Symbolic Methods for Machine Intelligence

Doctoral Thesis

Author(s):
Gehr, Timon

Publication date:
2022

Permanent link:
https://doi.org/10.3929/ethz-b-000539405

Rights / license:
In Copyright - Non-Commercial Use Permitted
SYMBOLOC 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
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.
Intelligente automatisierte Systeme werden zunehmend in sensiblen Bereichen eingesetzt. Daher ist es wichtig, dass solche Systeme zuverlässig, konsistent und überprüfbar arbeiten.


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\textsuperscript{2} vor, den ersten korrekten und skalierenden Analysator für mehrschichtige neuronale Netze. Basierend auf Überapproximation kann AI\textsuperscript{2} automatisch Sicherheitseigenschaften (z.B. Robustheit) von realistischen neuronalen Netzen (z.B. faltende neuronale Netze) verifizieren. Die wichtigste Erkenntnis hinter AI\textsuperscript{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\textsuperscript{2} zusammen mit einer umfassenden Evaluierung. Unsere Ergebnisse zeigen, dass AI\textsuperscript{2} deutlich schneller ist als bestehende Analysatoren, die auf symbolischen Verfahren basieren. Diese brauchen oft Stunden, um einfache, vollständig verbundene Netze zu verifizieren. AI\textsuperscript{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\textsuperscript{2} und der Zonotop-Domäne stellen wir DeepZ vor, ein Tool, das zusätzlich Tanh- und Sigmoid-Aktivierungsfunktionen unterstützt und dabei deutlich skalierbarer und präziser ist. Diese Vorteile sind auf sorgfältig entworfene abstrakte Transformatoren zurückzuführen, die sowohl auf neuronalen Netze als auch auf die Zonotop-Domäne zugeschnitten sind. Im Gegensatz zu vielen existierenden Analysatoren behandeln AI\textsuperscript{2} und DeepZ Gleitkomma-Arithmetik korrekt.
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
  *AI²: 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*

**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*

• 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 Stampfl, 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 Mutappillil, Jonas Meier, Adrian Roos, Alberts Reisons, Cédric Neukom, David Jenny, Elena Morbach, Elias Boschung, Erwan Serandour, Fabian Haller, Fabian Lyck, Florian Gatinon, 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.
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

2 Abstract Interpretation of Neural Networks 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 INFERENCES 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

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Adversarial attacks on neural networks.</td>
<td>4</td>
</tr>
<tr>
<td>1.2</td>
<td>Abstract interpretation overview. Sets of concrete neuron activations, starting with a norm ball of radius ( \epsilon ) around input ( x ), are enclosed by concretizations ( (\gamma) ) of abstract elements that symbolically describe overapproximations of those intermediate results after each layer, up to the image of the input set ( B_\epsilon(x) ) under the neural network ( f_\theta ). Robustness properties are verified by checking whether ( a' ) describes a set ( \gamma(a') ) enclosed within a region ( C ) of safe outputs (not shown).</td>
<td>5</td>
</tr>
<tr>
<td>1.3</td>
<td>Standard interface of PSI for users. \textit{Input}: A probabilistic program. \textit{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.)</td>
<td>10</td>
</tr>
<tr>
<td>1.4</td>
<td>One component of the generic abstract transformer for the ReLU activation function, instantiated to the Zonotope domain.</td>
<td>14</td>
</tr>
<tr>
<td>1.5</td>
<td>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.</td>
<td>15</td>
</tr>
<tr>
<td>1.6</td>
<td>Rejection sampling (top) vs. symbolic inference in ( \lambda )PSI (bottom).</td>
<td>16</td>
</tr>
<tr>
<td>2.1</td>
<td>Attacks applied to MNIST images [102].</td>
<td>22</td>
</tr>
<tr>
<td>2.2</td>
<td>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.</td>
<td>24</td>
</tr>
<tr>
<td>2.3</td>
<td>Definition of CAT functions.</td>
<td>26</td>
</tr>
<tr>
<td>2.4</td>
<td>One example computation for each of the three layer types supported by AI(^2).</td>
<td>26</td>
</tr>
<tr>
<td>2.5</td>
<td>The operation of the transformed max pooling layer.</td>
<td>30</td>
</tr>
<tr>
<td>2.6</td>
<td>(a) Abstracting four points with a polyhedron (gray), zonotope (green), and box (blue). (b) The points and abstractions resulting from the affine transformer.</td>
<td>33</td>
</tr>
</tbody>
</table>
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). 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 δ ∈ Δ (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 ≤ λ ≤ min(f′(lx)), f′(ux)). 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 [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 * 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 ≤ x] · [x ≤ 1] · (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 | 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>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Neural network architectures used in our experiments.</td>
<td>58</td>
</tr>
<tr>
<td>3.2</td>
<td>Verified robustness by DeepZ on the ConvBig and ConvSuper networks trained with DiffAI.</td>
<td>60</td>
</tr>
<tr>
<td>4.1</td>
<td>PDF and Correctness Conditions for Several Primitive Distributions.</td>
<td>74</td>
</tr>
<tr>
<td>4.2</td>
<td>Comparison of Exact and Interactive Symbolic Inference Approaches.</td>
<td>90</td>
</tr>
<tr>
<td>4.3</td>
<td>Performance (in s) of Exact and Interactive Symbolic Inference Approaches.</td>
<td>92</td>
</tr>
<tr>
<td>4.4</td>
<td>R2 Benchmark Precision and Analysis Time.</td>
<td>94</td>
</tr>
<tr>
<td>4.5</td>
<td>Infer.NET Benchmark Precision and Analysis Time.</td>
<td>95</td>
</tr>
<tr>
<td>5.1</td>
<td>Probabilistic programs used in evaluation (31 in total). For each program, we indicate if it involves higher-order functions (→), nested inference (⇝), first-class expectations (E), continuous distributions ((\wedge)), continuous observations ((\land)), or symbolic parameters ((\mathbb{R})). SocialCognition and TotalVarDist have multiple variants. Some programs are not expressible in Hakaru (−), while others lead to errors (×), unsimplified (■) or incorrect (××) results. aNot directly expressible; rewritten as first-order programs without function calls, multiple manual steps. bFor concrete instantiation of symbolic parameters.</td>
<td>120</td>
</tr>
</tbody>
</table>
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 \( f_\theta : \mathbb{R}^m \to \mathbb{R}^n \), parameterized by parameters \( \theta \), such that (we can pretend that) there is a gradient \( \nabla f_\theta(x) \) 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 \( \mathcal{L} : \mathbb{R}^n \times [n] \to \mathbb{R} \) (often, this is cross-entropy). Given a finite training set

\[
T = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\} \subseteq \mathbb{R}^m \times [n]
\]
drawn from some underlying, unknown distribution \( \mathcal{D} \), we heuristically optimize the function \( \mathbb{E}_{(x,y) \sim U(T)}[\mathcal{L}(f_\theta(x), y)] \) with respect to \( \theta \) using some variant of gradient descent. We then hope that \( f_\theta \) 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 \( \mathbb{E}_{(x,y) \sim \mathcal{D}}[\mathcal{L}(f_\theta(x), y)] \) 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 \(f_\theta: \mathbb{R}^m \rightarrow \mathbb{R}^n\), 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 \(x \in X\), the corresponding output \(f(x)\) is safe, i.e., \(f(x) \in C\).

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 \((x, y)\), the adversary can select a perturbed input \(x' \in X\), where \(X \ni x\) is some region around \(x\). We then let the safe set \(C\) be some set of outputs of \(f_\theta\) 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 \(A\) contains abstract elements \(a \in A\) that describe sets of program states \(\sigma \in \Sigma\) in a finite, symbolic way. The concretization function \(\gamma: A \rightarrow \mathcal{P}(\Sigma)\) maps a symbolic abstract element to the concrete set of program states it describes. For some function \(f: \Sigma \rightarrow \Sigma\), the concrete transformer \(T_f: \mathcal{P}(\Sigma) \rightarrow \mathcal{P}(\Sigma)\) 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 \(T^\#_f: A \rightarrow A\) has to satisfy the soundness condition \(T_f(\gamma(a)) \subseteq \gamma(T^\#_f(a))\) for all \(a \in A\).

(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 \(\alpha: \mathcal{P}(\Sigma) \rightarrow A\). The functions \(\gamma\) and \(\alpha\) 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].

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

Figure 1.1: Adversarial attacks on neural networks.
1.1 Abstract Interpretation of Neural Networks

Concrete Semantics $B_\epsilon(x) \xrightarrow{\text{Conv}} FC \xrightarrow{f_\theta[B_\epsilon(x)]}$

Abstract Interpretation $a \in A \xrightarrow{T^\#_{\text{Conv}}} T^\#_{\text{FC}} \xrightarrow{\gamma} a' \in A$

Figure 1.2: Abstract interpretation overview. Sets of concrete neuron activations, starting with a norm ball of radius $\epsilon$ around input $x$, are enclosed by concretizations ($\gamma$) of abstract elements that symbolically describe overapproximations of those intermediate results after each layer, up to the image of the input set $B_\epsilon(x)$ under the neural network $f_\theta$. 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 $f_\theta$, the region $X$ and abstract domains $A_m$ for concrete domains $\mathbb{R}^n$, we can construct an abstract transformer $T^\#_{f_\theta} : A_m \rightarrow A_n$ for $f_\theta$. 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 $f_\theta$. 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 $X = B_\epsilon(x)$ proceeds compositionally through each neural network layer.

Contribution In this thesis, we demonstrate how to do all of those steps. We have developed AI², 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 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 AI² 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, AI² 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 AI² 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 AI² 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 AI² and provides implementations of DeepZ, DeepPoly, kPoly and DeepG, among others.

• **Probabilistic Abstract Interpretation**: Generalizing the framework of AI², 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 AI² 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-
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:

$$\Pr[A \mid B] = \frac{\Pr[A \cap B]}{\Pr[B]}$$

The weight of an assignment $\alpha$ 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:

```python
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 $\text{sampleX}$ represents the density $\Pr[X = x]$ and $\text{sampleY}$ 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

$$\Pr[X = x \mid Y = y] = \frac{\Pr[Y = y, X = x]}{\Pr[Y = y]},$$

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

$$\Pr[X = x \mid Y = y] = \frac{\Pr[Y = y \mid X = x] \cdot \Pr[X = x]}{\Pr[Y = y]}.$$

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 explicitly 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 probabilistic 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
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 data structures 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.\footnote{Although we had to fix some bugs in existing libraries, as they had never been validated at the scale required by our system.} We present AI$^2$ in Chapter 2.
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**  
AI² 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 AI²’s generic transformers. Note how it has to split the abstract element on the condition $x_1 \geq 0$. 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 AI² 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 AI² 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 AI² 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 behavior can both be random, but the methods we use to analyze them are fully deterministic.
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
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
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 $\epsilon$ (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...
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.

Raghunathan 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. 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 $\epsilon$ 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 AI² to evaluate provable robustness of neural network defenses (§2.6).

AI² 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 interpretation 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 \( x \in \mathbb{R}^n \), \( x_i \) denotes its \( i \)th entry, and for a matrix \( W \in \mathbb{R}^{n \times m} \), \( W_i \) denotes its \( i \)th row and \( W_{ij} \) denotes the entry in its \( i \)th 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 \( f: \mathbb{R}^m \rightarrow \mathbb{R}^n \) for \( m, n \in \mathbb{N} \) and is defined recursively. Any affine transformation \( f(\bar{x}) = W \cdot \bar{x} + \bar{b} \) is a CAT function, for a matrix \( W \) and a vector \( \bar{b} \). Given sequences of conditions \( E_1, \ldots, E_k \) and CAT functions \( f_1, \ldots, f_k \), we write

\[
    f(\bar{x}) = \text{case } E_1: f_1(\bar{x}), \ldots, \text{case } E_k: f_k(\bar{x}).
\]

This is also a CAT function, which returns \( f_i(\bar{x}) \) for the first \( E_i \) satisfied by \( \bar{x} \). The conditions are conjunctions of constraints of the form \( x_i \geq x_j \), \( x_i \geq 0 \) 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 \( f(\bar{x}) = f''(f'(\bar{x})) \).

**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 \( \bar{x} \in \mathbb{R}^{m \times n \times r} \) can be reshaped to \( \bar{x}^v \in \mathbb{R}^{m \cdot n \cdot r} \) in a canonical way, first by depth, then by column, finally by row. That is, given \( \bar{x} \), we have

\[
    \bar{x}^v = (x_{1,1,1} \ldots x_{1,1,r} x_{1,2,1} \ldots x_{1,2,r} \ldots x_{m,n,1} \ldots x_{m,n,r})^T.
\]
Abstract interpretation for artificial intelligence

\[ f(\bar{x}) ::= W \cdot \bar{x} + b \]

| case \( E_1 : f_1(\bar{x}) \), ..., case \( E_k : f_k(\bar{x}) \) |
| f(f'(\bar{x})) |

\[ E ::= E \land E \mid x_i \geq x_j \mid x_i \geq 0 \mid x_i < 0 \]

Figure 2.3: Definition of CAT functions.

**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 \( \bar{x} \in \mathbb{R}^n \) as \( \text{ReLU}(\bar{x}) = (\text{ReLU}(x_1), \ldots, \text{ReLU}(x_n)) \).

**ReLU to CAT** We can express the ReLU activation function as \( \text{ReLU} = \text{ReLU}_n \circ \ldots \circ \text{ReLU}_1 \) where \( \text{ReLU}_i \) processes the \( i \)th entry of the input \( \bar{x} \) and is given by

\[
\text{ReLU}_i(\bar{x}) = \begin{cases} 
(\bar{x}_i \geq 0): \bar{x}_i, \\
(\bar{x}_i < 0): I_{i=0} \cdot \bar{x}.
\end{cases}
\]

\( I_{i=0} \) is the identity matrix with the \( i \)th 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 \( \text{FC}_{W, b} : \mathbb{R}^m \to \mathbb{R}^n \) parameterized by a weight matrix \( W \in \mathbb{R}^{n \times m} \) and a bias \( \bar{b} \in \mathbb{R}^n \). For \( \bar{x} \in \mathbb{R}^m \), we have

\[ \text{FC}_{W, b}(\bar{x}) = \text{ReLU}(W \cdot \bar{x} + \bar{b}) \]

Fig. 2.4a shows an FC layer computation for \( \bar{x} = (2, 3, 1) \).
CONVOLUTIONAL LAYER  A convolutional layer is defined by a series of \( t \) filters \( \mathcal{F}^{p,q} = (\mathcal{F}^{p,q}_1, \ldots, \mathcal{F}^{p,q}_t) \), parameterized by the same \( p \) and \( q \), where \( p \leq m \) and \( q \leq n \). A filter \( \mathcal{F}^{p,q}_i \) 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:

\[
\mathcal{F}^{p,q}_i : \mathbb{R}^{m \times n \times r} \rightarrow \mathbb{R}^{(m-p+1) \times (n-q+1)}.
\]

The entries of the output \( y \) for a given input \( x \) are given by

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

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 \( \text{Conv}_F \), corresponding to a convolutional layer with \( t \) filters, has the following type:

\[
\text{Conv}_F : \mathbb{R}^{m \times n \times r} \rightarrow \mathbb{R}^{(m-p+1) \times (n-q+1) \times t}.
\]

As expected, the function \( \text{Conv}_F \) 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 \( t \) filters, has the following type:

\[
\text{Conv}_F : \mathbb{R}^{m \times n \times r} \rightarrow \mathbb{R}^{(m-p+1) \times (n-q+1) \times t}.
\]

As expected, the function \( \text{Conv}_F \) 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 \( t \) filters. 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 \( \text{Conv}_F \), we define a matrix \( W^F \) whose entries are those of the weight matrices for each filter (replicated to simulate sliding), and a bias \( b^F \) consisting of copies of the filters’ biases. We then treat the convolutional layer \( \text{Conv}^F \) like the equivalent \( \text{FC}_{W^F, b^F} \).

We will provide formal definitions of \( W^F \) and \( b^F \) 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 \( x_1^v, x_2^v, x_3^v \) 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}^T. Thus, to obtain y^T_{1} = y_{1,1}, we define the first row in W^F such that its 1\textsuperscript{st}, 2\textsuperscript{nd}, 5\textsuperscript{th}, and 6\textsuperscript{th} 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^T_{2} = y_{1,2}, we define the second row in W^F such that its 2\textsuperscript{nd}, 3\textsuperscript{rd}, 6\textsuperscript{th}, and 7\textsuperscript{th} entries are W_{1,1}, W_{1,2}, W_{2,1} and W_{2,2} (also b = b). By following this transformation, we obtain the matrix W^F \in \mathbb{R}^9 \times \mathbb{R}^{16} and the bias b^F \in \mathbb{R}^9:

\[
W^F = \begin{pmatrix}
1 & 0 & 0 & -1 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & -1 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & -1 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & -1 & 2 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 \\
\end{pmatrix},
\]

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 \leq k \leq t, we have

\[
\text{Conv}_F(\tilde{x}): \mathbb{R}^{n \times m \times r} \rightarrow \mathbb{R}^{(m-p+1) \times (n-q+1) \times t}
\]

\[
\text{Conv}_F(\tilde{x})_{i,j,k} = \text{ReLU}(\sum_{i'=1}^{p} \sum_{j'=1}^{q} \sum_{k'=1}^{r} W^k_{i',j',k'} \cdot x_{(i+i'-1),(j+j'-1),k'} + b^k),
\]

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

\[
\text{Conv}'_F(\tilde{x}): \mathbb{R}^{n \cdot m \cdot r} \rightarrow \mathbb{R}^{(m-p+1) \cdot (n-q+1) \cdot t}
\]

\[
\text{Conv}'_F(\tilde{x})_{(i-1)+t \cdot (j-1)+k} = \text{ReLU}(\sum_{i'=1}^{p} \sum_{j'=1}^{q} \sum_{k'=1}^{r} W^k_{i',j',k'} \cdot x_{n \cdot r \cdot (i+i'-2) + r \cdot (j+j'-2) + k'} + b^k),
\]

for 1 \leq i \leq m - p + 1, 1 \leq j \leq n - q + 1 and 1 \leq k \leq 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 b^F \in \mathbb{R}^{((m-p+1) \cdot (n-q+1) \cdot t)} such that

\[
\text{Conv}_F(\tilde{x})^V = \text{Conv}'_F(\tilde{x}^V) = \text{ReLU}(W^F \cdot \tilde{x}^V + b^F) = FC_{W^F,b^F}(\tilde{x}).
\]
The entries of $W^F$ and $b^F$ are obtained by equating
\[
\begin{align*}
\text{FC}(\tilde{e}_l)_{(n-q+1) \cdot t \cdot (i-1) + t \cdot (j-1) + k} &= \text{ReLU}(W^F_{(n-q+1) \cdot t \cdot (i-1) + t \cdot (j-1) + k, l} + b^F_{(n-q+1) \cdot t \cdot (i-1) + t \cdot (j-1) + k})
\end{align*}
\]
with
\[
\begin{align*}
\text{Conv}^F_{F}(\tilde{e}_l)_{(n-q+1) \cdot t \cdot (i-1) + t \cdot (j-1) + k}
&= \text{ReLU}(\sum_{i'=1}^{p} \sum_{j'=1}^{q} \sum_{k'=1}^{r} W^k_{i', j', k'} \cdot [l = n \cdot r \cdot (i + i' - 2) + r \cdot (j + j' - 2) + k'] + b^k),
\end{align*}
\]
for standard basis vectors $\tilde{e}_l$ with $(\tilde{e}_l)_i = [l = i]$ for $1 \leq l \leq n$ and $1 \leq i \leq n \cdot m \cdot r$. This way, we obtain
\[
\begin{align*}
W^F_{(n-q+1) \cdot t \cdot (i-1) + t \cdot (j-1) + k, l}
&= \sum_{i'=1}^{p} \sum_{j'=1}^{q} \sum_{k'=1}^{r} W^k_{i', j', k'} \cdot [l = n \cdot r \cdot (i + i' - 2) + r \cdot (j + j' - 2) + k']
\end{align*}
\]
and
\[
\begin{align*}
&b^F_{(n-q+1) \cdot t \cdot (i-1) + t \cdot (j-1) + k} = b^k,
\end{align*}
\]
for $1 \leq i \leq m - p + 1$, $1 \leq j \leq n - q + 1$ and $1 \leq k \leq t$. Note that here, $[\varphi]$ is an Iverson bracket, which is equal to 1 if $\varphi$ holds and equal to 0 otherwise.

**Max Pooling (MP) Layer** An MP layer takes a three-dimensional array $\chi \in \mathbb{R}^{m \times n \times r}$ and reduces the height $m$ of $\chi$ by a factor of $p$ and the width $n$ of $\chi$ 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 $\chi$ of size $p \times q$ and return the maximal value in their subrectangle. Formally, the MP layer is a function $\text{MaxPool}_{p,q} : \mathbb{R}^{m \times n \times r} \rightarrow \mathbb{R}^{\frac{m}{p} \times \frac{n}{q} \times r}$ that for an input $\chi$ returns the three-dimensional array $\bar{y}$ given by:
\[
\begin{align*}
y_{i,j,k} &= \max([x_{i',j',k'} \mid p \cdot (i - 1) < i' \leq p \cdot i, q \cdot (j - 1) < j' \leq q \cdot j]).
\end{align*}
\]
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 $\text{MaxPool}^F_{p,q} : \mathbb{R}^{m \cdot n \cdot r} \rightarrow \mathbb{R}^{\frac{m}{p} \cdot \frac{n}{q} \cdot r}$ be the function that is obtained from $\text{MaxPool}_{p,q}$ by reshaping its input and output: $\text{MaxPool}^F_{p,q}(\chi^v) = \text{MaxPool}_{p,q}(\chi)^v$. To represent max pooling as a CAT function, we define a series of CAT functions whose composition is $\text{MaxPool}^F_{p,q}$:
\[
\text{MaxPool}_{p,q}^F = f_{\frac{m}{p} \cdot \frac{n}{q} \cdot r} \circ \ldots \circ f_1 \circ f_{\text{MP}}.
\]
The first function is \( f_{\text{MP}}(x^v) = W_{\text{MP}} \cdot x^v \), which reorders its input vector \( x^v \) to a vector \( \bar{x}^\text{MP} \) in which the values of each max pooling subrectangle of \( x \) are adjacent. The remaining functions execute standard max pooling. Concretely, the function \( f_i \in \{ f_1, \ldots, f_{\frac{m \cdot n}{p \cdot q}} \} \) executes max pooling on the \( i \)th subrectangle by selecting the maximal value and removing the other values. We provide formal definitions of the CAT functions \( f_{\text{MP}} \) and \( f_i \) further below.

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

The computation begins from the input vector \( x^v \), which is the reshaping of \( x \) from Fig. 2.4c. The values of the first \( 2 \times 2 \) subrectangle in \( x \) (namely, 0, 1, 2 and -4) are separated in \( x^v \) by values from another subrectangle (3 and -2). To make them contiguous, we reorder \( x^v \) using a permutation matrix \( W_{\text{MP}} \), yielding \( \bar{x}^\text{MP} \). In our example, \( W_{\text{MP}} \) is given by

\[
W_{\text{MP}} = \begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
\end{pmatrix}
\]

One entry in each row of \( W_{\text{MP}} \) is 1, all other entries are zeros. If row \( i \) has entry \( j \) set to 1, then the \( j \)th value of \( x^v \) is moved to the \( i \)th entry of \( \bar{x}^\text{MP} \). For example, we placed a one in the fifth column of the third row of \( W_{\text{MP}} \) to move the value \( x^v_5 \) to entry 3 of the output vector.

Next, for each \( i \in \{1, \ldots, \frac{m \cdot n}{p \cdot q} \} \), 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 \( i \)th subrectangle of \( x \) in Fig. 2.4c. It then replaces those \( p \cdot q \) values by their maximum:

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

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

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

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:

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

As, in general, $k$ is not known in advance, we need to write $f_1$ 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:

$$f_1(\overline{x}) = \begin{cases} \text{case } (x_1 \geq x_2) \land (x_1 \geq x_3) \land (x_1 \geq x_4) : & W^{(1,1)} \cdot \overline{x}, \\ \text{case } (x_2 \geq x_1) \land (x_2 \geq x_3) \land (x_2 \geq x_4) : & W^{(1,2)} \cdot \overline{x}, \\ \text{case } (x_3 \geq x_1) \land (x_3 \geq x_2) \land (x_3 \geq x_4) : & W^{(1,3)} \cdot \overline{x}, \\ \text{case } (x_4 \geq x_1) \land (x_4 \geq x_2) \land (x_4 \geq x_3) : & W^{(1,4)} \cdot \overline{x}. \end{cases}$$

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:

$$\text{MaxPool}^{1}_{2,2} = f_4 \circ f_3 \circ f_2 \circ f_1 \circ f^{MP}.$$ 

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 $\overline{x}^v$ in Fig. 2.5:

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

**Max Pooling Layer Details** MaxPool$^{1}_{p,q} : \mathbb{R}^{m \times n \times r} \rightarrow \mathbb{R}^{\frac{m}{p} \times \frac{n}{q} \times r}$ partitions the input vector into disjoint blocks of size $p \times q \times 1$ and replaces each block by its maximum value. Furthermore, MaxPool$^{1}_{p,q} : \mathbb{R}^{m \cdot n \cdot r} \rightarrow \mathbb{R}^{\frac{m}{p} \cdot \frac{n}{q} \cdot r}$ is obtained from MaxPool$^{1}_{p,q}$ by reshaping both the input and output: MaxPool$^{1}_{p,q}(\overline{x}^v) = \text{MaxPool}^{1}_{p,q}(\overline{x})^v$. We will represent MaxPool$^{1}_{p,q}$ as a composition of CAT functions,

$$\text{MaxPool}^{1}_{p,q} = f_{\frac{m}{p}} \circ f_{\frac{n}{q}} \circ \ldots \circ f_1 \circ f^{MP}.$$
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 \( f : \mathbb{R}^m \to \mathbb{R}^n \), a set of inputs \( X \subseteq \mathbb{R}^m \), and
2.2 BACKGROUND: ABSTRACT INTERPRETATION

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.

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

Fig. 2.6 shows a CAT function $f: \mathbb{R}^2 \rightarrow \mathbb{R}^2$ that is defined as $f(x) = \begin{pmatrix} 2 & -1 \\ 0 & 1 \end{pmatrix} \cdot x$ and four input points for the function $f$, given as $X = \{(0,1), (1,1), (1,3), (2,2)\}$. Let the property be $C = \{(y_1, y_2) \in \mathbb{R}^2 \mid y_1 \geq -2\}$, 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:

$T_f: \mathcal{P}(\mathbb{R}^m) \rightarrow \mathcal{P}(\mathbb{R}^n), \quad T_f(X) = \{f(\overline{x}) \mid \overline{x} \in X\}.$

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

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

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:

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

In other words, $z$ has the form $z(\bar{e}) = M \cdot \bar{e} + \bar{b}$ where $\bar{e}$ is a vector of error terms satisfying interval constraints $\bar{e}_i \in [a_i, b_i]$ for $1 \leq i \leq m$. The bias vector $\bar{b}$ captures the center of the zonotope, while the matrix $M$ captures the boundaries of the zonotope around $\bar{b}$. 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

$$z(e_1, e_2, e_3) = (1 + 0.5 \cdot e_1 + 0.5 \cdot e_2, 2 + 0.5 \cdot e_1 + 0.5 \cdot e_3).$$

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 $A \cdot \bar{x} \leq \bar{b}$, for some matrix $A$ and a vector $\bar{b}$. A polyhedron $C$ contains all points which satisfy the constraints in $C$. In our example, $X$ can be abstracted by the following polyhedron:

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

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 $T_f: \mathcal{P}(\mathbb{R}^m) \rightarrow \mathcal{P}(\mathbb{R}^n)$ of a function $f: \mathbb{R}^m \rightarrow \mathbb{R}^n$ 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 $T_f^*: \mathcal{A}^m \rightarrow \mathcal{A}^n$. 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 $A^m$ 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 $\alpha$ and the concretization function $\gamma$. Such an abstraction function $\alpha^m: \mathcal{P}(\mathbb{R}^m) \rightarrow A^m$ maps a set of vectors to an abstract element in $A^m$ that overapproximates it. For example, in the Box domain:

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

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

$$\gamma^2([0 \leq x_1 \leq 2, 1 \leq x_2 \leq 3]) = \{(0,1), (0,2), (0,3),
(1,1), (1,2), (1,3),
(2,1), (2,2), (2,3), \ldots\}.$$ 

This only shows the 9 vectors with integer components. We can now define soundness. An abstract transformer $\mathcal{T}_f^\#$ is sound if for all $a \in A^m$, we have $\mathcal{T}_f(\gamma^m(a)) \subseteq \gamma^n(\mathcal{T}_f^\#(a))$, where $\mathcal{T}_f$ 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 $a = \mathcal{T}_f^\#(\alpha^m(X))$ proves a property $\mathcal{T}_f(X) \subseteq C$ if $\gamma^m(a) \subseteq C$. 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 $C = \{(y_1, y_2) \in \mathbb{R}^2 \mid y_1 \geq -2\}$ 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
2.3 \( \text{AI}^2\): \( \text{AI} \) for Neural Networks

In this section we present \( \text{AI}^2 \), 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

\[
f(\overline{x}) = \text{ReLU}_2 \left( \text{ReLU}_1 \left( \left( \begin{array}{c} 2 \\ -1 \end{array} \right) \cdot \overline{x} \right) \right).
\]

Recall that

\[
\text{ReLU}_i(\overline{x}) = \begin{cases} x_i & (x_i \geq 0) \\ 0 & (x_i < 0) \end{cases}
\]

where \( I_{1 \times 0} \) is the identity matrix with the \( i \)th 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 \( \alpha \)). 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 \( z_0: [-1, 1]^3 \rightarrow \mathbb{R}^2 \), 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 \( N = f'_1 \circ \cdots \circ f'_i \) is a composition of multiple CAT functions \( f'_i \) of the shape either \( f'_i(\bar{x}) = W \cdot \bar{x} + \bar{b} \) or \( f'_i(\bar{x}) = \text{case } E_1 : f_1(\bar{x}), \ldots, \text{case } E_k : f_k(\bar{x}) \), we only have to define abstract transformers for these two kinds of functions. We then obtain the abstract transformer \( T^\#_N = T^\#_{f'_i} \circ \cdots \circ T^\#_{f'_1} \).

**Abstracting affine functions** To abstract functions of the form

\[
f(\bar{x}) = W \cdot \bar{x} + \bar{b},
\]

we assume that the underlying abstract domain supports an operator \( \text{Aff} \) that overapproximates such functions. We note that for Zonotope and Polyhedra, this operation is supported and exact. Fig. 2.7 demonstrates \( \text{Aff} \) 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

\[
z_1(\epsilon_1, \epsilon_2, \epsilon_3) = (2 \cdot (1 + 0.5 \cdot \epsilon_1 + 0.5 \cdot \epsilon_2) - (2 + 0.5 \cdot \epsilon_1 + 0.5 \cdot \epsilon_3), 2 + 0.5 \cdot \epsilon_1 + 0.5 \cdot \epsilon_3) = (0.5 \cdot \epsilon_1 + \epsilon_2 - 0.5 \cdot \epsilon_3, 2 + 0.5 \cdot \epsilon_1 + 0.5 \cdot \epsilon_3).
\]

**Abstracting case functions** To abstract functions of the form

\[
f(\bar{x}) = \text{case } E_1 : f_1(\bar{x}), \ldots, \text{case } E_k : f_k(\bar{x}),
\]

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) \( \text{meet} \) with a conjunction of linear constraints and (2) \( \text{join} \). The \( \text{meet} \) (\( \sqcap \)) operator is an abstract transformer for set intersection: for an inequality expression \( E \) from Fig. 2.3, we have

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

The \( \text{join} \) (\( \sqcup \)) 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

\[
\gamma^n(\bot) = \emptyset,
\]

\( \bot \sqcap E = \bot \) and \( a \sqcup \bot = a \) for all \( a \in A \).
For \( f(\vec{x}) = W \cdot \vec{x} + \vec{b} \), \( T^f_i(a) = \text{Aff}(a, W, \vec{b}) \).

For \( f(\vec{x}) = \text{case} \ E_1: f_1(\vec{x}), \ldots, \text{case} \ E_k: f_k(\vec{y}) \),
\[
T^f_i(a) = \bigcup_{1 \leq i \leq k} T^f_i(a \cap E_i).
\]

For \( f(\vec{x}) = f_2(f_1(\vec{x})) \), \( T^f_i(a) = T^f_{i2}(T^f_{i1}(a)) \).

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 \( z_2 = z_1 \cap (x_1 \geq 0) \) and \( z_3 = z_1 \cap (x_1 < 0) \). 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 \( z_2 : [-1, 1] \times [0, 1] \times [-1, 1] \rightarrow \mathbb{R}^2 \) such that \( z_2(\vec{e}) = z_1(\bar{\vec{e}}) \) and \( z_3 : [-1, 1] \times [-1, 0] \times [-1, 1] \rightarrow \mathbb{R}^2 \) such that \( z_3(\vec{e}) = z_1(\bar{\vec{e}}) \) for \( \bar{\vec{e}} \) 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 \( \cap \) 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 \( f_1(\vec{x}) = \bar{\vec{x}} \) to \( z_2 \) and we apply the abstract transformer of \( f_2(\vec{x}) = I_{0+0} \cdot \vec{x} \) to \( z_3 \). The resulting zonotopes are \( z_4 = z_2 \) and \( z_5 : [-1, 1]^2 \rightarrow \mathbb{R}^2 \) such that \( z_5(\vec{e}_1, \vec{e}_3) = (0.2 + 0.5 \cdot \vec{e}_1 + 0.5 \cdot \vec{e}_3) \). 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 \( z_6 \cap (x_1 < 0) = \bot \), this results in \( z_7 = z_6 \). Finally, \( y^2(z_7) \) 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 \( \bot \) 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 \( T_f : \mathcal{P}(\mathbb{R}^m) \rightarrow \mathcal{P}(\mathbb{R}^n) \) and any abstract input \( a \in \mathcal{A}^m \),
\[
T_f(y^m(a)) \subseteq y^n(T^f_i(a)).
\]

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 \( N : \mathbb{R}^m \rightarrow \mathbb{R}^n \) 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 \( N(\bar{x}) \in C \) for all \( \bar{x} \in X \).

**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 \( L \):

\[
C_L = \left\{ \bar{y} \in \mathbb{R}^n \left| \arg \max_{i \in \{1, \ldots, n\}} (y_i) = L \right. \right\}.
\]

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

\[
C_2 = \{\bar{y} \mid \arg \max_{i, j} (y_i, y_j) = 2\}.
\]

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 \( N : \mathbb{R}^m \rightarrow \mathbb{R}^n \), a robustness property \((X, C)\) and an abstract domain \( \mathcal{A} \) (supporting \( \sqcup, \sqcap \) with a conjunction of linear constraints, \text{Aff}, and \( \perp \)) with an abstraction function \( \alpha \) and a concretization function \( \gamma \). 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 \( \alpha \) and \( \gamma \). 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.
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 \(a \sqcap \left(\bigwedge_j \neg l_{i,j}\right) = \perp\) 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 \(y_2 \geq y_1\). 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 \(z_7 \sqcap (y_2 < y_1) = \perp\), the property is successfully verified. However, note that we would not be able to prove some other true properties, such as \(y_1 \geq 0\). 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., \(\gamma(\mathcal{A}) = \bigcup_{a \in \mathcal{A}} \gamma(a)\) for \(\mathcal{A} \in \mathcal{P}(\mathcal{A})\)).

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, \( \text{AI}^2 \) heuristically selects two abstract elements that minimize the distance between the centers of their bounding boxes. In our experiments, we denote by \( \text{ZonotopeN} \) or \( ZN \) the bounded powerset domain with bound \( N \geq 2 \) and underlying abstract domain \( \text{Zonotope} \).

### 2.5 Evaluation of \( \text{AI}^2 \)

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

- \( \text{AI}^2 \) can prove useful robustness properties for convolutional networks with 53,000 neurons and large fully connected feedforward networks with 1,800 neurons.

- \( \text{AI}^2 \) benefits from more precise abstract domains: \( \text{Zonotope} \) enables \( \text{AI}^2 \) to prove substantially more properties over Box. Further, \( \text{ZonotopeN} \), with \( N \geq 2 \), can prove stronger robustness properties than \( \text{Zonotope} \) alone.

- \( \text{AI}^2 \) scales better than the SMT-based Reluplex \([88]\): \( \text{AI}^2 \) is able to verify robustness properties on large networks with \( \geq 1,200 \) 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 \( 28 \times 28 \) pixels. The images show white digits on a black background.

CIFAR consists of 60,000 colored photographs with 3 color channels, whose resolution is \( 32 \times 32 \) 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.
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 \(S_{\bar{x},\delta}\) that are parameterized by an input \(\bar{x} \in \mathbb{R}^m\) and a robustness bound \(\delta \in [0, 1]\). The robustness region is defined as:

\[
S_{\bar{x},\delta} = \{\bar{x}' \in \mathbb{R}^m \mid \forall i \in [1, m]. \, 1 - \delta \leq x_i \leq x_i' \leq 1 \lor x_i' = x_i\}.
\]

For example, the robustness region for \(\bar{x} = (0.6, 0.85, 0.9)\) 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 $\delta$ increases the region’s size.

In our experiments, we used $\text{AI}^2$ to check whether all inputs in a given region $S_{x,\delta}$ are classified to the label assigned to $x$. We consider 6 different robustness bounds $\delta$, 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 $S_{x,\delta}$, each pixel of the image $\bar{x}$ 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 $\text{AI}^2$ is not limited to certifying robustness against such brightening perturbations. In general, $\text{AI}^2$ 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 $\ell_\infty$ 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 $\Delta$, resulting in 60 properties per dataset. We ran $\text{AI}^2$ 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 $\text{AI}^2$ 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 $\text{AI}^2$’s ability to verify robustness. We also plot $\text{AI}^2$’s running times for different abstract domains to investigate the trade-off between precision and scalability. Finally, we compare $\text{AI}^2$ to Reluplex.

**Proving Robustness of Convolutional Networks** We start with our results for convolutional networks. $\text{AI}^2$ 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 \(9 \times 200\) 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
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 AI² to prove stronger robustness properties. In this experiment, we consider our $9 \times 200$ 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 \( \text{AI}^2 \). We plot a separate line for each robustness bound. All runs of \( \text{AI}^2 \) 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 \( \text{AI}^2 \)'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 \( \delta \geq 0.025 \)). 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 \( \text{AI}^2 \) as a function of the abstract domain. Each point in the graph represents the average running time of \( \text{AI}^2 \) 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 \( \text{AI}^2 \) can efficiently verify robustness of large networks. \( \text{AI}^2 \) terminates within a few minutes for all MNIST FNNs and all considered domains. Further, we observe that \( \text{AI}^2 \) 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 \( \text{AI}^2 \)'s running time. More importantly, \( \text{AI}^2 \)'s running time is polynomial in the bound \( N \) of ZonotopeN, which allows one to adjust \( \text{AI}^2 \)'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 \( \text{AI}^2 \), we ran both systems on all MNIST FNN benchmarks. We ran \( \text{AI}^2 \) using Zonotope and Zonotope64. For both Reluplex and \( \text{AI}^2 \), 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 \( \text{AI}^2 \) 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 \(\mathbf{x}\) and computing a vector \(\mathbf{p}_{N,\mathbf{x}}\) in the input space along which an adversarial example is likely to be found. An adversarial input \(\mathbf{a}\) is then generated by taking a step \(\epsilon\) along this vector: \(\mathbf{a} = \mathbf{x} + \epsilon \cdot \mathbf{p}_{N,\mathbf{x}}\).
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 \( x \) to \( x + \delta \cdot \mathbf{p}_{N,x} \) for some robustness bound \( \delta \):

\[
L_{N,x,\delta} = \{ x + \epsilon \cdot \mathbf{p}_{N,x} \mid \epsilon \in [0, \delta] \}.
\]

This robustness region is a zonotope and can thus be precisely captured by \( \text{AI}^2 \).

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 \( x \) and computed \( \mathbf{p}_{N,x} \). Then, for each network, our goal was to find the largest bound \( \delta \) for which \( \text{AI}^2 \) proves the region \( L_{N,x,\delta} \) robust. To approximate the largest robustness bound, we ran binary search to depth 6 and ran \( \text{AI}^2 \) with the Zonotope domain for each candidate bound \( \delta \). We refer to the largest robustness bound verified by \( \text{AI}^2 \) 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 $\delta$ for the middle 50% of the images, whereas the whiskers represent the minimum and maximum $\delta$. 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.
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.
In Chapter 2, we have shown how to use the classic framework of abstract interpretation [35] to soundly approximate the behavior of neural networks. AI² 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 AI² was limited to brightening attacks [130] and did not consider $L_\infty$-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 $L_\infty$-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 $L_\infty$ 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 $L_\infty$-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 $x$ and relevant perturbations to $x$ (resulting in many different inputs) into one abstract element $\alpha_R$ and then analyze the network starting from $\alpha_R$, producing an abstract output $\alpha^0_R$. We can then verify the robustness property of interest over $\alpha^0_R$: 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 $(R_{x,e}, C_L)$ where $R_{x,e}$ represents the set of perturbed inputs around the original input $x \in \mathbb{R}^m$ based on a small constant $e > 0$. $C_L$ is a robustness condition which defines the set of outputs that all have the same label $L$:

$$C_L = \left\{ y \in \mathbb{R}^n \mid \arg\max_{i \in \{1, \ldots, n\}} (y_i) = L \right\}.$$
A robustness property \((R_{\pi,\epsilon}, C_L)\) holds iff the set of outputs \(O_R\) corresponding to all inputs in \(R_{\pi,\epsilon}\) is included in \(C_L\). \((R_{\pi,\epsilon}, C_L)\) can be verified using abstract interpretation by checking if the abstract output \(\alpha_R^0\) resulting from analyzing the network with an abstraction of \(R_{\pi,\epsilon}\) is included in \(C_L\).

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

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

This expression consists of a center coefficient \(\alpha_0\), a set of noise symbols \(\epsilon_i\), and coefficients \(\alpha_i\) representing partial deviations around the center. Crucially, the noise symbols \(\epsilon_i\) 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 \(Z \subseteq \mathbb{R}^n\) is represented by a collection of affine forms for all variables \(x \in X\), and is the set of all possible (joint) values of the affine forms for an arbitrary instantiation of the noise symbols \(\epsilon_i\). 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 of 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
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 $Z$ can be represented with the assignment $y := \max(0, x)$ where $x, y \in X$. If $x$ can only have positive ($l_x > 0$) or non-positive values ($u_x \leq 0$) in $Z$, 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 $Z$ 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 $\lambda$, which is a parameter. To ensure soundness for all approximations in Fig. 3.1 (a), we require $0 \leq \lambda \leq \frac{u_x}{u_x - l_x}$. Similarly, $\frac{u_x}{u_x - l_x} \leq \lambda \leq 1$ 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:

$$A_1(\lambda) = (1 - \lambda) \cdot u_x \cdot (u_x - l_x).$$  \hfill (3.3)
\( A_1(\lambda) \) is a decreasing function of \( \lambda \). 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:

\[
A_2(\lambda) = \lambda \cdot (-l_x) \cdot (u_x - l_x).
\] (3.4)

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

**Theorem 2.** Let \( Z \) be the input to a ReLU function \( y = \text{ReLU}(x) \). Consider the set of pointwise Zonotope approximations \( O \) 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:

\[
\hat{y} = \begin{cases} 
\hat{x}, & \text{if } l_x > 0, \\
[0, 0], & \text{if } u_x \leq 0, \\
[\lambda_l, \lambda_u] \cdot \hat{x} + [\mu_l, \mu_u] + [\mu_l, \mu_u] \cdot \epsilon_{\text{new}}, & \text{otherwise}.
\end{cases}
\] (3.5)

Here \( \lambda_l, \lambda_u \) are floating point representations of \( \lambda_{\text{opt}} = \frac{u_x}{u_x - l_x} \) using rounding towards \(-\infty \) and \(+\infty \) respectively. Similarly, \( \mu_l, \mu_u \) are floating point representations of \( \mu = -\frac{u_x - l_x}{2(u_x - l_x)} \) 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 \( Z \) can be represented with the assignment \( y := \sigma(x) \) where \( x, y \in X \) 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 \( \lambda \) which parameterizes the set. To ensure soundness, we have \( 0 \leq \lambda \leq \min(\sigma'(l_x), \sigma'(u_x)) \) where \( \sigma' = \frac{e^x}{(1 + e^x)^2} \).

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

\[
A(\lambda) = (\sigma(u_x) - \sigma(l_x) - \lambda \cdot (u_x - l_x)) \cdot (u_x - l_x)
\] (3.6)
A smooth 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:

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

Here, $f(u_x)_l, f(u_x)_u$ are floating point representations of $f(u_x)$ rounded towards $-\infty$ and $+\infty$ respectively and $\lambda_l, \lambda_u$ are floating point representations of $\lambda_{\text{opt}} = \min(f'(l_x), f'(u_x))$ using rounding towards $-\infty$ and $+\infty$ respectively. Similarly $\mu_l^1, \mu_u^1$ and $\mu_l^2, \mu_u^2$ are floating point representations of $\mu_1 = \frac{1}{2}(f(u_x) + f(l_x) - \lambda_{\text{opt}} \cdot (u_x + l_x))$ and $\mu_2 = \frac{1}{2}f(u_x) - f(l_x) - \lambda_{\text{opt}} \cdot (u_x - l_x)$ computed using rounding towards $-\infty$ and $+\infty$ and adding the error due to the non-associativity of floating point addition, and $\epsilon_{\text{new}} \in [-1, 1]$ 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 $f: \mathbb{R} \rightarrow \mathbb{R}$ is said to be S-shaped if $f'(x) \geq 0$ and there exists a value $x'$ such that for all $x \in \mathbb{R}$, we have $f''(x) \leq 0 \iff x \leq x'$. 
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 $28 \times 28$ pixels and CIFAR10 consists of 60,000 RGB images of size $32 \times 32$ 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 $\epsilon$. 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 $L_\infty$-norm-based perturbation regions $R_{\bar{x}, \epsilon}$ [25], where $R_{\bar{x}, \epsilon}$ contains all perturbed inputs $\bar{x}'$ where each pixel $\bar{x}'_i$ has a distance of at most $\epsilon$ from the corresponding pixel $\bar{x}_i$ in the original input $\bar{x}$. $R_{\bar{x}, \epsilon}$ 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 $\epsilon$.

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.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Model Type</th>
<th>Model</th>
<th>#Hidden units</th>
</tr>
</thead>
<tbody>
<tr>
<td>MNIST</td>
<td>FFNNSmall</td>
<td>fully connected</td>
<td>610</td>
</tr>
<tr>
<td></td>
<td>FFNNBig</td>
<td>fully connected</td>
<td>3010</td>
</tr>
<tr>
<td></td>
<td>ConvSmall</td>
<td>convolutional</td>
<td>3604</td>
</tr>
<tr>
<td></td>
<td>ConvMed</td>
<td>convolutional</td>
<td>4804</td>
</tr>
<tr>
<td></td>
<td>ConvBig</td>
<td>convolutional</td>
<td>34688</td>
</tr>
<tr>
<td></td>
<td>ConvSuper</td>
<td>convolutional</td>
<td>88500</td>
</tr>
<tr>
<td>CIFAR10</td>
<td>FFNNBig</td>
<td>fully connected</td>
<td>3010</td>
</tr>
<tr>
<td></td>
<td>ConvSmall</td>
<td>convolutional</td>
<td>4852</td>
</tr>
<tr>
<td></td>
<td>ConvBig</td>
<td>convolutional</td>
<td>62464</td>
</tr>
</tbody>
</table>

Figure 3.3: Comparing the performance and precision of DeepZ with the state of the art. [170] and AI² 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 AI² 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 $\epsilon$ 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 $\epsilon$; 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 $\epsilon$. On the other hand, AI² 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.
3.3 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 $\leq 22$ seconds and proves 95% of the robustness properties for $\epsilon = 0.04$ for the defended PGD$_{\epsilon=0.3}$ network. DeepZ is able to analyze all ConvSmall networks with average runtime $\leq 2$ 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

![Verified robustness](image1)

![Time (s)](image2)
Table 3.2: Verified robustness by DeepZ on the ConvBig and ConvSuper networks trained with DiffAI.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Model</th>
<th>$\epsilon$</th>
<th>% verified robustness</th>
<th>average runtime(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MNIST</td>
<td>ConvBig</td>
<td>0.1</td>
<td>97</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>ConvBig</td>
<td>0.2</td>
<td>79</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>ConvBig</td>
<td>0.3</td>
<td>37</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>ConvSuper</td>
<td>0.1</td>
<td>97</td>
<td>133</td>
</tr>
<tr>
<td>CIFAR10</td>
<td>ConvBig</td>
<td>0.006</td>
<td>50</td>
<td>39</td>
</tr>
</tbody>
</table>

Verified robustness

![Verified robustness](image)

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
Figure 3.6: Verified robustness by DeepZ on the CIFAR10 networks with ReLU activations. The 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 $\epsilon$ and is not plotted. DeepZ has an average runtime of $\leq 35$ and $\leq 22$ seconds on all FFNNBig and ConvMed networks, respectively.

Fig. 3.6 shows that DeepZ has an average runtime of $\leq 50$ 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 $\leq 3$ seconds on all ConvSmall networks. DeepZ is able to verify 50% of robustness...
Verified 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.

DeepZ verifies 82% of robustness properties on the Sigmoid network defended with PGD for \( \epsilon = 0.0078 \) and PGD for \( \epsilon = 0.0313 \) as shown in Table 3.2. It verifies 46% of the robustness properties on the FFNNBig network with Tanh activation trained using PGD for \( \epsilon = 0.0313 \) for the same \( \epsilon \). The average runtime of DeepZ on all networks is \( \leq 90 \) seconds.

3.3.3 Dataset Normalization

For each dataset we include a normalization layer (which gets applied after the \( \epsilon \)-sized box has been calculated) using an approximated mean \( \mu \) and standard deviation \( \sigma \) 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 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 $\epsilon$ 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 $\text{Act} = \text{ReLU}, \text{Sigmoid}, \text{Tanh}$.

$$x \rightarrow \text{Conv}_{2}16 \times 4 \times 4 \rightarrow \text{Act} \rightarrow \text{Conv}_{2}32 \times 4 \times 4 \rightarrow \text{Act} \rightarrow \text{FC} \ 1000 \rightarrow z.$$  

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

$$x \rightarrow \text{Conv}_{1}32 \times 3 \times 3 \rightarrow \text{ReLU} \rightarrow \text{Conv}_{2}32 \times 4 \times 4 \rightarrow \text{ReLU} \rightarrow \text{Conv}_{1}64 \times 3 \times 3 \rightarrow \text{ReLU} \rightarrow \text{Conv}_{2}64 \times 4 \times 4 \rightarrow \text{ReLU} \rightarrow \text{FC} \ 512 \rightarrow \text{ReLU} \rightarrow \text{FC} \ 512 \rightarrow z.$$  

**CONVSUPER** Our largest convolutional network with no padding.

$$x \rightarrow \text{Conv}_{1}32 \times 3 \times 3 \rightarrow \text{Conv}_{1}32 \times 4 \times 4 \rightarrow \text{Conv}_{1}64 \times 3 \times 3 \rightarrow \text{Conv}_{1}64 \times 4 \times 4 \rightarrow \text{FC} \ 512 \rightarrow \text{ReLU} \rightarrow \text{FC} \ 512 \rightarrow z.$$  

### 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 $L_\infty$-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
PSI: EXACT SYMBOLIC INFEREN EC 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](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.
```python
def ClickGraph(){
    simAll := Uniform(0,1);
    clicksA := [1, 1, 1, 0, 0];
    clicksB := [1, 1, 1, 0, 0];
    for i in [0..5) {
        sim := Bernoulli(simAll);
        p1 := Uniform(0,1); p2 := p1;
        if sim {
            p1 = Uniform(0,1); p2 = p1;
        } else {
            p1 = Uniform(0,1); p2 = Uniform(0,1);
        }
        clickA := Bernoulli(p1);
        clickB := Bernoulli(p2);
        observe(clickA==clicksA[i]);
        observe(clickB==clicksB[i]);
    }
    return simAll;
}
```

**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 from the beginning of the function. As a preprocessing 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 \(\text{simAll}\), which has the \(\text{Uniform}(0,1)\) distribution. In general, for \(x := \text{Uniform}(a,b)\), the analysis generates the expression \([a \leq x] \cdot [x \leq b]/(b - a)\), which denotes the density of this distribution. The factors \([a \leq x]\) and \([x \leq b]\) 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 \(\text{simAll}\), the analysis generates \(e_{L.2} = [0 \leq \text{simAll}] \cdot [\text{simAll} \leq 1]\).

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 \(\text{sim}\) to the state, and multiplies the expression \(e_{L.2}\) with the density function for the distribution \(\text{Bernoulli(\text{simAll})}\):

\[
e_{L.7} = [0 \leq \text{simAll}] \cdot [\text{simAll} \leq 1] \cdot (\text{simAll} \cdot \delta(1 - \text{sim}) + (1 - \text{simAll}) \cdot \delta(\text{sim}))
\]

\(\bar{e}_{L.7} = 0\)
The expression $e_{L.7}$ represents a generalized joint probability density over $\text{simAll}$ and $\text{sim}$. 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 $[0 \leq \text{simAll}] \cdot [\text{simAll} \leq 1]$ 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: $\text{sim}$, $\text{p1}$, $\text{p2}$, $\text{clickA}$, and $\text{clickB}$. The analysis integrates out these local variables because they will not be referenced by the subsequent computation. It first removes the discrete variables $\text{sim}$, $\text{clickA}$, and $\text{clickB}$ by exploiting the properties of Dirac deltas. For the continuous variables $\text{p1}$ and $\text{p2}$, 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 $\text{simAll}$:

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

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 $\text{FromMarginal}(e)$ and $\text{Expectation}(e)$ return
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 INFERENCEx

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.
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 $\pi$), logarithms and uninterpreted functions. These terms can form sums, products, or exponents. Division is handled using the rewrite $a/b \rightarrow a \cdot b^{-1}$.

- **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.
The translation to this intermediate language from the mathematical definition of parameters should satisfy. Both inputs and the parameters can be random quantities.

The resulting distribution using the distribution in PSI’s intermediate language from Fig. 5.5, using the SampleFrom(pdf_expression,...) primitive, which takes as inputs a distribution expression and a set of its parameters.

**Program State**

A symbolic program state \( \sigma \) denotes a probability distribution over the program variables with an additional error state:
\[ A_e : \text{Expr} \rightarrow (\Sigma \rightarrow \Sigma \times \mathbb{E}) \]
\[ A_e(x) := \lambda \sigma. (\sigma, x) \]
\[ A_e(se_1 \ bop \ se_2) := \lambda \sigma. \text{let} \ (\sigma_1, e_1) = A_e(se_1)(\sigma) \text{ and } (\sigma_2, e_2) = A_e(se_2)(\sigma_1) \]
\[ \text{in } (\sigma_2, e_1 \ \text{SymbolicOp}(bop) \ e_2), \ \ bop \in \{+, -, *, \} \]
\[ A_e(se_1 / se_2) := \lambda \sigma. \text{let} \ (\sigma_1, e_1) = A_e(se_1)(\sigma) \text{ and } (\sigma_2, e_2) = A_e(se_2)(\sigma) \]
\[ \text{in } (\lambda s. e_2 \neq 0)(\sigma_2), e_1 \cdot e_2^{-1} \]
\[ A_e(se_1 \ cop \ se_2) := \lambda \sigma. \text{let} \ (\sigma_1, e_1) = A_e(se_1)(\sigma) \text{ and } (\sigma_2, e_2) = A_e(se_2)(\sigma) \text{ in } \\
(\sigma_2, \text{TranslateCop}(\text{cop}, e_1, e_2)) \]
\[ \text{where} \]
\[ \text{cop} \in \{==, \neq, <, >, \leq, \geq\}, \]
\[ \text{TranslateCop}(==, e_1, e_2) := [e_1 = e_2], \text{TranslateCop}(\neq, e_1, e_2) := [e_1 \neq e_2], \]
\[ \text{TranslateCop}(<, e_1, e_2) := [e_1 < e_2], \text{TranslateCop}(>, e_1, e_2) := [e_2 < e_1], \]
\[ \text{ TranslateCop}(\leq, e_1, e_2) := [e_1 \leq e_2], \text{TranslateCop}(\geq, e_1, e_2) := [e_2 \leq e_1]. \]
\[ A_e(\text{!se}) := \lambda \sigma. \text{let} \ (\sigma_1, e) = A_e(\text{se}) \text{ in } (\sigma_1, [e = 0]) \]
\[ A_e(\text{se}_1 \ & \ & \text{se}_2) := \lambda \sigma. \text{let} \ (\sigma_1, e_1) = A_e(\text{se}_1)(\sigma) \text{ and } (\sigma_2, e_2) = A_e(\text{se}_2)(\sigma_1) \]
\[ \text{in } (\sigma_2, [e_1 \neq 0] \cdot [e_2 \neq 0]) \]
\[ A_e(\text{se}_1 \ | \ \text{se}_2) := \lambda \sigma. \text{let} \ (\sigma_1, e_1) = A_e(\text{se}_1)(\sigma) \text{ and } (\sigma_2, e_2) = A_e(\text{se}_2)(\sigma_1) \]
\[ \text{in } (\sigma_2, [e_1 \neq 0] + [e_2 \neq 0] 
eq 0) \]
\[ A_e(\text{Dist}(\text{se}_1, \ldots, \text{se}_n)) := \lambda \sigma. \text{let} \ (\sigma_1, [e_1, \ldots, e_n]) = A_e^{*}(\text{se}_1, \ldots, \text{se}_n)(\sigma) \text{ and } \text{FreshVar}(\tau) \]
\[ \text{and } (P, C) = (\text{PDF}_{\text{Dist}}(\tau, e_1, \ldots, e_n), \text{Conditions}_{\text{Dist}}(e_1, \ldots, e_n)) \]
\[ \text{in let } \sigma_2 = (\text{Distribute}(\tau, P) \circ \text{Assert}(C))(\sigma_1) \text{ in } (\sigma_2, \tau), \]

\[ \text{Assert}(e[x_1, \ldots, x_n])^{(\lambda M. \text{case } M \text{ of } (x_1, \ldots, x_n) \Rightarrow e_1[x_1, \ldots, x_n], \bot \Rightarrow e_2)} := \\
\lambda M. \text{case } M \text{ of } (x_1, \ldots, x_n) \Rightarrow (e_1 \cdot [e \neq 0][x_1, \ldots, x_n] \cdot \\
\bot \Rightarrow e_2 + \text{MarginalizeAll}([e = 0] \cdot e_1) \]
\[ \text{Distribute}(x, e[x_1, \ldots, x_n])^{(\lambda M. \text{case } M \text{ of } (x_1, \ldots, x_n) \Rightarrow e_1[x_1, \ldots, x_n], \bot \Rightarrow e_2)} := \\
\lambda M. \text{case } M \text{ of } (x_1, \ldots, x_n, x) \Rightarrow e_1[x_1, \ldots, x_n] \cdot e[x_1, \ldots, x_n, x] \cdot \bot \Rightarrow e_2 \]

Figure 4.4: Symbolic Analysis of Expressions

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

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 \(\sigma\) 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 $t \in \Sigma \rightarrow \Sigma \times \mathbb{E}$ on the symbolic representation. The transformer returns both a new state ($\sigma \in \Sigma$) and a result of expression evaluation ($e \in \mathbb{E}$), 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 $\text{Dist}(se_1, \ldots, se_n)$ accepts distribution parameters $se_1, \ldots, se_n$, which can be arbitrary expressions. For a primitive distribution $\text{Dist}$, the analysis obtains expressions from the mappings $PDF_{\text{Dist}}$ and $Conditions_{\text{Dist}}$ (§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

$$
A^*_{e}([], \sigma) := \lambda \sigma. (\sigma, [])
$$

$$
A^*_{e}(se : t) := \lambda \sigma. \text{let} (\sigma_1, e) = A_e(se)(\sigma) \text{ and } (\sigma_2, t') = A^*_{e}(t)(\sigma_1) \text{ in } (\sigma_2, e : t').
$$

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

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

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

The function \text{KeepOnly} performs selective marginalization. It takes as input the variables \(x'_1, \ldots, x'_m\) to keep and the input state \(\sigma\), and marginalizes out the remaining variables in \(\sigma\)'s distribution expressions:

\[
\text{KeepOnly}(x'_1, \ldots, x'_m)(\lambda M. \text{ case } M \text{ of } (x_1, \ldots, x_n) \Rightarrow e_1[x_1, \ldots, x_n], \perp \Rightarrow e_2) = \\
\quad \text{ let } \{x''_1, \ldots, x''_m\} = \{x_1, \ldots, x_n\} \setminus \{x'_1, \ldots, x'_m\} \\
\quad \text{ in } \lambda M. \text{ case } M \text{ of } (x''_1, \ldots, x''_m) \Rightarrow \int_{\mathbb{R}} \cdots \int_{\mathbb{R}} dx''_1 \cdots dx''_m e_1[x_1, \ldots, x_n], \perp \Rightarrow e_2
\]
Rename(x, x')(\lambda M. case M of (x_1, ..., x_n) \Rightarrow e_1 \[x_1, ..., x_n\], \bot \Rightarrow e_2) :=
\lambda M. case M of (x_1, ..., x', ..., x_n) \Rightarrow e_1 \[x_1, ..., x', x_n\], \bot \Rightarrow e_2

Observe(e[\[\tau/x\]])(\lambda M. case M of (\overline{x}) \Rightarrow e_1 \[\overline{x}\], \bot \Rightarrow e_2) :=
\lambda M. case M of (\overline{x}) \Rightarrow e_1 \[\overline{x}\] \cdot e[\[\tau/x\]], \bot \Rightarrow e_2

Join((\lambda M. case M of (\overline{x}) \Rightarrow e_1 \[\overline{x}\], \bot \Rightarrow e_2), \sigma_{\text{then}}, \sigma_{\text{else}}) :=
\begin{align*}
\text{let } & (\lambda M. case M of (\overline{x}) \Rightarrow e_1' \[\overline{x}\], \bot \Rightarrow e_2') = \text{KeepOnly}(\overline{x})(\sigma_{\text{then}}) \\
\text{and } & (\lambda M. case M of (\overline{x}) \Rightarrow e_1'' \[\overline{x}\], \bot \Rightarrow e_2'') = \text{KeepOnly}(\overline{x})(\sigma_{\text{else}}) \\
\text{in } & \lambda M. case M of (\overline{x}) \Rightarrow e_1' \[\overline{x}\] + e_1'' \[\overline{x}\], \bot \Rightarrow e_2' \[\overline{x}\] + e_2'' \[\overline{x}\] - e_2 \[\overline{x}\]
\end{align*}

Figure 4.6: Analysis of Statements - Helper Functions

**Definitions**

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

**Assignments**

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

**Observations**

Observations are handled by a call to the helper function \( \text{Observe} \) (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 \( \sigma_0 \). 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 \( \sigma_1 \) and \( \sigma_2 \). Finally, \( \sigma_1 \) and \( \sigma_2 \) 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

\[
\text{Renormalize}(\lambda M. \text{case } M \text{ of } (x_1, \ldots, x_n) \Rightarrow e_1[x_1, \ldots, x_n], \bot \Rightarrow e_2) := \\
\text{let } e_Z = \text{MarginalizeAll}(e_1) + e_2 \\
\text{in } \lambda M. \text{case } M \text{ of } (x_1, \ldots, x_n) \Rightarrow [e_Z \neq 0] \cdot e_1[x_1, \ldots, x_n] \cdot e_Z^{-1}, \\
\bot \Rightarrow [e_Z \neq 0] \cdot e_2 \cdot e_Z^{-1} + [e_Z = 0]
\]

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 \( N_{\text{max}} \), a developer can encode the loop as

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

To handle for-loops with unbounded random variables and general while-loops, a developer can select \( N_{\text{max}} \) such that the probability of error (i.e., probability that the loop runs for more than \( N_{\text{max}} \) 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 \( f(x_1, \ldots, x_n) \) 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 \( \varphi(x_1, \ldots, x_n) \) 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 \( e_f[x_1', \ldots, x_m'] \) and \( e'_f \) each containing \( \varphi \), 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

\[
\sigma_0 := (\lambda M. \text{case } M \text{ of } \allowbreak (x_1, \ldots, x_n) \Rightarrow \varphi(x_1, \ldots, x_n) \allowbreak \downarrow \Rightarrow 0).
\]

and extracting the two distribution expressions from the final state.

The rule for the analysis of function calls first creates temporary variables \(a_1, \ldots, a_n\) for each argument of \(f\), and variables \(r_1, \ldots, r_m\) for each result returned by \(f\). The variables \(a_1, \ldots, a_n\) are then initialized by the actual parameters \(e_1, \ldots, e_n\) 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 \(r_1, \ldots, r_m\), and the uninterpreted function \(\varphi\) 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

\[
\begin{align*}
A_e(f(se_1, \ldots, se_n)) := & \lambda \sigma. \text{let } \text{FreshVars}(a_1, \ldots, a_n) \text{ and } \text{FreshVars}(r_1, \ldots, r_m) \\
& \text{and } e''_f[r_1, \ldots, r_m] := e_f[r_1/x'_1] \cdots [r_m/x'_m] \\
& \text{and } \sigma' := (A_s(a_n := se_n) \circ \cdots \circ A_s(a_1 := se_1)) (\sigma) \\
& \text{and } \sigma'' := \lambda M. \text{case } M \text{ of } \\
& (x_1, \ldots, x_k, r_1, \ldots, r_m) \Rightarrow \\
& e''_f[\lambda(a_1, \ldots, a_n). e_c[x_1, \ldots, x_k, a_1, \ldots, a_n]]/\varphi], \\
& \downarrow \Rightarrow e'_c + \int_{\mathbb{R}^k} dx_1 \ldots dx_k e'_f[\lambda(a_1, \ldots, a_n). e_c[x_1, \ldots, x_k, a_1, \ldots, a_n]]/\varphi] \\
& \text{in } (\sigma'', (r_1, \ldots, r_m)) \\
\end{align*}
\]

where \(e_c[x_1, \ldots, x_k, a_1, \ldots, a_n]\) and \(e'_c\) are expressions with \(\sigma' = \lambda M. \text{case } M \text{ of } (x_1, \ldots, x_k, a_1, \ldots, a_n) \Rightarrow \\
\quad e_c[x_1, \ldots, x_k, a_1, \ldots, a_n] \\
& \downarrow \Rightarrow e'_c
\)

**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 \([e = 0], [e \neq 0] \text{ and } [e \leq 0]\). 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 
\([x = 0] \cdot [y \leq 0] = 0\) is transformed into 
\([x = 0] \cdot [y \leq 0] \cdot [1 \cdot 1 = 0] + [x \neq 0] \cdot [y \leq 0] \cdot [0 \cdot 1 = 0] + [x = 0] \cdot [-y \leq 0] \cdot [y \neq 0] \cdot [1 \cdot 0 = 0] + [x \neq 0] \cdot [-y \leq 0] \cdot [y \neq 0] \cdot [0 \cdot 0 = 0]\)
which is in turn simplified to 
\([x \neq 0] \cdot [y \leq 0] + [x = 0] \cdot [-y \leq 0] \cdot [y \neq 0] + [x \neq 0] \cdot [-y \leq 0] \cdot [y \neq 0].\)

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

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 $[e \leq 0], [e = 0], [e \neq 0]$) 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** $[e \leq 0]$:
  
  $[e \leq 0]$ is equivalent to $[0 \leq p] \cdot [l \leq r] + [p < 0] \cdot [r \leq l]$.

- **Iverson brackets of the form** $[e = 0]$:
  
  $[e = 0]$ is equivalent to $[l = r]$.

- **Dirac deltas $\delta(e)$**:
  
  $\delta(e)$ is equivalent to \[ \sum_{x' \in \mathbb{R}: l \leq x' \leq r} \frac{\delta(x-x')}{\frac{\partial}{\partial x} e(x'/x)} + [\frac{\partial}{\partial x} e = 0] \cdot \delta(e). \]

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 $[e \leq 0]$.

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 recures, 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 recures 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 recures 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 \( [e \leq 0] \), guard linearization is applied recursively to the expression \([p \cdot i \leq 0]\). 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 \( z_2 = r^{l_2/n}, z_1 = -z_2 \). The result depends on the kind of guard under consideration.
  - For \([e \leq 0]\): Constraint linearization is performed recursively on \([0 \leq p]\). The algorithm performs a symbolic case split on the result. For \( 0 \leq p \), 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 \([r \geq 0]\) (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 \([r \leq 0]\) is added as a summand (as all even powers are at least 0).
4.3 Symbolic Optimizations

- 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 \([0 \leq r]\). 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 \(0 \leq r\). For \(0 \leq r\), 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 \(p' = p, l' = l_1, r' = r^{m/n}\) and multiplies the result with \([0 \leq r]\) (in order to make sure \(r^{m/n}\) is well-formed). For \([e \leq 0]\), the algorithm additionally adds a summand \(p' \cdot [r < 0]\) where \(p'\) is the result of running constraint linearization recursively 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 \( \text{min}(a, b) = [a \leq b] \cdot a + [b < a] \cdot b \).) Note that the bounds of integration can also be \(-\infty\) and \(\infty\) 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 $e^{y \log|x|}$. 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 $a > 0$, \[ \int dx e^{-a \cdot x^2 + b \cdot x + c} = e^{\frac{b^2}{4a} + c} \cdot (d/dx)^{-1} \left[ e^{-x^2} \right] \left( \sqrt{a} \cdot x - \frac{b}{2\sqrt{a}} \right) + C \]

• 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 $\Gamma(a, z) = \int_{\mathbb{R}} dt [z \leq t] \cdot t^{a-1} \cdot e^{-t}$ 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 $a > 0$, \[ \int dx (d/dx)^{-1} \left[ e^{-x^2} \right](a \cdot x + b) = \frac{1}{a} (d/dx)^{-1} \left[ e^{-x^2} \right] \cdot (a \cdot x + b) - e^{-(a \cdot x + b)^2} + C \]

• 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 $\infty$ 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, $\infty$ 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](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 Bruiin-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.
- **ClinicalTrial**: 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.
• **LinearRegression**: 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.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Type</th>
<th>Dataset</th>
<th>PSI</th>
<th>Mathem.</th>
<th>Maple</th>
<th>Hakaru</th>
</tr>
</thead>
<tbody>
<tr>
<td>BurglarAlarm</td>
<td>D</td>
<td>–</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>ClinicalTrial1</td>
<td>DC</td>
<td>100/1000</td>
<td>●</td>
<td>–</td>
<td>–</td>
<td>××</td>
</tr>
<tr>
<td>CoinBias</td>
<td>DC</td>
<td>5/5</td>
<td>●</td>
<td>t/o</td>
<td>t/o</td>
<td>●n</td>
</tr>
<tr>
<td>DigitRecognition</td>
<td>D</td>
<td>784/784</td>
<td>●</td>
<td>–</td>
<td>–</td>
<td>×</td>
</tr>
<tr>
<td>Grass</td>
<td>D</td>
<td>–</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>×</td>
</tr>
<tr>
<td>HIV</td>
<td>Co</td>
<td>10/369</td>
<td>●</td>
<td>n</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>LinearRegression1</td>
<td>C</td>
<td>100/1000</td>
<td>●</td>
<td>t/o</td>
<td>t/o</td>
<td>●n</td>
</tr>
<tr>
<td>NoisyOr</td>
<td>D</td>
<td>–</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>SurveyUnbias</td>
<td>DC</td>
<td>5/5</td>
<td>●</td>
<td>n</td>
<td>t/o</td>
<td>×</td>
</tr>
<tr>
<td>TrueSkill</td>
<td>C</td>
<td>3/3</td>
<td>●</td>
<td>t/o</td>
<td>t/o</td>
<td>●n</td>
</tr>
<tr>
<td>TwoCoins</td>
<td>D</td>
<td>–</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>–</td>
</tr>
<tr>
<td>AddFun/max</td>
<td>C</td>
<td>–</td>
<td>●</td>
<td>o</td>
<td>×</td>
<td>o</td>
</tr>
<tr>
<td>AddFun/sum</td>
<td>C</td>
<td>–</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●n</td>
</tr>
<tr>
<td>BayesPointMachine</td>
<td>C</td>
<td>6/6</td>
<td>●</td>
<td>n</td>
<td>t/o</td>
<td>t/o</td>
</tr>
<tr>
<td>ClickGraph</td>
<td>DC</td>
<td>5/5</td>
<td>●</td>
<td>t/o</td>
<td>t/o</td>
<td>●n</td>
</tr>
<tr>
<td>ClinicalTrial2</td>
<td>DC</td>
<td>5/5</td>
<td>●</td>
<td>t/o</td>
<td>t/o</td>
<td>●n</td>
</tr>
<tr>
<td>Coins</td>
<td>D</td>
<td>–</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>×</td>
</tr>
<tr>
<td>Evidence/model1</td>
<td>D</td>
<td>–</td>
<td>●</td>
<td>o</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Evidence/model2</td>
<td>D</td>
<td>–</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>•</td>
</tr>
<tr>
<td>LearningGaussian</td>
<td>Co</td>
<td>100/1000</td>
<td>●</td>
<td>n</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>MurderMystery</td>
<td>D</td>
<td>–</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>

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 (●∫), 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 e61cc7200b5caeldee33bee26daa53c0599f0bc), 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
evaluation

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
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.
**4.4 Evaluation**

**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 f4ce11f7d489933b36a43c753710bd14ef9f77). 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>
<thead>
<tr>
<th>Benchmark</th>
<th>Expectation PSI</th>
<th>Expectation R2</th>
<th>Time PSI</th>
<th>Time R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>BurglarAlarm</td>
<td>2969983/992160802 ≈ 0.002993...</td>
<td>0.0036</td>
<td>70 ms</td>
<td>173 ms</td>
</tr>
<tr>
<td>CoinBias</td>
<td>5/12 = 0.416</td>
<td>0.417</td>
<td>100 ms</td>
<td>356 ms</td>
</tr>
<tr>
<td>Grass</td>
<td>509/719 ≈ 0.7079...</td>
<td>0.715</td>
<td>680 ms</td>
<td>336 ms</td>
</tr>
<tr>
<td>NoisyOR</td>
<td>130307/160000 ≈ 0.8144...</td>
<td>0.814</td>
<td>530 ms</td>
<td>250 ms</td>
</tr>
<tr>
<td>TwoCoins</td>
<td>(1/3, 1/3)</td>
<td>(0.324, 0.336)</td>
<td>40 ms</td>
<td>108 ms</td>
</tr>
</tbody>
</table>

**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, Gauss-Sum, and MurderMystery) Infer.NET computes the almost the same distributions (with numerical error of distribution parameters less than 0.01%). For Clinical-Trial, 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.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Distribution</th>
<th>PSI Distribution</th>
<th>Infer.NET Distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>BurglarAlarm</td>
<td>((2969983 \cdot \delta(1 - b) + 989190819 \cdot \delta(b))/992160802)</td>
<td>Bernoulli(0.002995)</td>
<td></td>
</tr>
<tr>
<td>ClickGraph</td>
<td>(\delta(s + 3)^3 \cdot [0 \leq s \leq 1])</td>
<td>Beta(1.625, 1)</td>
<td></td>
</tr>
<tr>
<td>ClinicalTrial</td>
<td>(900/102 \cdot c \cdot (1 - c)^4 \cdot t^4 \cdot (1 - t))</td>
<td>(Beta(5, 2), Beta(2, 5))</td>
<td></td>
</tr>
<tr>
<td>AddFun/max</td>
<td>(\sqrt{2}/\pi \cdot (d/dx)^{-1}<a href="r/%5Csqrt%7B2%7D">e^{-x^2}</a> \cdot e^{-r^2/2})</td>
<td>Gauss(0.56, 0.68)</td>
<td></td>
</tr>
<tr>
<td>AddFun/sum</td>
<td>((2 \cdot \sqrt{\pi})^{-1} \cdot e^{-r^2/4})</td>
<td>Gauss(0, 2)</td>
<td></td>
</tr>
<tr>
<td>MurderMystery</td>
<td>((9 \cdot \delta(1 - p) + 560 \cdot \delta(p))/569)</td>
<td>Bernoulli(0.0158)</td>
<td></td>
</tr>
</tbody>
</table>

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.
**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.
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.
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.
5.1 Motivation and Overview

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 $100 \times 100$, 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 $y \leq x$ to produce the nested posterior. Note that the
function we pass to \texttt{infer} is itself random, as it depends on the external random variable \( x \). \texttt{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 \( \lambda \text{PSI} \) 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 \( \lambda \text{PSI} \):

\[
\int dv \int dz \underbrace{\frac{1}{2} \mathbb{I}[0 \leq v] \cdot [v \leq 2] \cdot \lambda[v]}_{(a)} \cdot \frac{1}{3} \sum_{k=1}^{3} \delta(k/3)[z] \cdot \delta(v, z) [a] \cdot \frac{1}{2} \left( \delta(a_0)[x] + \delta(a_1)[x] \right). 
\]

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 \( \{1/3, 2/3, 1\} \). 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 \( \lambda \text{PSI} \) performs. A plot of the cumulative distribution function of the final result computed by \( \lambda \text{PSI} \) is shown in Fig. 5.2. The full symbolic expression for this final result computed by \( \lambda \text{PSI} \) (ignoring errors) is given by

\[
\lambda x, y. \mathbb{I}[0 \leq y] \cdot [y \leq 1] \cdot \lambda[y] \cdot \left( \frac{1}{2} \cdot [y \leq \frac{1}{2}] \cdot \delta(\frac{1}{6})[x] \right) + \frac{1}{4} \cdot [0 \leq x] \cdot [x < \frac{1}{2}] \cdot [y \leq 2x] \cdot \frac{1}{x} \cdot \lambda[x] + \frac{1}{4} \cdot [y \leq \frac{2}{3}] \cdot \delta(\frac{1}{3})[x] + \frac{5}{12} \cdot \delta(\frac{1}{2})[x].
\]

5.2 \textbf{THE \textsc{psi} PPL}

We next describe the higher-order probabilistic programming language \( \lambda \text{PSI} \), 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 \( \lambda \text{PSI} \), which is
def main(){
a := (uniform(0,2), uniformInt(1,3)/3);
x := a[flip(1/2)];
p := infer({
y := uniform(0,1);
observe(y <= x);
return y;
});
return (expectation(p), sample(p));
}

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 | bop Ex | (Ex, ..., Ex) | Ex[Ex] |  
     (x: Type, ..., x: Type) Body | Ex(Ex)  
Stmt ::= x := Ex; | Ex = Ex; | return Ex; |  
     observe(Ex); | coobserve(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.
\[\text{Prog} ::= \text{Decl}^*\]
\[\text{Decl} ::= \text{Func} | \text{VarDecl}\]
\[\text{Func} ::= \text{def} \ x \ \text{ParamList}^*:(\text{Ex})^2 \ \text{Body}\]
\[\text{ParamList} ::= (\text{Param}^*,(,)^2) | [\text{Param}^*,(,)^2]\]
\[\text{Param} ::= x: \text{Ex}\]
\[\text{Body} ::= \{ \text{Stmt}^* \} \Rightarrow \text{Ex};\]
\[\text{Type} ::= x | * | \mathbb{B} | \mathbb{N} | \mathbb{Z} | \mathbb{Q} | \mathbb{R} | 1 | \text{Ex}^\text{Ex} | (\text{Ex} \times \cdots \times \text{Ex}) | \text{Ex}[] | (\text{Ex} \rightarrow \text{Ex}) | (\prod_{x: \text{Ex}} \text{Ex}[x])\]
\[\text{VarDecl} ::= x := \text{Ex};\]
\[\text{Lambda} ::= \text{ParamList}^* \ \text{LambdaBody}\]
\[\text{LambdaBody} ::= \{ \text{Stmt}^* \} \Rightarrow \text{Ex}\]
\[\text{Ex} ::= n \ | \ x \ | \text{BuiltIn} \ | (\text{Ex}) \ | (\text{Ex}: \text{Ex}) \ | (\text{uop} \ \text{Ex}) \ | (\text{Ex} \ \text{bop} \ \text{Ex}) \ |
\ (\ | (\text{Ex},) \ | (\text{Ex}, \text{Ex},(\text{Ex}),^2) \ | \ [ ] \ | [\text{Ex},(\text{Ex}),^2] \ | \text{Ex}.\text{length} \ | \text{Ex}[\text{Ex}] \ | \text{Ex}(\text{Ex}) \ | \text{Lambda} \ | \text{Type}\]
\[\text{AssignLhs} ::= x \ | \text{AssignLhs}[\text{Ex}] \ | (\text{AssignLhs}: \text{Ex}) \ | (\text{AssignLhs}, \ldots, \text{AssignLhs},)\]
\[\text{Stmt} ::= \text{AssignLhs} = \text{Ex}; \ | \text{Decl} \ | \text{return} \ \text{Ex}; \ | \text{observe}(\text{Ex}); \ | \text{cobserve}(\text{Ex}, \text{Ex}); \ | \text{if} \ \text{Ex} \ \{ \text{Stmt}^* \} \ \text{else} \ \{ \text{Stmt}^* \}^2 \ | \text{for} \ x \ \text{in} \ [n..n] \ \{ \text{Stmt}^* \} \ | \text{assert}(\text{Ex});\]

\[n \in \{0,1,2,3,\ldots\}\]
\[x \in \text{Vars}\]
\[\text{bop} \in \{+, -, *, \%, /, ^, \&\&, |, ==, !=, <, >, <=, >=\}\]
\[\text{uop} \in \{+, -, !\}\]

\[\text{BuiltIn} \in \{\text{Distribution, infer, sample, expectation, array, \pi, exp, log, floor, ceil, \ldots}\} \cup \{\text{Flip, Gauss, Uniform, UniformInt, Categorical, Exponential, Dirac, \ldots}\}\]

Figure 5.4: Full syntax of the higher-order probabilistic programming language \(\lambda\text{PSI}\) (extension of Fig. 5.3).

We support dependent types \(\text{Ex}^\text{Ex}\) 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 \(\mathbb{R} \rightarrow \mathbb{N} \subseteq \mathbb{Z} \rightarrow \mathbb{Q}\)). We do not distinguish types and other expressions in our grammar. Types are compared modulo partial evaluation of numerical expressions.

**Programs and Functions** A \(\lambda\text{PSI}\) 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

```python
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) ⇒ (x: 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 \times 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)`).
\( e ::= x \mid q \mid e \pi \mid -e \mid e_1 + \ldots + e_n \mid e_1 \ldots \ldots e_n \mid e_1^{e_2} \mid \log(e) \mid [e] \mid [e] \mid \\
[| {e_1 = e_2} ] \mid [e_1 \leq e_2] \mid [e_1 \neq e_2] \mid [e_1 < e_2] \mid \sum_{x \in \mathbb{Z}} e[x] \mid \int dx e[x] \mid (d/dx)^{-1}[e^{-x^2}] \mid \\
(e_1, \ldots, e_k) \mid [x \mapsto e[x]](e) \mid e_{1 e_2} \mid e_{1 e_2 \rightarrow e_3} \mid \{ f_1 \mapsto e_1, \ldots, f_k \mapsto e_k \} \mid e.f | e_1[f \mapsto e_2] \mid \\
\lambda x. e[x] \mid e_{1 (e_2)} \mid \lambda x. e(x) \mid e[x] \mid \delta(e)[x] \mid \lambda[x] \mid e_1 / e_2 \mid \bot \mid e_1?(e_2) \)

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 \( \text{\texttt{\lambda PSI}} \) and new compared to PSI.

Statements. Our language distinguishes variable declarations (e.g., \( x:=3 \)) and assignments (e.g., \( x=3 \)). The \texttt{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 \texttt{observe (x>=2)}; conditions the distribution on program states on the observed fact that \( x \) is at least 2.

The \texttt{cobserve} (“continuous observation”) statement is used to condition on a possible, but probability-zero event. In particular, \texttt{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, \( \text{\texttt{\lambda PSI}} \) supports standard \texttt{if} statements, and \texttt{for} loops with statically-known bounds.

First-class inference. The built-in \texttt{infer} function enables us to perform nested inference. It reifies a function \( f \) with return type \( \texttt{a} \) to a \texttt{Distribution[a]}, for any type \( a \). If \( f \) does not execute \texttt{observe} statements, \texttt{infer} can be thought of as the inverse of sample. This is because \texttt{infer()} \( \Rightarrow \texttt{sample(d)} \) returns the distribution \( d \) and \( () \Rightarrow \texttt{sample(infer(f))} \) yields the function \( f \). Otherwise, \texttt{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 \texttt{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, \( \text{\texttt{\lambda PSI}} \) simplifies representations of distributions in this domain. In §5.4, we will see how any \( \text{\texttt{\lambda PSI}} \) program is translated to this domain.
5.3.1 Representation

\( \lambda \)PSI’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 \( \pi \), 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 \( \lambda \)PSI’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 \( \{ f \mapsto 1 \} \) is a record with one field \( f \), 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. \( a[f \mapsto b] \)) represents a modification of a tuple or array (resp. record) \( a \) where the \( i \)-th value (resp. the value of field \( f \)) 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 (\( \Lambda x.e[x] \)). For example, \( \lambda x.f(x) \) is the same as the function \( f \), while \( \Lambda x.p[x] \) 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 \( \lambda \)PSI 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 $\parallel$ 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 $\bot$ denotes a special error value, and $\delta(\bot)[x]$ 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 $e_2 \neq \bot$, while $e_1?(\bot)$ reduces to $\delta(\bot)$.

### 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 $\tau$ (e.g., $\mathbb{R}$) is a bounded measure on $\tau$ (it is not necessarily normalized) and we write $D[\tau]$ to denote the set of all distributions over $\tau$. 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 $\tau$ can be formally interpreted as a random variable: for any $S \subseteq \tau$, it is

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

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 $\tau, \tau'$, distribution $f \in D[\tau]$, and function $g \in \tau \rightarrow \tau'$, 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 $\tau$ 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 \( \tau \), we can e.g. write

\[
\forall \tau. \ h[\tau] = \forall \tau. \int dx \ x[\tau] \cdot f[x].
\]

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 \( \tau \), \( \delta: \tau \to D[\tau] \) is defined as

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

The expression \( \forall x. \delta(v)[x] \) 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 \( \infty \) 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., \text{flip}(1/3)) can be written as

\[
\forall x. \ \text{Bernoulli}(\frac{1}{3})[x] := \forall x. \ \frac{2}{3}\delta(0)[x] + \frac{1}{3}\delta(1)[x].
\]

Note that as expected, the probability of value 1 is

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

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

\[
\forall x. \ \sum_{i \in \mathbb{Z}} [i \geq 0] \cdot \left(\frac{3}{4}\right)^i \cdot \frac{1}{4} \cdot \delta(i)[x]. \tag{5.1}
\]

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

\[
\forall x. \ [0 \leq x] \cdot 2e^{-2x} \cdot \lambda[x].
\]
Variable read \( x \) = \( \lambda \sigma. \lambda \tau. \delta(\sigma; x)[\tau] \) (5.5)

Binary operation \( a + b \) = \( \lambda \sigma. \lambda \tau. \int dz \sigma[[\tau]] \cdot \delta(\sigma + \tau)[z] \cdot \delta(y + z)[x] \) (5.6)

Assignment \( x = e \) = \( \lambda \sigma. \lambda \tau. \int dx' \delta(\sigma; x') \cdot \delta(\sigma(x \mapsto x'))[\sigma'] \) (5.7)

Seq. composition \( A; B \) = \( \lambda \sigma. \lambda \tau. \int d\sigma'' \int A\sigma[\sigma'] \cdot B\sigma''[\sigma'] \) (5.8)

Observation \( \text{observe}(e) \) = \( \lambda \sigma. \lambda \tau. \delta(\sigma; \cdot) \cdot p(\sigma) \) where \( p(\sigma) := \int dx \sigma[[\tau]][x] \cdot \delta(x) \) (5.9)

Continuous obs. \( A; \text{observe}(b,c) \) = \( \lambda \sigma. \lambda \tau. \int dy \left( \int A\sigma[\sigma'][y][\lambda[y]] \cdot \delta(\sigma')[y] \right) \) (5.10)

Control flow \( \text{if } e \text{ then } (A) \text{ else } (B) \) = \( \lambda \sigma. \lambda \tau. \int d\sigma'' \int A\sigma[\sigma'][\lambda[y]] \cdot \delta(\sigma'')[y] \) (5.11)

Scoping \( [A] \) = \( \lambda \sigma. \lambda \tau. \int d\sigma'' \int A\sigma[\sigma'][\lambda[y]] \cdot \delta(\sigma'' \cdot [x : \text{variable } x \text{ introduced in } \lambda])[\sigma'] \) (5.12)

Inference \( \text{infer}(f) \) = \( \lambda \sigma. \lambda \tau. \delta(Ay. [y]) \cdot Z^{-1}[x] \) where \( Z := \int dz \sigma[[z]] \) (5.13)

Sample \( \text{sample}(d) \) = \( \lambda \sigma. \lambda \tau. \int dx \sigma[[\tau]][x] \cdot \delta(y)[y] \) (5.14)

Expectation \( \text{expectation}(d) \) = \( \lambda \sigma. \lambda \tau. \int dx \delta(A\sigma[\sigma'][\lambda[y]] \cdot \delta(y)[y]) \cdot \delta(y)[y] \) (5.15)

Function \( (\{ A ; \text{return } e \} \) = \( \lambda \sigma. \lambda \tau. \int d\sigma'' \int A\sigma[\sigma'][\lambda[y]] \cdot \delta(y)[y] \) (5.16)

Figure 5.6: Key translation rules, ignoring error states. The rules are recursive and we write \( \mathbb{B} \) to denote the translation of \( a \).

A key property of \( \lambda \text{PSI}'s \) 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., \text{uniform}(a, b)) is represented by

\[
\lambda a, b. \lambda x. [a < x < b] \cdot \frac{1}{b-a} \cdot [a \leq x \leq b] \cdot \lambda[x] \quad (5.2)
\]

\[
+ [a = b] \cdot \delta(0)[x] \quad (5.3)
\]

\[
+ [b < a] \cdot \delta(\perp)[x]. \quad (5.4)
\]

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 \( \perp \) in part (5.4).

### 5.3.4 Comparison to \text{PSI}

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

Note that in \( \lambda \text{PSI} \), the error state \( \perp \) 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 \( \lambda \text{PSI}'s \) representation.
5.4 FROM PROGRAMS TO SYMBOLIC REPRESENTATIONS

We now show how \(\lambda\)PSI 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 \(\lambda\)PSI 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 \(\sigma\) before executing Stmt, and returns the distribution over the state \(\sigma'\) after executing Stmt in \(\sigma\). 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 \(\sigma\) and returns the distribution over the value of Ex in the state \(\sigma\).

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 \(\lambda\)PSI expressions do not have side-effects, the probabilities for the values of \(a\) and \(b\) are independent given the current state \(\sigma\).

DISTRIBUTIONS Sampling from built-in distributions (using, for example, the functions \(\text{flip}\) or \(\text{exponential}\)) 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 \(\sigma\). The new state \(\sigma'\) is equal to \(\sigma\) 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 \(\sigma'\).
observations

An observation \texttt{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 \( \sigma \). If \( e \) is deterministic, \( p(\sigma) \) is either 0 or 1. However, note that \( e \) may involve random choices such as in \texttt{observe}(uniform(0,2) < 1), where \( p(\sigma) \) is \( \frac{1}{2} \). Next, the rule rescales the probability of the current state \( \sigma \) using \( \delta(\sigma)[\sigma'] \cdot p(\sigma) \). The resulting distribution over \( \sigma' \) may not be normalized any more, but will be re-normalized later.

The effect of \texttt{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 \( x \geq 3 \) evaluates to true is \( [x \geq 3] \). 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 \( \lambda\text{PSI} \) formally derives this in a sequence of translation and simplification steps.

continuous observations

Rule (5.10) translates continuous observations of the shape \texttt{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 \( \sigma' \) and the value \( y \) of \( b \). For \texttt{cobserve} to be defined, it must be possible to rewrite this distribution such that it involves a Lebesgue measure factor \( \lambda[y] \). 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 \( \lambda\text{PSI} \) are bounded and are unrolled during translation.
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 \( \text{infer}(f) \), 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 \( \text{sample}(d) \) draws a sample from \( d \). Assume \( d \) is computed as follows (note that \( \text{Flip} \) is a distribution, while \( \text{flip} \) is a sample):

\[
d := \text{Flip}(1/2);
\text{if} \ \text{flip}(1/4) \ { \ d = \text{Flip}(1/3); } \}
\]

To compute the probability of a sample \( z \) from \( d \) we need to sum (i) the probability that \( \text{flip}(1/4) \) is false and \( z \) is generated by \( \text{Flip}(1/2) \), and (ii) the probability that \( \text{flip}(1/4) \) is true and \( z \) is generated by \( \text{Flip}(1/3) \). 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 \( \text{expectation}(d) \), we also integrate over all possible distributions \( x \). For each such distribution, we compute the expectation by the standard definition (i.e., \( \int \text{d}y \ x[y] \cdot y \)) and construct the point mass on that value. The result is a distribution over the expected value of \( d \).

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 \( \sigma' \), which comprises all variables in the function’s scope. Then, we translate \( e \) to obtain a distribution over the return value in the state \( \sigma' \). Finally, we integrate over all possible states \( \sigma' \). Note that in general, the resulting distribution may be parameterized by the function’s arguments (not shown).

The entry point for a λPSI program is its \texttt{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 \texttt{infer}). Note that the normalization constant may depend on the parameters of \texttt{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 \( \text{uniform}(a,b) \). 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 \( \sigma \) the distribution over the output state \( \sigma' \),
\[ 2 + \text{uniform}(0,3) = \lambda x. \int dy \int dz \left[ f \sigma \right][y] \cdot \text{uniform}(0,3) \left[ \sigma \right][z] \cdot \delta(y + z)[x] \] (5.17)

\[ \lambda x. \lambda y. \int dy \int dz \left[ \delta(z)[y] \cdot \frac{1}{2} \cdot [0 \leq z] \cdot [z \leq 3] \cdot \lambda \left[ \delta(2 + z)[y] \right] \right] \] (5.18)

\[ \lambda x. \lambda y. \int \left[ 0 \leq z \right] \cdot [z \leq 3] \cdot \lambda \left[ \delta(2 + z)[y] \right] \] (5.19)

\[ \lambda x. \lambda y. \int \left[ 0 \leq z \right] \cdot [z \leq 3] \cdot \lambda \left[ \delta(x - 2)[z] \right] \] (5.20)

\[ \lambda x. \lambda y. \int \left[ 2 \leq x \right] \cdot [x \leq 5] \cdot \lambda \left[ \delta \right] \] (5.21)

\[ x := 2 + \text{uniform}(0,3); \quad \text{observe}(x \gg 3); \]

\[ \lambda \sigma. \lambda \sigma' . \int d \sigma' \left[ \sigma' \cdot \text{observe}(x \gg 3) \right] \] (5.22)

\[ \lambda \sigma. \lambda \sigma' . \int d \sigma' \left[ 2 \leq [x'] \cdot [x' \leq 5] \cdot \lambda \left[ \sigma' \cdot \delta(\sigma(x \rightarrow x')) \right] \right] \] (5.23)

\[ \lambda \sigma. \lambda \sigma'' . \int d x' \left[ 2 \leq [x'] \cdot [x' \leq 3] \cdot [x' \leq 5] \cdot \lambda \left[ \delta(\sigma(x \rightarrow x')) \right] \right] \] (5.24)

\[ \lambda \sigma. \lambda \sigma'' . \int d x' \left[ 3 \leq [x'] \cdot [x' \leq 5] \cdot \lambda \left[ \delta(\sigma(x \rightarrow x')) \right] \right] \] (5.25)

\[ \lambda \sigma . \lambda \sigma''. \int d \sigma \left[ [x \gg 3] \cdot [0 \leq y] \cdot [y \leq 5] \cdot \lambda[y] \cdot \left[ \int d z [3 \leq z] \cdot [z \leq 5] \cdot \lambda(z) [z]^{-1} \right] \right] \] (5.26)

\[ \lambda \sigma . \lambda \sigma''. \int d \sigma \left[ [x \gg 3] \cdot [0 \leq y] \cdot [y \leq 5] \cdot \lambda[y] \cdot \left[ \int d z [3 \leq z] \cdot [z \leq 5] \cdot \lambda(z) [z]^{-1} \right] \right] \] (5.27)

\[ \lambda \sigma . \lambda \sigma''. \int d \sigma \left[ [x \gg 3] \cdot [0 \leq y] \cdot [y \leq 5] \cdot \lambda[y] \cdot \left[ \int d z [3 \leq z] \cdot [z \leq 5] \cdot \lambda(z) [z]^{-1} \right] \right] \] (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 \( \bot \) (similarly for expressions). Symbolic distributions make use of Dirac deltas \( \delta(\bot)[x] \) 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 \( \bot \) (as \( f \) may cause an error) and we hence must analyze the case \( \sigma = \bot \) separately. For instance, the rule for sequential composition needs to propagate errors caused in \( \Lambda \) through \( \delta \) using an expression of the form \( e_1? e_2 \):

\[ \lambda \sigma . \Lambda \sigma''. \int d \sigma' \left[ \Lambda(\sigma') \left[ \sigma'' \right] \cdot \delta(\sigma') \left[ \sigma'' \right] \right]. \]

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 \( \lambda \text{PSI} \)'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 \( \lambda \text{PSI} \)'s more powerful symbolic domain, and (ii) improve the former’s efficiency using various (low-level) optimizations.
Substitution
\[
\int dx \ f(x) \cdot \delta(v)[x] = f(v)
\]  \hspace{1cm} (5.29)

Linearization
\[
\delta(f(x))[y] \cdot \lambda[x] = [f'(x) = 0] \cdot \delta(f(x))[y] \cdot \lambda[x] \\
+ [f'(x) \neq 0] \cdot \sum_{z:f(z)=y} \delta(z)[x] / |f'(z) \cdot \lambda[y]|
\]  \hspace{1cm} (5.30)

Disintegration
\[
(e \cdot \lambda[x]) / \lambda[x] = e
\]  \hspace{1cm} (5.31)

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

Basic algebraic simplifications \( \lambda \text{PSI} \) 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 \( \lambda \text{PSI} \) expression in Fig. 5.7, while discussing a selection of interesting simplification steps (Fig. 5.8).

We start by translating and simplifying the expression \( 2 + \text{uniform}(0,3) \) (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 \text{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 \( \delta \). In general, \( \lambda \text{PSI} \) 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 \( g[z] = \frac{1}{3} \cdot [0 \leq z] \cdot [z \leq 3] \) 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 \( \delta \), 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:

\[
\lambda(y). \int dx \ [0 \leq x] \cdot [x \leq 1] \cdot \lambda[x] \delta(2x)[y].
\]

Incorrectly linearizing \( \lambda[x] \delta(2x)[y] \) to \( \lambda[y] \delta(y/2)[x] \) gives

\[
\int dx \ [0 \leq x] \cdot [x \leq 1] \cdot \lambda[y] \delta(y/2)[x] \overset{(5.29)}{=} [0 \leq y] \cdot [y \leq 2] \cdot \lambda[y]
\]

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 \([x \leq -1] + [x \geq 1] + [x = 0]\) 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
5.5 Inference by Symbolic Simplification

The function \( f(x) = 1 + [−1 ≤ x] \cdot [x ≤ 1] \cdot (1 − x^2) \)

is the same value (namely, 1) for all \( x ≤ −1 \). Hence, we can rewrite \( [x ≤ −1] \cdot \delta(f(x)) \cdot y \)

to \( [x ≤ −1] \cdot \delta(1) \cdot y \) (similarly for \( x ≥ 1 \)).

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 \( \sigma' \) is the output of a Dirac delta (see highlighted), we can again apply substitution (5.29). Note how the access of field \( x' \) in \( \sigma'.x \)

is simplified to \( x' \), because \( \sigma' \) is substituted by \( \sigma[x ↦ x'] \).

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 \( 2 ≤ x' \) is implied by \( x' ≥ 3 \), we simplify the two factors to \( [3 ≤ x'] \) in (5.25).

In addition to eliminating redundant guards, \( \lambda \)PSI 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 \( [x = 0] \cdot [x ≠ 0] \)). Also, \( \lambda \)PSI analyzes complex guard constraints (such as quadratic polynomials) to rewrite them as a combination of simpler, linear guard constraints (e.g., we rewrite \( [x^2 ≥ 4] \) as \( [x ≥ 2] + [x ≤ −2] \)). 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 \( \lambda \)PSI’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** \(\lambda\)PSI extends PSI’s powerful engine for symbolic integration of a wide class of functions not involving Dirac deltas. To simplify such integrals, \(\lambda\)PSI 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, \(\lambda\)PSI 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** \(\lambda\)PSI 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(\(1/4\)) \(\lambda\)PSI encounters the following expression (cp. (5.1)):

\[
\int dx \sum_{i \in \mathbb{Z}} [i \geq 0] \cdot (\frac{3}{4})^i \cdot \frac{1}{4} \cdot \delta(i) \cdot x.
\]

(5.29)

\[
\frac{1}{4} \cdot \sum_{i \in \mathbb{Z}} [i \geq 0] \cdot (\frac{3}{4})^i \cdot i.
\]

(5.32)

\(\lambda\)PSI 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:

```plaintext
x := gauss(\(\mu, \nu\)); cobserve(2 \cdot x, y);
```

The `cobserve` statement conditions on the possible but probability zero event that \(2 \cdot x\) 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 \(2 \cdot x\) is fixed to the observed value \(y\). We now derive these effects from our translation and simplification rules. After translation and some simplification steps (\(\sigma\) and \(\sigma'\) omitted for brevity), the distribution of \(x\) is

\[
\int dz \left( (f(x; \mu, \nu) \cdot \lambda[x] \cdot \delta(2 \cdot x)[z]) / \lambda[z] \right) \cdot \delta(y)[z]
\]

(5.30)

\[
\int dz \left( \left( \frac{1}{2} f(x; \mu, \nu) \cdot \delta(z/2)[x] \cdot \lambda[z] \right) / \lambda[z] \right) \cdot \delta(y)[z].
\]
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:

$$\int dz \frac{1}{2} f(x; \mu, \nu) \cdot \delta\left(\frac{z}{2}\right)[x] \cdot \delta(y)[z] = \frac{1}{2} f\left(\frac{y}{2}; \mu, \nu\right) \cdot \delta\left(\frac{y}{2}\right)[x]$$

In general, rule (5.31) transforms the density of the first argument of $\text{cobserve}$ to a weight for the remaining distribution. Note that due to its powerful Dirac delta linearizer, $\lambda$PSI 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, $\lambda$PSI 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 $\lambda$PSI 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 $\lambda$PSI 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, $\lambda$PSI introduces many non-trivial simplifications of sums (e.g., to simplify expectations). Disintegration (§5.5.5) is new, as PSI does not support $\text{cobserve}$.

### 5.5.7 Limitations

$\lambda$PSI’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 $\lambda$PSI 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 $\lambda$PSI’s simplification rules work well for a set of benchmark programs. We also show an example which can not be simplified by $\lambda$PSI.
Table 5.1: Probabilistic programs used in evaluation (31 in total). For each program, we indicate if it involves higher-order functions (→), nested inference (⇝), first-class expectations (𝔼), continuous distributions (λ), continuous observations (ψ), or symbolic parameters (Π). SocialCognition and TotalVarDist have multiple variants. Some programs are not expressible in Hakaru (−), while others lead to errors (∗), unsimplified (■) or incorrect (∗∗) results. aNot directly expressible; rewritten as first-order programs without function calls, multiple manual steps. bFor concrete instantiation of symbolic parameters.

<table>
<thead>
<tr>
<th>Program(s)</th>
<th>Description</th>
<th>Features</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>SocialCognition (12) [64]</td>
<td>Multiple rational agent models (see §5.1)</td>
<td>→ E λ Π</td>
<td>&lt;5.5s</td>
</tr>
<tr>
<td>CondProb (Fig. 5.13)</td>
<td>Compute conditional probability using expectation operator</td>
<td>→ → E λ</td>
<td>1s</td>
</tr>
<tr>
<td>Overview (Fig. 5.2)</td>
<td>Example involving multiple language features</td>
<td>→ → E →</td>
<td>0.3s</td>
</tr>
<tr>
<td>ChannelCap</td>
<td>Mutual information between input and output of noisy channel</td>
<td>→ → E →</td>
<td>2s</td>
</tr>
<tr>
<td>Entropy</td>
<td>Entropy of randomly generated sequence</td>
<td>→ → E →</td>
<td>3s</td>
</tr>
<tr>
<td>GenCap [22]</td>
<td>Generalization capacity of sorting algorithms (see Fig. 5.14)</td>
<td>→ → E →</td>
<td>16s</td>
</tr>
<tr>
<td>AIDE [58]</td>
<td>KL-divergence between particle filter and exact inference on HMM</td>
<td>→ → E →</td>
<td>4.2s</td>
</tr>
<tr>
<td>BivariateIndep</td>
<td>Verify that bivariate distribution has independent components</td>
<td>→ → E →</td>
<td>0.1s</td>
</tr>
<tr>
<td>SecretSanta</td>
<td>Five people guess secret santa in turn, based on uniform prior</td>
<td>→ → E →</td>
<td>0.5s</td>
</tr>
<tr>
<td>TotalVarDist (2) [13]</td>
<td>Total variation distance for random walk and Dynkin process (20 steps)</td>
<td>→ → E →</td>
<td>&lt;5.5s</td>
</tr>
<tr>
<td>MontyHall</td>
<td>Monty hall problem variants modeled using nested inference</td>
<td>→ → E →</td>
<td>0.1s</td>
</tr>
<tr>
<td>Variance</td>
<td>Compute variance of given distribution</td>
<td>→ → E →</td>
<td>0.5s</td>
</tr>
<tr>
<td>CDF</td>
<td>Compute CDF of Gaussian distribution at a point drawn from it</td>
<td>→ → E →</td>
<td>0.1s</td>
</tr>
<tr>
<td>GANLoss</td>
<td>GAN loss for simple probabilistic model against optimal discriminator</td>
<td>→ → E →</td>
<td>0.7s</td>
</tr>
<tr>
<td>FairSVM [164]</td>
<td>Infer weights for fair SVM classifier</td>
<td>→ → E →</td>
<td>0.2s</td>
</tr>
<tr>
<td>BayesLinReg [75]</td>
<td>Bayesian linear regression from 5 data points with Gaussian noise</td>
<td>→ → E →</td>
<td>24s</td>
</tr>
<tr>
<td>BayesPiecewiseLR [76]</td>
<td>Bayesian piecewise linear regression from 7 data points</td>
<td>→ → E →</td>
<td>33s</td>
</tr>
<tr>
<td>DisintegrateLinear [148]</td>
<td>Motivating example from [148], disintegrate linear function two ways</td>
<td>→ → E →</td>
<td>48s</td>
</tr>
<tr>
<td>DisintegrateQuadratic</td>
<td>Disintegrate quadratic function (involves \texttt{observe}(x-1)^2,y))</td>
<td>→ → E →</td>
<td>0.1s</td>
</tr>
</tbody>
</table>

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 \(\lambda PSI\)** We can express all programs succinctly in \(\lambda PSI\), 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 \(\lambda PSI\)’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 \(\lambda PSI\) for the same reason:

```python
def main() ⇒ gauss(0, 1)*gauss(0, 1) < 1;
```

For the remaining 30 examples, \(\lambda PSI\) successfully infers a closed-form exact result (no integrals left) within at most 42 seconds. We conclude that \(\lambda PSI\) is powerful enough to express interesting applications and that its simplification engine is effective.

### 5.6.1 Comparison to Previous Work

We compare \(\lambda PSI\) 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 \(\lambda PSI\) 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 \(\lambda PSI\), `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
\begin{lstlisting}[language=Python]
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);
    }
}
\end{lstlisting}

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 \(\lambda\text{PSI}\), 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 \(\lambda\text{PSI}\) to compute the posterior density \(p(y) \mid y = f(x)\) in terms of \(x\). Fig. 5.11 shows how to encode this inference problem in \(\lambda\text{PSI}\). 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 $Pr[A | B]$ (shown in Fig. 5.13). We use Bayes’ rule directly (instead of observe). $\lambda$PSI 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-
\texttt{def PrAgB(d: Distribution[\text{R} \times \text{R}], A: \text{R} \times \text{R} \rightarrow \text{B}, B: \text{R} \times \text{R} \rightarrow \text{B})}{
  \texttt{prAB := expectation(infer(){
    x := sample(d);
    return A(x) \&\& B(x);
  })};
  \texttt{prB := expectation(infer(){
    x := sample(d);
    return B(x);
  })};
  \texttt{return prAB / prB};
}\n\texttt{def main(X,Y){
  joint := infer(){
    x := uniform(0,1);
    y := x^2 + uniform(0,1);
    return (x,y);
  });
  A := (x,y) \Rightarrow x<X;
  B := (x,y) \Rightarrow y>Y;
  return PrAgB(joint,A,B);
}\

Figure 5.13: Conditional probability $\Pr[x < X \mid y > Y]$ 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 \texttt{\lambda PSI} encoding of this task.

The \texttt{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 \texttt{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. \texttt{\lambda PSI} 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.
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.
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.

AI², 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 AI². 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.
While already powerful, AI² (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

$$\mu f. \lambda n. [0 \leq n] \cdot ([n] = n) \cdot (([n = 1] + [2 \leq n] \cdot (f(n - 1) + f(n - 2)))$$

represents the Fibonacci sequence as a symbolic expression, while

$$\mu d. \lambda x. p \cdot \delta(0)[x] + (1 - p) \cdot \int dx' \cdot d[x'] \cdot \delta(x' + 1)[x]$$

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 $\lambda n. [0 \leq n] \cdot ([n] = n) \cdot (((((1 + \sqrt{5})/2)^n - ((1 - \sqrt{5})/2)^n)/\sqrt{5})$ using simple and general simplification rules for recurrences. Similarly, the second term could be simplified to $\lambda x. p \cdot \sum_{i \in \mathbb{Z}} [0 < i] \cdot (1 - p)^i \cdot \delta(i)[x]$. 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

$$\lambda n. \lambda a. \int dx_0 \cdots \int dx_{n-1} \delta(x)[a] \cdot \prod_{i \in \mathbb{Z}} \frac{1}{\sqrt{2\pi}} \cdot e^{-\frac{1}{2}(x_i - [i \neq 0] \cdot x_{i-1})^2} \cdot \lambda[x_i]$$,
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

\[
\lambda_n \cdot \delta(n)[a][\text{length}] \cdot \prod_{0 \leq i < n} \int dx \frac{1}{\sqrt{2\pi}} \cdot e^{-\frac{1}{2}(x-a_{i-1})^2} \cdot \lambda[x] \cdot \delta(x)[a_i]
\]

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:

```c
def main(){
  f := (x: R) \Rightarrow \text{gauss}(0,1);
  // independent samples, probability 0 for coincidence:
  assert(\text{gaussians}(0) \neq \text{gaussians}(0));
  gaussians := memo(f); // create/sample memoized function
  // calling with the same argument always gives the same result:
  assert(\text{gaussians}(0) = \text{gaussians}(0));
  // calling with different arguments, independent samples:
  assert(\text{gaussians}(0) \neq \text{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:

```c
def main(t_0: N, t_1: N){
  step := (step: N \rightarrow R, t: N) \Rightarrow 0 \text{ if } t=0 \text{ else } \text{step}(t-1)+\text{gauss}(0,1);
  walk := memofix(step); // (or some more convenient built-in syntax)
  return walk(t_1) - walk(t_0); // (always zero if t_0=t_1)
}
```

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

\[
\text{variance(infer(())
\{ 
    x := \text{flip}(1/2);
    y := \text{choose}({0,1}); // nondeterministic choice from set \{0,1\}, made by adversary
    \text{return} \; x-y;
\});
\]

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

\[
\text{variance(infer(())
\{ 
    y := \text{choose}({0,1}); // nondeterministic choice from set \{0,1\}, made by adversary
    x := \text{flip}(1/2);
\});
\]
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 \( \frac{1}{4} \). However, actually, an adversary can choose to pick either \( 0 \) or \( 1 \) with probability \( \frac{1}{2} \) each, and in this case we get variance \( \frac{1}{2} \), which is higher.

Pushing this kind of reasoning further and adding full support for nondeterminism to PSI is an exciting future research direction.

**Quanifiers**  
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.
**Conclusion and Future Work**

**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.
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 pursuit a typed version of the symbolic domain that guarantees all expressions that can be assigned a valid type have a meaning.

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

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.

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.

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 $\epsilon$ from the exact result with probability at least $\delta$”. (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.)

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 INFEERENCE 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 INFEERENCE  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.
BIBLIOGRAPHY


[6] Niels Henrik Abel. Untersuchungen über die Reihe: 1 + (m/1)x + m-(m - 1)/(1·2)-x²+ m-(m - 1)(m - 2)/(1·2·3)-x³+ ... In Reine und angewandte Mathematik, volume 1, pages 311–339. 1826. 118


[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 Marg- 
getson, and Jurgen Van Gael. Measure transformer semantics for bayesian 
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, 
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 
nearal 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 
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 

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 Navid- 
symposium and the 13th European conference on Foundations of software engineering 
(ESEC/FSE), pages 102–112. ACM, 2011. 49

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


[51] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev. AI2: Safety and robustness certification of


[76] Chris Heunen, Ohad Kammar, Sean Moss, Adam ´Scibior, and Hongseok Yang. The semantic structure of quasi-borel spaces. In PPS’18, January 2018. 120, 122


[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


