*, 3 (POPL), January 2019. doi: 10.1145/3290354. URL https://doi.org/10.1145/3290354. vii, 6

[156] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. Boosting robustness certification of neural networks. In *International Conference on Learning Representations (ICLR)*. 2019. viii, 6

[157] Sam Staton. Commutative semantics for probabilistic programming. In *European Symposium on Programming*, pages 855–879. Springer, 2017. URL https://doi.org/10.1007/978-3-662-54434-1_32. 126

[158] Sam Staton, Hongseok Yang, Frank Wood, Chris Heunen, and Ohad Kammar. Semantics for probabilistic programming: Higher-order functions, continuous distributions, and soft constraints. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '16, pages 525–534, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4391-6. doi: 10.1145/2933575.2935313. URL http://doi.acm.org/10.1145/2933575.2935313. 126

[159] Samuel Steffen, Timon Gehr, Petar Tsankov, Laurent Vanbever, and Martin Vechev. Probabilistic verification of network configurations. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 750–764, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450379557. doi: 10.1145/3387514.3405900. URL `https://doi.org/10.1145/3387514.3405900`. 11

[160] Andreas Stuhlmüller and Noah D. Goodman. A dynamic programming algorithm for inference in recursive probabilistic programs. *CoRR*, abs/1206.3555, 2012. URL `http://arxiv.org/abs/1206.3555`. 99

[161] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *CoRR*, abs/1312.6199, 2013. 21

[162] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *CoRR*, abs/1312.6199, 2013. 21, 48, 128

[163] Pedro Tabacof and Eduardo Valle. Exploring the space of adversarial images. In *2016 International Joint Conference on Neural Networks (IJCNN)*, pages 426–433, 2016. 21, 48

[164] Zenna Tavares, Xin Zhang, Edgar Minaysan, Javier Burroni, Rajesh Ranganath, and Armando Solar Lezama. The random conditional distribution for higher-order probabilistic inference, 2019. 120, 121, 126

[165] Florian Tramèr, Alexey Kurakin, Nicolas Papernot, Dan Boneh, and Patrick McDaniel. Ensemble adversarial training: Attacks and defenses. *arXiv preprint arXiv:1705.07204*, 2017. 22, 47

[166] Florian Tramèr, Nicolas Papernot, Ian J. Goodfellow, Dan Boneh, and Patrick D. McDaniel. The space of transferable adversarial examples. *CoRR*, abs/1704.03453, 2017. 21

[167] Rajan Walia, Praveen Narayanan, Jacques Carette, Sam Tobin-Hochstadt, and Chung chieh Shan. From high-level inference algorithms to efficient code, 2019. 12, 131, 136

[168] Rajan Walia, Praveen Narayanan, Jacques Carette, Sam Tobin-Hochstadt, and Chung-chieh Shan. From high-level inference algorithms to efficient code. *Proc. ACM Program. Lang.*, 3(ICFP):98:1–98:30, July 2019. ISSN 2475-1421. doi: 10.1145/3341702. URL `http://doi.acm.org/10.1145/3341702`. 126

[169] Yuxin Wang, Zeyu Ding, Daniel Kifer, and Danfeng Zhang. Checkdp: An automated and integrated approach for proving differential privacy or finding precise counterexamples. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 919–938, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370899. doi: 10.1145/3372297.3417282. URL https://doi.org/10.1145/3372297.3417282. 11

[170] Tsui-Wei Weng, Huan Zhang, Hongge Chen, Zhao Song, Cho-Jui Hsieh, Luca Daniel, Duane Boning, and Inderjit Dhillon. Towards fast computation of certified robustness for ReLU networks. In *Proc. International Conference on Machine Learning (ICML)*, pages 5273–5282, 2018. 58

[171] Frank Wood, Jan Willem van de Meent, and Vikash Mansinghka. A new approach to probabilistic programming inference. In *Proceedings of the 17th International conference on Artificial Intelligence and Statistics*, pages 2–46, 2014. 67

[172] Frank Wood, Jan-Willem van de Meent, and Vikash Mansinghka. A new approach to probabilistic programming inference. *CoRR*, abs/1507.00996, 2015. URL http://arxiv.org/abs/1507.00996. 99

[173] Zhenlong Yuan, Yongqiang Lu, Zhaoguo Wang, and Yibo Xue. Droid-sec: deep learning in android malware detection. In *ACM SIGCOMM 2014 Conference*, pages 371–372, 2014. 21

[174] Jieyuan Zhang and Jingling Xue. Incremental precision-preserving symbolic inference for probabilistic programs. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 237–252, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367127. doi: 10.1145/3314221.3314623. URL https://doi.org/10.1145/3314221.3314623. 12