

DISS. ETH NO. 27096

SCALABLE AUTOMATED REASONING FOR PROGRAMS AND  
DEEP LEARNING

A dissertation submitted to attain the degree of  
DOCTOR OF SCIENCES OF ETH ZURICH  
(Dr. sc. ETH Zurich)

by

GAGANDEEP SINGH

Bachelor in Computer Science and Engineering, IIT Patna  
Master in Computer Science, ETH Zurich  
born on 22.09.1989  
citizen of India

accepted on the recommendation of  
Prof. Martin Vechev (advisor)  
Prof. Markus Püschel (co-examiner)  
Prof. Patrick Cousot (co-examiner)  
Prof. Clark Barrett (co-examiner)

2020



---

## ABSTRACT

---

With the widespread adoption of modern computing systems in different real-world applications such as autonomous vehicles, medical diagnosis, and aviation, it is critical to establish formal guarantees on their correctness before they are employed in the real-world. Automated formal reasoning about modern systems has been one of the core problems in computer science and has therefore attracted considerable interest from the research community. However, the problem has turned out to be quite challenging because of the ever-increasing scale, complexity, and diversity of these systems, which has so far limited the applicability of formal methods for their automated analysis.

The central problem addressed in this dissertation is: are there generic methods for designing fast and precise automated reasoning for modern systems? We focus on two practically important problem domains: numerical software and deep learning models. We design new concepts, representations, and algorithms for providing the desired formal guarantees. We build our methodology on the elegant abstract interpretation framework for static analysis, which enables automated reasoning about infinite concrete behaviors with finite computable representations. Our methods are generic for the particular problem domain and allow precise analysis beyond the reach of prior work.

For programs, we present a new theory of online decomposition that dynamically decomposes expensive computations, thereby reducing their complexity without any precision loss. Our theory is generic and can be used for decomposing all existing subpolyhedra domains without sacrificing precision. We leverage data-driven machine learning to further improve the performance of numerical program analysis without significant precision loss. For neural networks, we designed a new abstraction equipped with custom approximations of non-linearities commonly used in neural networks for fast and scalable analysis. We also created a new convex relaxations framework that produces more precise relaxations than possible with prior work. We provide a novel combination of our abstractions and relaxation framework with precise solvers, which enables state-of-the-art certification results.

This thesis presents two new publicly available software systems: ELINA and ERAN. ELINA provides optimized implementations of the popular Polyhedra, Octagon, and Zone domain, based on our theory of online decomposition enabling fast and precise analysis of large Linux device drivers containing  $> 500$  variables in a few seconds. ERAN contains our custom abstraction, convex relaxation framework, and combination of relaxations with solvers for enabling fast and precise analysis of large neural networks, containing tens of thousands of neurons, within

a few seconds. Both the systems were developed from scratch, and are currently state-of-the-art for their respective domains, producing results not possible with other competing systems.

---

## ZUSAMMENFASSUNG

---

Durch die verbreitete Nutzung moderner Computersysteme in verschiedenen Anwendungen, wie zum Beispiel autonomen Fahrzeugen, der medizinischen Diagnostik und der Luftfahrt, ist es entscheidend geworden die formale Korrektheit von diesen Computersystemen zu beweisen, bevor sie in der echten Welt eingesetzt werden. Die automatisierte Anwendung Formaler Methoden für moderne Systeme ist eines der Kernprobleme der Informatik und hat daher in der Forschungsgemeinschaft grosses Interesse geweckt. Aufgrund des ständig zunehmenden Umfangs, der Komplexität und Vielfalt dieser Systeme, ist die Anwendbarkeit Formaler Methoden bisher eingeschränkt gewesen. Daher hat sich dieses Problem als schwierig erwiesen.

Das zentrale Problem, das in dieser Dissertation behandelt wird, lautet: Gibt es generische Methoden um schnelle und präzise automatisierte Korrektheitsbeweise für moderne Systeme zu entwerfen? Wir konzentrieren uns auf zwei praxisrelevante Problembereiche: numerische Software und künstliche neuronale Netze. Wir entwerfen neue Konzepte, Repräsentationen und Algorithmen um die gewünschten formalen Garantien zu erhalten. Wir bauen unsere Methodik auf der eleganten Theorie der abstrakten Interpretation für statische Analyse auf, welche automatisierte Beweisführung über unendliche, konkrete Verhaltensweisen mit endlichen, berechenbaren Darstellungen ermöglicht. Unsere Methoden sind generisch für die jeweiligen Problembereiche und ermöglichen eine präzise Analyse weit über existierende Werke hinaus.

Für numerische Software präsentieren wir eine neue Theorie der Online-Zerlegung, die dynamisch teure Berechnungen zerlegt und dadurch die Komplexität ohne Präzisionsverlust reduziert. Unsere Theorie ist generisch und kann ohne Präzisionsverlust zur Zerlegung aller bestehender Subpolyederdomänen verwendet werden. Wir setzen datengesteuertes maschinelles Lernen ein, um die Leistung der numerischen Programmanalyse ohne signifikanten Präzisionsverlust weiter zu verbessern. Für neuronale Netze entwerfen wir eine neue Abstraktion für schnelle und skalierbare Analysen, die mit anwendungsspezifischen Approximationen von Nichtlinearitäten, welche üblicherweise in neuronalen Netzen verwendet werden, ausgestattet ist. Wir entwerfen auch eine neue konvexe Relaxationsmethodik, welche Relaxationen produziert, welche präziser sind als jene früherer Ansätze. Wir stellen eine neuartige Kombination unserer Abstraktions- und Relaxationsmethodik mit präzisen Solvern zur Verfügung und ermöglichen damit Zertifizierung nach dem neuesten Stand der Technik.

In dieser Arbeit stellen wir zwei neue, öffentlich verfügbare Softwaresysteme vor: ELINA und ERAN. ELINA bietet optimierte Implementierungen der populären

Polyeder-, Oktagon- und Zonendomäne, basierend auf unserer Theorie der Online-Zerlegung, die eine schnelle und präzise Analyse großer Linux-Gerätetreiber mit  $> 500$  Variablen in wenigen Sekunden ermöglicht. ERAN enthält unsere benutzerdefinierte Abstraktions- und konvexe Relaxationsmethodik, und eine Kombination von Relaxationen mit Solvern, um eine schnelle und präzise Analyse großer neuronaler Netzwerke mit Zehntausenden von Neuronen innerhalb weniger Sekunden zu ermöglichen. Beide Systeme wurden von Grund auf neu entwickelt, setzen den derzeitigen Stand der Technik für ihren jeweiligen Bereich und erzielen Ergebnisse, die ausserhalb der Reichweite konkurrierender Systeme liegen.

---

## PUBLICATIONS

---

This thesis is based on the following publications:

- **Gagandeep Singh**, Markus Püschel, Martin Vechev.  
*Fast Polyhedra Abstract Domain.*  
ACM Principles of Programming Languages (POPL), 2017. [190]
- **Gagandeep Singh**, Markus Püschel, Martin Vechev.  
*A Practical Construction for Decomposing Numerical Abstract Domains.*  
ACM Principles of Programming Languages (POPL), 2018. [191]
- **Gagandeep Singh**, Markus Püschel, Martin Vechev.  
*Fast Numerical Program Analysis with Reinforcement Learning.*  
Computer Aided Verification (CAV), 2018. [192]
- **Gagandeep Singh**, Timon Gehr, Markus Püschel, Martin Vechev.  
*An Abstract Domain for Certifying Neural Networks.*  
ACM Principles of Programming Languages (POPL), 2019. [188]
- **Gagandeep Singh**, Timon Gehr, Markus Püschel, Martin Vechev.  
*Boosting Robustness Certification of Neural Networks.*  
International Conference on Learning Representations (ICLR), 2019. [187]
- **Gagandeep Singh**, Rupanshu Ganvir, Markus Püschel, Martin Vechev.  
*Beyond the Single Neuron Convex Barrier for Neural Network Certification.*  
Neural Information Processing Systems (NeurIPS), 2019 [185]

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

- **Gagandeep Singh**, Markus Püschel, Martin Vechev.  
*Making Numerical Program Analysis Fast.*  
ACM Programming Language Design and Implementation (PLDI), 2015.  
[189]
- **Gagandeep Singh**, Timon Gehr, Matthew Mirman, Markus Püschel, Martin Vechev.  
*Fast and Effective Robustness Certification.*  
Neural Information Processing Systems (NeurIPS), 2018. [186]

- Mislav Balunovic, Maximilian Baader, **Gagandeep Singh**, Timon Gehr, Martin Vechev.  
*Certifying Geometric Robustness of Neural Networks*.  
Neural Information Processing Systems (NeurIPS), 2019. [15]
- Jingxuan He, **Gagandeep Singh**, Markus Püschel, Martin Vechev.  
*Learning Fast and Precise Numerical Analysis*.  
ACM Programming Language Design and Implementation (PLDI), 2020. [96]
- Raphaël Dang Nhu, **Gagandeep Singh**, Pavol Bielik, Martin Vechev.  
*Adversarial Attacks on Probabilistic Autoregressive Forecasting Models*.  
International Conference on Machine Learning (ICML), 2020. [62]

The following publications were part of my Ph.D. research and are available on Arxiv:

- Matthew Mirman, **Gagandeep Singh**, Martin Vechev.  
*A Provable Defense for Deep Residual Networks*.  
Arxiv, 2019. [146]
- Wonryong Ryou, Jiayu Chen, Mislav Balunovic, **Gagandeep Singh**, Andrei Dan, Martin Vechev.  
*Fast and Effective Robustness Certification for Recurrent Neural Networks*.  
Arxiv, 2020. [172]
- Christoph Müller, **Gagandeep Singh**, Markus Püschel, Martin Vechev.  
*Neural Network Robustness Verification on GPUs*.  
Arxiv, 2020. [152]
- Dimitar I. Dimitrov, **Gagandeep Singh**, Timon Gehr, Martin Vechev.  
*Scalable Inference of Symbolic Adversarial Examples*.  
Arxiv, 2020. [63]



---

## ACKNOWLEDGEMENTS

---

I would like to use this page to thank all those people that directly and indirectly supported me throughout my doctoral studies.

My gratitude goes first of all to my advisors Prof. Markus Püschel and Prof. Martin Vechev. Your valuable advice has shaped my perspective on research and life. I would also like to express my gratitude to the reviewers: Prof. Patrick Cousot and Prof. Clark Barrett, for providing constructive feedback on the thesis that I have incorporated in the final version. I am thankful to the ETH faculty that helped me at various points of my doctoral studies: Prof. Peter Müller, Prof. Zhendong Su, Prof. Ghaffari Mohsen, and Prof. Srdjan Capkun.

I would like to acknowledge the co-authors of papers published during my Ph.D. I really enjoyed working with all of you: Maximilian Baader, Mislav Balunovic, Pavol Bielik, Jiayu Chen, Andrei Dan, Raphaël Dang Nhu, Dimitar I. Dimitrov, Rupanshu Ganvir, Timon Gehr, Jingxuan He, Matthew Mirman, Christoph Müller, and Wonryong Ryou.

I would also like to thank many past and present colleagues in the software group at ETH Zurich, in particular Victoria Caparrós Cabezas, Makarchuk Gleb, Georg Ofenbeck, Joao Rivera, Bastian Seifert, Francois Serre, Tyler Smith, Daniele Spampinato, Alen Stojanov, Chris Wendler, Eliza Wszola, Luca Della Toffola, Afra Amini, Benjamin Bichsel, Rudiger Birkner, Dana Drachler Cohen, Dimitar K. Dimitrov, Marc Fischer, Inna Grijnevitch, Viktor Ivanov, Pesho Ivanov, Jonathan Mauer, Sasa Misailovic, Rumen Paletov, Momchil Peychev, Veselin Raychev, Anian Ruoss, Samuel Steffen, Petar Tsankov, Vytautas Astrauskas, Lucas Brutschy, Alexandra Bugariu, Fábio Pakk Selmi Dei, Jérôme Dohrau, Marco Eilers, Uri Juhasz, Gaurav Parthasarathy, Federico Poli, Alexander Summers, Caterina Urban, Arshavir Ter Gabrielyan, and Manuel Rigger for many insightful discussions about research and beyond.

I would also like to thank the administrative staff at ETH that helped me navigate through many of the Swiss rules related to immigration and beyond: Fiorella Meyer, Mirella Rutz, Sandra Schneider, and Marlies Weissert.

I am grateful to all my friends and flatmates in Zurich. Without you, my time in Zurich would have been quite boring. Special thanks to Kushagra Alankar and Jagannath Biswakarma, we started our journey in Zurich together in the same building. It has been great knowing you all these years and I will cherish our great memories of cooking, traveling, and of course, watching cricket together.

Finally, I would like to thank my parents and sister, without whom none of this would have been possible.



---

# CONTENTS

---

ABSTRACT	iii
ACKNOWLEDGMENTS	ix
<b>1 INTRODUCTION</b>	<b>1</b>
1.1 Abstract Interpretation . . . . .	6
1.2 Fast and Precise Numerical Program Analysis . . . . .	7
1.3 Fast and Precise Neural Network Certification . . . . .	11
<b>I Fast and Precise Numerical Program Analysis</b>	<b>19</b>
<b>2 FAST POLYHEDRA ANALYSIS VIA ONLINE DECOMPOSITION</b>	<b>21</b>
2.1 Background on Polyhedra Analysis . . . . .	22
2.1.1 Representation of Polyhedra . . . . .	23
2.1.2 Polyhedra Domain . . . . .	24
2.1.3 Polyhedra Domain Analysis: Example . . . . .	26
2.1.4 Transformers and Asymptotic Complexity . . . . .	27
2.2 Polyhedra Decomposition . . . . .	29
2.2.1 Partitions . . . . .	29
2.2.2 Transformers and Partitions . . . . .	30
2.3 Polyhedra Domain Analysis with Partitions . . . . .	35
2.3.1 Polyhedra Encoding . . . . .	35
2.3.2 Transformers and Permissible Partitions . . . . .	36
2.4 Polyhedra transformers . . . . .	38
2.4.1 Auxiliary Transformers . . . . .	38
2.4.2 Meet ( $\sqcap$ ) . . . . .	40
2.4.3 Inclusion ( $\sqsubseteq$ ) . . . . .	40
2.4.4 Conditional . . . . .	41
2.4.5 Assignment . . . . .	43
2.4.6 Widening ( $\nabla$ ) . . . . .	43
2.4.7 Join ( $\sqcup$ ) . . . . .	44
2.5 Experimental Evaluation . . . . .	50
2.5.1 Experimental Setup . . . . .	50
2.5.2 Experimental Results . . . . .	50
2.6 Discussion . . . . .	54
<b>3 GENERALIZING ONLINE DECOMPOSITION</b>	<b>55</b>
	xi

3.1	Generic Model for Numerical Abstract Domains . . . . .	56
3.2	Decomposing Abstract Elements . . . . .	60
3.3	Recipe for Decomposing Transformers . . . . .	61
3.4	Decomposing Domain Transformers . . . . .	63
3.4.1	Conditional . . . . .	64
3.4.2	Assignment . . . . .	67
3.4.3	Meet ( $\sqcap$ ) . . . . .	72
3.4.4	Join ( $\sqcup$ ) . . . . .	73
3.4.5	Widening ( $\nabla$ ) . . . . .	76
3.5	Experimental Evaluation . . . . .	78
3.5.1	Polyhedra . . . . .	79
3.5.2	Octagon . . . . .	81
3.5.3	Zone . . . . .	83
3.5.4	Summary . . . . .	85
3.6	Related Work . . . . .	85
3.7	Discussion . . . . .	88
4	REINFORCEMENT LEARNING FOR NUMERICAL DOMAINS	89
4.1	Reinforcement Learning for Static Analysis . . . . .	92
4.1.1	Reinforcement Learning . . . . .	92
4.1.2	Instantiation of RL to Static Analysis . . . . .	94
4.2	Polyhedra Analysis and Approximate Transformers . . . . .	95
4.2.1	Block Splitting . . . . .	95
4.2.2	Merging of Blocks . . . . .	98
4.2.3	Approximation for Polyhedra Analysis . . . . .	98
4.3	Reinforcement Learning for Polyhedra Analysis . . . . .	100
4.4	Experimental Evaluation . . . . .	103
4.5	Related Work . . . . .	106
4.6	Discussion . . . . .	109
<b>II</b>	<b>Fast and Precise Neural Network Certification</b>	<b>111</b>
5	DEEPPOLY DOMAIN FOR CERTIFYING NEURAL NETWORKS	113
5.1	Overview . . . . .	117
5.1.1	Running example on a fully-connected network with ReLU activation . . . . .	117
5.2	Background: Neural Networks and Adversarial Regions . . . . .	123
5.3	Abstract Domain and Transformers . . . . .	125
5.3.1	ReLU Abstract Transformer . . . . .	125
5.3.2	Sigmoid and Tanh Abstract Transformers . . . . .	126
5.3.3	Maxpool Abstract Transformer . . . . .	126
5.3.4	Affine Abstract Transformer . . . . .	127

5.3.5	Neural Network Robustness Analysis . . . . .	127
5.3.6	Correctness of Abstract Transformers . . . . .	128
5.3.7	Soundness under Floating-Point Arithmetic . . . . .	133
5.4	Refinement of Analysis Results . . . . .	135
5.5	Experimental Evaluation . . . . .	137
5.5.1	Experimental setup . . . . .	138
5.5.2	$L_\infty$ -Norm Perturbation . . . . .	139
5.5.3	Rotation perturbation . . . . .	146
5.6	Discussion . . . . .	147
6	COMBINING ABSTRACTIONS WITH SOLVERS . . . . .	149
6.1	Overview . . . . .	152
6.2	Refinement with solvers . . . . .	154
6.3	k-ReLU relaxation framework . . . . .	159
6.3.1	Best convex relaxation . . . . .	160
6.3.2	1-ReLU . . . . .	160
6.3.3	k-ReLU relaxations . . . . .	161
6.4	Instantiating the k-ReLU framework . . . . .	162
6.4.1	Computing key parameters . . . . .	163
6.4.2	Certification and refinement with k-ReLU framework . . . . .	164
6.5	Evaluation . . . . .	164
6.5.1	Complete certification . . . . .	166
6.5.2	Incomplete certification . . . . .	167
6.6	Related work . . . . .	168
6.6.1	Neural Network Certification . . . . .	168
6.6.2	Constructing adversarial examples . . . . .	170
6.6.3	Adversarial training . . . . .	170
6.7	Discussion . . . . .	171
7	CONCLUSION AND FUTURE WORK . . . . .	173
7.1	Numerical Program analysis . . . . .	174
7.2	Neural network certification . . . . .	175
7.3	formal reasoning about cyber-physical systems . . . . .	176



---

## LIST OF FIGURES

---

Figure 1.1	The high-level idea behind abstract interpretation. . . . .	2
Figure 1.2	Number of problem instances certified by different certifiers at VNN-COMP'20. . . . .	4
Figure 1.3	An example of online decomposition. . . . .	8
Figure 1.4	Precision and cost of the Zone, Octagon and Polyhedra domain with and without ELINA. . . . .	9
Figure 1.5	Three Polyhedra analysis traces, the left-most and middle trace obtain precise results (the polyhedron at the bottom), however the analysis cost of the middle trace is lower. The right-most trace obtains imprecise result. . . . .	10
Figure 1.6	Neural network certification problem. . . . .	12
Figure 1.7	Different dimensions of the neural network certification problem. The text in green, blue, and black respectively represent cases included in this thesis, those that we consider in our work but not covered in this thesis, those that are not considered in our work. . . . .	14
Figure 1.8	ERAN certification framework. . . . .	16
Figure 2.1	Two representations of polyhedron defined over variables $x_1$ and $x_2$ . (a) Bounded polyhedron; (b) unbounded polyhedron. . . . .	24
Figure 2.2	Code with assertion for static analysis. . . . .	25
Figure 2.3	Polyhedra domain analysis (first iteration) on the example program on the left. The polyhedra are shown in constraint representation. . . . .	26
Figure 2.4	Two examples of $P \sqcup Q$ with $\pi_P = \pi_Q = \{\{x_1\}, \{x_2\}\}$ . (a) $P_1 \neq Q_1, P_2 \neq Q_2$ ; (b) $P_1 = Q_1, P_2 \neq Q_2$ . . . . .	33
Figure 2.5	Example of complexity reduction through decomposition for Polyhedra analysis on an example program. . . . .	37
Figure 2.6	Precision loss for static partitioning. . . . .	49
Figure 2.7	The join transformer during the analysis of the <code>usb_core_main0</code> benchmark. The x-axis shows the join number and the y-axis shows the number of variables in $\mathcal{N} = \bigcup_{\mathcal{A} \in \bar{\pi} \setminus \mathcal{U}} \mathcal{A}$ (subset of variables affected by the join) and in $\mathcal{X}$ . The first figure shows these values for all joins whereas the second figure shows it for one of the expensive regions of the analysis. . . . .	53
Figure 4.1	Policies for balancing precision and speed in static analysis. . . . .	89
Figure 4.2	Reinforcement learning for static analysis. . . . .	90

Figure 4.3	Graph $G$ for $P(\mathcal{X}_t)$ in Example 4.2.1 . . . . .	97
Figure 5.1	Two different attacks applied to MNIST images. . . . .	115
Figure 5.2	Example fully-connected neural network with ReLU activations. . . . .	117
Figure 5.3	The neural network from Fig. 5.2 transformed for analysis with the DeepPoly abstract domain. . . . .	118
Figure 5.4	Convex approximations for the ReLU function: (a) shows the convex approximation [69] with the minimum area in the input-output plane, (b) and (c) show the two convex approximations used in DeepPoly. In the figure, $\lambda = u_i / (u_i - l_i)$ and $\mu = -l_i \cdot u_i / (u_i - l_i)$ . . . . .	120
Figure 5.5	Certified robustness and average runtime for $L_\infty$ -norm perturbations by DeepPoly against $AI^2$ , Fast-Lin, and DeepZ on the MNIST <code>FFNNSmall</code> . DeepZ and Fast-Lin are equivalent in robustness. . . . .	140
Figure 5.6	Certified robustness and average runtime for $L_\infty$ -norm perturbations by DeepPoly and DeepZ on the MNIST <code>FFNNMed</code> and <code>FFNNBig</code> networks. . . . .	141
Figure 5.7	Average percentage of ReLU inputs that can take both positive and negative values for DeepPoly and DeepZ on the MNIST <code>FFNNSmall</code> and <code>FFNNMed</code> networks. . . . .	142
Figure 5.8	Certified robustness and average runtime for $L_\infty$ -norm perturbations by DeepPoly and DeepZ on the MNIST <code>FFNNSigmoid</code> and <code>FFNNTanh</code> networks. . . . .	142
Figure 5.9	Certified robustness and average runtime for $L_\infty$ -norm perturbations by DeepPoly and DeepZ on the MNIST <code>ConvSmall</code> networks. . . . .	143
Figure 5.10	Certified robustness and average runtime for $L_\infty$ -norm perturbations by DeepPoly and DeepZ on the CIFAR10 fully-connected networks. . . . .	144
Figure 5.11	Certified robustness and average runtime for $L_\infty$ -norm perturbations by DeepPoly and DeepZ on the CIFAR10 <code>ConvSmall</code> networks. . . . .	145



Figure 5.12	Results for robustness against rotations with the MNIST FFNN <sub>small</sub> network. Each row shows a different attempt to prove that the given image of the digit 3 can be perturbed within an $L_\infty$ ball of radius $\epsilon = 0.001$ and rotated by an arbitrary angle $\theta$ between $-45$ to $65$ degrees without changing its classification. For the last two attempts, we show 4 representative combined regions (out of 220, one per batch). The running time is split into two components: (i) the time used for interval analysis on the rotation algorithm and (ii), the time used to prove the neural network robust with all of the computed bounding boxes using DeepPoly. . . . .	146
Figure 6.1	The input space for the ReLU assignments $y_1 := \text{ReLU}(x_1)$ , $y_2 := \text{ReLU}(x_2)$ is shown on the left in blue. Shapes of the relaxations projected to 3D are shown on the right in red. . .	150
Figure 6.2	Certification of property $x_9 \leq 2$ . Refining DeepPoly with 1-ReLU fails to prove the property whereas 2-ReLU adds extra constraints (in green) that help in verifying the property. . .	153
Figure 6.3	DeepPoly relaxations for $x_i := \text{ReLU}(x_j)$ using the original bounds $l_j, u_j$ (in blue) and the refined bounds $l'_j, u'_j$ (in green) for $x_j$ . The refined relaxations have smaller area in the $x_i x_j$ -plane. . . . .	157
Figure 6.4	Steps to instantiating the k-ReLU framework. . . . .	163

---

## LIST OF TABLES

---

Table 1.1	Polyhedra analysis of Linux device drivers with ELINA vs. PPL. . . . .	3
Table 2.1	Asymptotic complexity of Polyhedra operators with different representations. . . . .	28
Table 2.2	Asymptotic time complexity of Polyhedra operators with decomposition. . . . .	36
Table 2.3	Speedup of Polyhedra domain analysis for ELINA over NewPolka and PPL. . . . .	51
Table 2.4	Partition statistics for Polyhedra analysis with ELINA. . . .	52
Table 3.1	Instantiation of constraints expressible in various numerical domains. . . . .	58
Table 3.2	Speedup for the Polyhedra analysis with our decomposition vs. PPL and ELINA. . . . .	80
Table 3.3	Partition statistics for the Polyhedra domain analysis. . . .	81
Table 3.4	Asymptotic time complexity of the Octagon transformers. . .	82

Table 3.5	Speedup for the Octagon domain analysis with our decomposition over the non-decomposed and the decomposed versions of ELINA. . . . .	82
Table 3.6	Partition statistics for the Octagon domain analysis. . . . .	83
Table 3.7	Speedup for the Zone domain analysis with our decomposition over the non-decomposed implementation. . . . .	84
Table 3.8	Partition statistics for the Zone domain analysis. . . . .	84
Table 4.1	Mapping of RL concepts to Static analysis concepts. . . . .	94
Table 4.2	Features for describing RL state $s$ ( $m \in \{1, 2\}, 0 \leq j \leq 8, 0 \leq h \leq 3$ ). . . . .	101
Table 4.3	Instantiation of Q-learning to Polyhedra domain analysis. . . . .	103
Table 4.4	Timings (seconds) and precision of approximations (%) w.r.t. ELINA. . . . .	105
Table 5.1	Neural network architectures used in our experiments. . . . .	139
Table 5.2	Certified robustness by DeepZ and DeepPoly on the large convolutional networks trained with DiffAI. . . . .	143
Table 6.1	Volume of the output bounding box from kPoly on the MNIST FFNNMed network. . . . .	151
Table 6.2	Neural network architectures and parameters used in our experiments. . . . .	165
Table 6.3	Number of certified adversarial regions and runtime of kPoly vs. DeepPoly and RefineZono. . . . .	166

---

## INTRODUCTION

---

Providing formal guarantees on the safety and reliability of modern computing systems is of fundamental importance in today’s digital society. However, ensuring this objective has turned out to be a very challenging research problem as modern systems continue to grow in scale, complexity, and diversity, which dictates the need for ever more efficient, advanced, and rigorous methods able to provide the required guarantees. In this thesis, we address this challenge with novel automated reasoning methods for two critical problem domains: programs and deep learning models. Our methods come with clean theoretical guarantees and scale the precise and fast analysis of realistic systems beyond the reach of existing methods, thereby advancing the state-of-the-art in automated formal reasoning.

Our methodology is based on the theory of abstract interpretation [55], an elegant mathematical framework for statically overapproximating (potentially infinite) concrete behaviors with a finite representation. Abstract interpretation has been successfully applied in many domains including the safety and correctness verification of critical avionics software [27], windows libraries [134], untrusted Linux kernels [80], hybrid systems [97], distributed networks [23], embedded systems [111], and neural networks [78]. Fig. 1.1 depicts the high-level idea behind abstract interpretation. Assume we want to compute a function  $f$  for a given set of inputs  $\phi$  and check if a safety property  $\psi$  holds, i.e., the set of outputs satisfies  $f(\phi) \subseteq \psi$ . In most cases, and for the problem instances considered in this work, computing the entire set  $f(\phi)$  exactly is impossible. Abstract interpretation instead provides a framework to compute an overapproximation  $g(\phi) \supseteq f(\phi)$ .  $g(\phi)$  can then be used as a proxy for proving the safety property on  $f(\phi)$  since  $g(\phi) \subseteq \psi$  implies  $f(\phi) \subseteq \psi$ .

There is an inherent precision/cost tradeoff in computing  $g(\phi)$ . An imprecise  $g(\phi)$ , as in Fig. 1.1 (a), is fast to compute, but may be insufficient for proving  $\psi$ . On the other hand, computing a precise  $g(\phi)$ , as in Fig. 1.1 (b), can be too expensive for large benchmarks. Thus the main challenge in the adoption of abstract interpretation for analyzing real-world systems lies in designing methods for computing  $g(\phi)$  that are both fast and precise. Further, such methods should be as generic as possible so that they can be applied to a variety of computing systems. This leads us to the main research question that we address in this thesis:

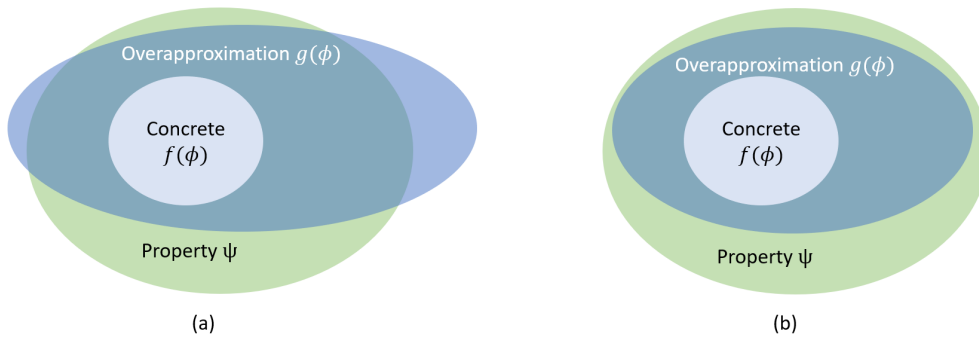


Figure 1.1: The high-level idea behind abstract interpretation.

*Are there generic methods for designing fast and precise automated analyzers for modern systems?*

**MAIN CONTRIBUTIONS** Our core contributions are at the intersection of programming language and machine learning research. We target numerical functions  $f$  and properties  $\psi$  with abstract interpretation in two critical and challenging domains: programs and deep learning models. Numerical abstract interpretation is an essential component of modern program analyzers [27, 80, 134, 156, 200, 201, 204, 209]. It is crucial, for example, for proving in programs the absence of critical bugs such as buffer overflow, division by zero, or non-termination. Our contributions are divided into two lines of work:

1. *Fast and precise numerical program analysis:* We develop general methods to make numerical program analysis significantly faster, often by orders of magnitude, without losing precision. This enables precise analysis of real-world programs beyond the reach of prior work.
2. *Fast and precise neural network certification:* We designed new automated methods for certifying large real-world neural networks. Our methods enable state-of-the-art certification beyond the reach of other existing methods.

We briefly expand on these contributions next.

**FAST AND PRECISE NUMERICAL PROGRAM ANALYSIS** We developed a theoretical framework for speeding up existing numerical program analyzers without any loss of precision. Our key idea is to dynamically decompose the computation of  $g(\phi)$  to reduce the asymptotic cost and enable orders of magnitude speedup in many practical cases. We implemented our theory in the form of a library for numerical program analysis, called ELINA (ETH Library for Numerical Analysis). ELINA was developed from scratch, has  $\approx 70K$  lines of code (LOC) in C, and is currently the state-of-the-art for numerical analysis.

Table 1.1 compares the runtime and memory consumption for the Polyhedra analysis of large real-world Linux device drivers with ELINA and state-of-the-art Parma Polyhedra Library (PPL) [12]. The second column shows the maximum

number of variables occurring in a polyhedron during the analysis. In the table, the entry T0 means that the analysis timed out after 4 hours while entry M0 represents memory consumption  $> 16$  GB. The last column of Table 1.1 shows the speedup of ELINA over PPL. We provide a lower bound on the speedup in the case of a timeout with PPL. The speedup when the Polyhedra analysis with PPL runs out of memory is  $\infty$  as it can never finish on the given machine. It can be seen that ELINA is significantly more memory and time-efficient than PPL. It finishes the analysis of 12 out of 13 benchmarks in a few seconds and never consumes more than 1 GB of memory. In contrast, PPL either times out or runs out of memory on 8 benchmarks. Currently, ELINA is used in several research projects in both academia and industry. Its Github repository at the time of this writing has 77 stars and 31 forks.

Table 1.1: Polyhedra analysis of Linux device drivers with ELINA vs. PPL.

Benchmark	n	PPL		ELINA		Speedup
		time(s)	memory(GB)	time(s)	memory(GB)	
firewire_firedtv	159	331	0.9	0.2	0.2	1527
net_fddi_skfp	589	6142	7.2	4.4	0.3	1386
mtd_ubi	528	M0	M0	1.9	0.3	$\infty$
usb_core_main0	365	4003	1.4	29	0.7	136
tty_synclinkmp	332	M0	M0	2.5	0.1	$\infty$
scsi_advansys	282	T0	T0	3.4	0.2	$>4183$
staging_vt6656	675	T0	T0	0.5	0.1	$>28800$
net_ppp	218	10530	0.1	891	0.1	11.8
p10_l00	303	121	0.9	5.4	0.2	22.4
p16_l40	874	M0	M0	2.9	0.4	$\infty$
p12_l57	921	M0	M0	6.5	0.3	$\infty$
p13_l53	1631	M0	M0	25	0.9	$\infty$
p19_l59	1272	M0	M0	12	0.6	$\infty$

In a second step, we leveraged machine learning for further speeding up numerical reasoning in ELINA. Our key insight here is that there are redundant computations during the intermediate steps for computing  $g(\phi)$ . To remove this redundancy, we established a new connection between reinforcement learning and program analysis by instantiating the concepts of reinforcement learning for numerical program analysis and learned adaptive heuristics for further speeding up ELINA. Our results show further speedups over ELINA of up to 1 or 2 orders of magnitude with little loss in precision.

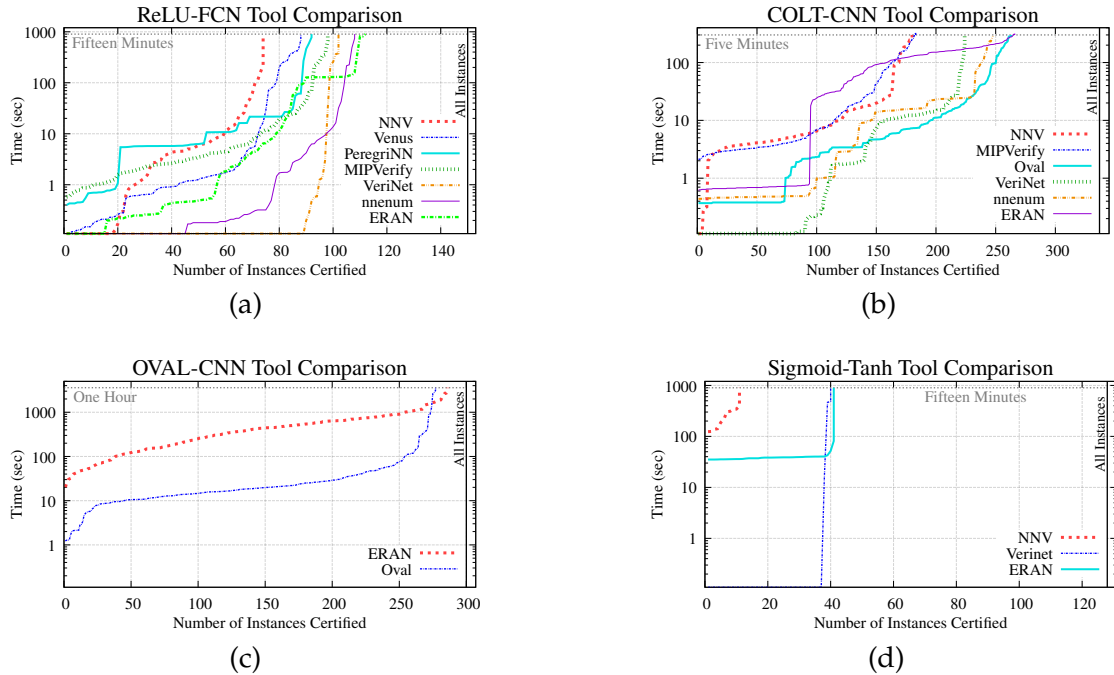


Figure 1.2: Number of problem instances certified by different certifiers at VNN-COMP’20.

**FAST AND PRECISE NEURAL NETWORK CERTIFICATION** In a second, independent line of work, we applied numerical abstract interpretation to certify the safety and robustness of deep neural networks. Our key idea is to design custom methods for precisely approximating the non-linearities such as ReLU and sigmoid used in modern neural networks. We designed numerical reasoning methods for the precise and scalable certification of large neural networks containing tens of thousands of neurons within a few minutes. Our methods are implemented in the form of a certifier called ERAN (ETH Robustness Analyzer for Neural Networks). ERAN is currently the state-of-the-art for neural network certification and can certify properties beyond the reach of existing certifiers. It recently verified the highest number of benchmarks in the neural network verification competition (VNN-COMP’20) held as part of CAV’20.

Fig. 1.2 shows the number of robustness based specifications certified by the different competing state-of-the-art certifiers in the 4 (out of 5) most challenging categories of the VNN-COMP’20 competition within a specified time limit. The plots are taken from the competition report publicly available at <https://sites.google.com/view/vnn20/vnncomp>. Fig. 1.2 (a) compares the certification results on 3 ReLU-based MNIST fully-connected networks with 7 different certifiers participating in this category. The number of problem instances in this category was 150; the time limit was set to 15 minutes per instance. The largest network had 6 layers containing 256 neurons each. It can be seen that ERAN (in green) certified the most certifying 112/150 instances.

Fig. 1.2 (b) compares the results on 4 MNIST and CIFAR<sub>10</sub> ReLU-based convolutional networks with 6 competing certifiers. These networks were trained by COLT

[148] and currently achieve state-of-the-art accuracy and provable robustness for their respective datasets. The number of problem instances in this category was 337; the time limit was set to 5 minutes per instance. The largest network in this category had  $\approx 50\text{K}$  neurons. Out of the 6, ERAN (in purple) certified the highest number of instances certifying 266/337 instances.

Fig. 1.2 (c) compares the results on 3 CIFAR<sub>10</sub> ReLU-based convolutional networks trained by [135]. The largest network had  $> 6\text{K}$  neurons. The specifications for these networks are harder; therefore, the time limit was increased to 1 hour per instance. Only 2 certifiers competed in this category with ERAN (in red) certifying 286/300 instances within the time limit which is 9 more than the competition.

Fig. 1.2 (d) compares the number of instances certified on the MNIST Sigmoid and Tanh based fully-connected networks. The number of instances in this category was 128; the time limit was set to 15 minutes per instance. The largest network had 3 layers and 350 neurons. Only 3 competing certifiers supported networks with Sigmoid and Tanh activations. It can be seen that ERAN (in sky blue) certified the highest number of problem instances.

On the remaining category, ERAN verified the second-highest number of benchmarks. We note that ERAN is the only verifier that competed in all the categories demonstrating its ability to handle diverse networks and specifications in a precise and scalable manner. Based on its demonstrated performance and flexibility, ERAN is already widely used in several research projects in both academia and industry. Its repository on Github at the time of this writing has 145 stars and 49 forks.

**MAIN CONTRIBUTIONS: IMPACT.** In summary, this thesis provides state-of-the-art methods for numerical program analysis and neural network certification. Our methods scale to problem sizes not possible with prior work and are also precise. We note that our approach for numerical program analysis differs significantly from that for neural network certification. This is because the occurring functions  $f$  are different in these domains. In programs,  $f$  is unbounded but often decomposable while  $f$  for neural networks is bounded but non-decomposable and highly non-linear. Further, programs typically have loops, and thus their analysis may iterate many times before reaching a fixpoint. In contrast, neural networks do not have loops but contain orders of magnitude more variables than programs. As a result, our novel and generic methods for program analysis are not suited for neural networks and vice-versa. Both ELINA and ERAN have a significant number of users and thus our work already has real-world impact. Thus, in summary, we believe that this thesis expands the limits of formal methods for ensuring the safety and reliability of real-world systems.

Our work opens up a number of promising research directions such as applying decomposition beyond numerical reasoning for programs (e.g., heaps), using machine learning techniques for improving the speed of formal methods in other domains (e.g., operating systems), applying automated formal reasoning of machine learning models to more general specifications (e.g., probabilistic) and models (e.g.,

generative), and formal reasoning of cyber-physical systems with machine learning components (e.g., self-driving cars, robots). We provide more details in Chapter 7.

Next, we provide a brief informal overview of the abstract interpretation framework to discuss our contributions in greater detail.

## 1.1 ABSTRACT INTERPRETATION

Abstract interpretation is a theory for computing overapproximations ( $g(\phi)$  in Fig. 1.1) of (potentially infinite) concrete sets ( $f(\phi)$  in Fig. 1.1) through static analysis without actually running the system. The key concept in abstract interpretation is that of an *abstract domain*. An abstract domain has two main components: abstract elements and abstract transformers. Abstract elements represent an overapproximation of the potentially infinite and uncomputable concrete sets. The abstract transformers operating over abstract elements overapproximate the effect of applying transformations in the computing system on the corresponding concrete sets.

**Example 1.1.1.** Consider a concrete set  $\mathcal{C}$  containing only the integers that are both positive and even. An abstraction of  $\mathcal{C}$  in the Sign domain will represent it finitely using its sign, i.e.,  $+$ . A transformation on  $\mathcal{C}$  which multiplies a negative integer with  $\mathcal{C}$  can be overapproximated by an abstract transformer that operates on  $+$  and returns  $-$  approximating the concrete output containing even and negative integers.

In this thesis, we focus on numerical abstract interpretation, i.e., both  $f(\phi)$  and its approximations  $g(\phi)$  in Fig. 1.1 computed via an abstract domain are numerical functions. There is a tradeoff between the expressivity of a numerical domain and its cost. The Polyhedra domain [57] is the most expressive linear relational domain as it can capture all linear constraints between variables, i.e., constraints of the form  $\sum_i a_i \cdot x_i \leq b$  where  $a_i, b \in \mathbb{Z}$  and  $x_i$  is a variable. In an ideal setting, one would simply use the Polyhedra domain for numerical analysis. However, it has a worst-case exponential cost in time and space. Thus, an analyzer using Polyhedra can easily fail to analyze large programs by running out of memory or by timing out. Because of this, the Polyhedra domain has been often thought to be impractical [51, 122, 176]. On the other hand, the Interval domain [54] has a linear cost but is very imprecise as it captures only constraints of the form  $l_i \leq x_i \leq u_i$  where  $l_i, u_i \in \mathbb{Z}$ , thus ignoring relationships between variables. To balance the expressivity-cost tradeoff, researchers have designed domains that limit a domain's expressivity in exchange for better asymptotic complexity. Examples include Octagon [142], Zone [140], Pentagon [134], SubPolyhedra [122] and Gauge [205].



## 1.2 FAST AND PRECISE NUMERICAL PROGRAM ANALYSIS

The goal of numerical program analysis is computing the set  $f(\phi)$  of numerical values that the program variables can take during all program executions starting from an initial program state  $\phi$ . Here,  $\phi$  encodes the set of initial variable values. The problem of exactly computing  $f(\phi)$  is in general undecidable for programs due to Rice's Theorem [168]. Numerical domains allow obtaining a numerical over-approximation  $g(\phi)$  of  $f(\phi)$ . The design of these domains for program analysis remains an art requiring considerable and rare expertise. The various available domains tend to work well for the particular applications for which they were designed but may not be as effective on others. For example, the Octagon domain with variable packing is fast and precise for analyzing avionics software [27], but it is not as effective for windows libraries. Similarly, the Pentagon domain [134] used for effectively analyzing Windows libraries is imprecise for analyzing avionics software. The question then is if there are general methods for improving the speed of existing domains without sacrificing precision.

The starting point of our work is in identifying the considerable redundancy in numerical program analysis: unnecessary computations that are performed without affecting the final results. Removing this redundancy reduces the cost of analysis and improves speed without sacrificing precision. Numerical analysis has two types of redundancies:

- Single-step redundancy: At each step of the analysis, redundant computations are performed which do not affect the output of that step.
- Redundancy across analysis steps: The results of expensive computations are sometimes not needed to compute the final results as they are discarded later in the analysis.

To remove the single-step redundancy, we next present our theoretical framework of online decomposition.

**KEY IDEA: ONLINE DECOMPOSITION.** Online decomposition is based on the observation that the set of program variables can be partitioned into subsets with respect to the abstract elements computed during the analysis such that linear constraints exist only between the variables in the same subset. This partitioning thus decomposes the corresponding abstract elements into smaller elements. The transformers then operate over these smaller elements, which reduces their asymptotic complexity without losing precision.

**Example 1.2.1.** Fig. 1.3 shows an example of online decomposition where a polyhedron  $P$  is defined over 6 variables  $x_1 - x_6$ . The variable set can be decomposed into a partition  $\pi_P$  with three subsets  $\{\{x_1, x_2, x_3\}, \{x_4, x_5\}, \{x_6\}\}$ . Using  $\pi_P$ ,  $P$  can be decomposed into three pieces without losing any precision. The assignment transformer for  $x_2 := 2x_4$  then only needs to be applied on the parts that it affects, in this

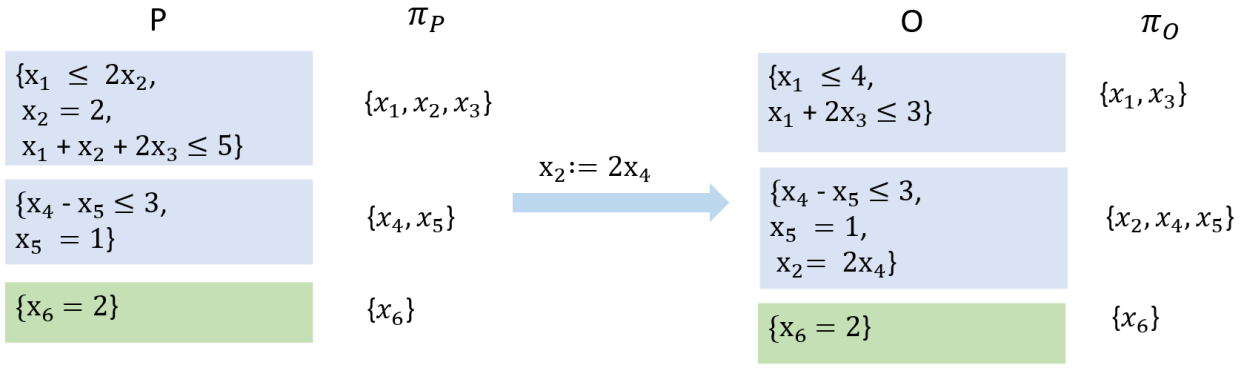


Figure 1.3: An example of online decomposition.

case, the first two, leaving the third part unchanged. This reduces complexity and improves performance.

The key challenge is in maintaining the partition dynamically as different transformers modify the partitioning in non-trivial ways during the analysis. Instead statically fixing the partition ahead of the analysis significantly loses precision as the modifications are based on the computed abstract elements. For example, the partition  $\pi_O$  of the output  $O$  in Fig. 1.3 depends on  $\pi_P$  which is dynamically computed from  $P$ . Thus, the main challenge is in maintaining the partitions efficiently and dynamically during analysis without losing precision. Prior work from [93, 94] applied online decomposition for speeding up the Polyhedra domain. However, the computed partitions were too coarse limiting the resulting speed up. Moreover, their approach is not general and does not work for decomposing other existing domains.

Our main contribution here is the design of mathematically clean, computationally efficient, and general algorithms for computing partitions of the output for all transformers (usually 40 or more) without precision loss. Our algorithms can efficiently decompose all existing subpolyhedra numerical domains thus enabling fast and precise analysis of large real-world programs beyond the reach of existing approaches.

**ONLINE DECOMPOSITION FOR SUBPOLYHEDRA DOMAINS.** Our work [190] is the first to show the applicability of the exponentially expensive Polyhedra abstraction for analyzing large real-world benchmarks. Prior work on applying online decomposition for improving Polyhedra performance [93, 94] was based on syntactic information from the input constraints and produced coarse partitions limiting its effectiveness for analyzing large programs. We designed algorithms for producing finer partitions based on semantic information which drastically reduces the observed complexity thus often obtaining two, three, or more orders of magnitude speedup over existing state-of-the-art approaches. We describe this work in detail in Chapter 3.

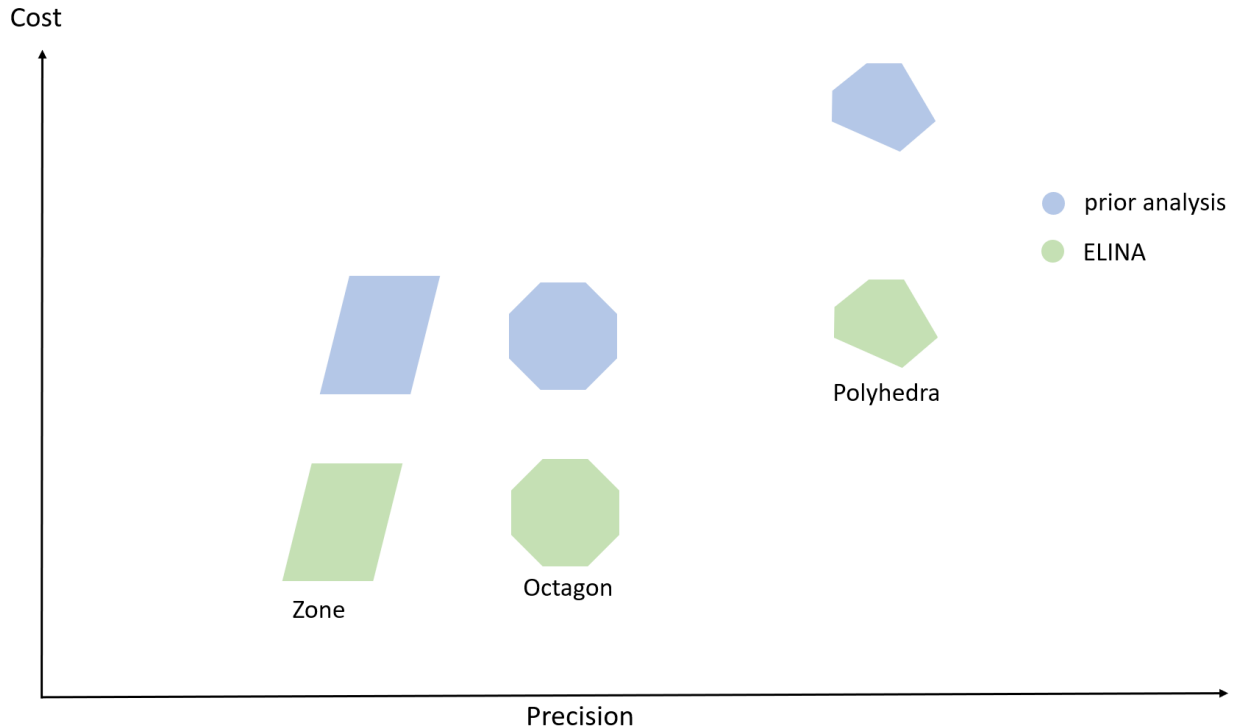


Figure 1.4: Precision and cost of the Zone, Octagon and Polyhedra domain with and without ELINA.

In our following work [191], we generalized the applicability of online decomposition from the Polyhedra to all subpolyhedra domains. We provided theory to refactor existing implementations of these domains to support decomposition. Our theory gives a general construction for obtaining decomposed transformers from original non-decomposed transformers for any subpolyhedra abstraction. We identified theoretical conditions where the decomposed analysis has the same precision as the original non-decomposed analysis. We then showed that these conditions are met by existing abstract domains. This allows for improving the performance of all existing domains without sacrificing any precision. Our construction is described in detail in Chapter 3. Follow up work [56] from Cousot et al. showed that online decomposition can be generalized for decomposing any relational abstraction, not only numerical.

**ELINA LIBRARY.** We have implemented all of the methods in the publicly available library ELINA available at <https://github.com/eth-sri/ELINA>. ELINA is the current state-of-the-art for numerical domains containing complete end-to-end implementations of the popular and expensive Zone, Octagon, and Polyhedra domains. Besides online decomposition, ELINA also contains domain-specific algorithmic improvements and performance optimizations including for cache locality, vectorization, and more. Fig. 1.4 shows the cost and precision of the Zone, Octagon, and Polyhedra domain analysis with and without ELINA. Both Zone and Octagon domains have asymptotic worst-case cubic cost, while that of Polyhedra is expo-

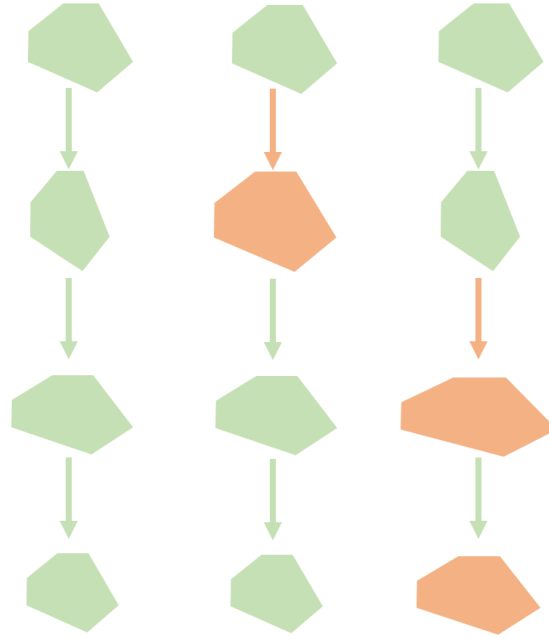


Figure 1.5: Three Polyhedra analysis traces, the left-most and middle trace obtain precise results (the polyhedron at the bottom), however the analysis cost of the middle trace is lower. The right-most trace obtains imprecise result.

nential. It can be seen in Fig. 1.4 that ELINA reduces the cost of domain analysis significantly without affecting precision.

ELINA enables the analysis of large real-world software, for example, Linux device drivers, containing thousands of lines of code and  $>500$  variables with the exponentially expensive Polyhedra in few seconds. Before our work, Polyhedra analysis was considered practically infeasible for such benchmarks as the resulting analysis did not finish even after several hours. For the Octagon analysis, ELINA gave typical speedups up to 32x faster than prior state-of-the-art libraries. ELINA is currently used for numerical analysis in several projects, including the popular Seahorn verification framework [91].

Online decomposition is effective at removing redundant computations at each analysis step without losing precision. Next we discuss our approach for removing redundancies across different analysis steps.

**MACHINE LEARNING FOR PROGRAM ANALYSIS.** Our next key observation for speeding up numerical program analysis is that the results of all abstract transformers applied for obtaining the final result in a sequence need not be precise, i.e., it is possible to apply imprecise intermediate transformers and still obtain a precise result. This points to redundancy in the abstract sequences. This redundancy occurs because some of the precise intermediate results are discarded later in the analysis. For example, an assignment transformer may remove all previous constraints involving the assigned variable.

**Example 1.2.2.** Fig. 1.5 shows three Polyhedra analysis traces for an overview of our approach. The nodes in the traces are the Polyhedra, and the edges represent Polyhedra transformers. The green and orange nodes respectively denote precise and imprecise Polyhedra. Similarly, the green and orange edges respectively denote precise but expensive and imprecise but fast transformers. In the left-most trace, the precise transformer is applied at each step to obtain a precise final result. In the middle trace, an approximate transformer is applied at the first node, however, this does not affect the final result computed faster. The choice of the node for the approximate transformer is crucial, as the final result in the right-most trace after applying the approximate transformer at the second node is imprecise.

Our key idea for removing redundancy across sequences is learning policies via reinforcement learning for selectively losing precision at different analysis steps such that the performance improves while the precision loss is as little as possible. We take this approach as in practice, hand-crafted or fixed policies often yield suboptimal results as the resulting analysis is either too slow or too imprecise. This is because policies maximizing precision and minimizing cost need to make adaptive decisions based on high-dimensional abstract elements computed dynamically during the analysis. Further, the sequence of transformers is usually quite long in practice. Using our approach, we showed for the first time that reinforcement learning can benefit static analysis in [192]. We created approximate transformers for the Polyhedra domain that enforce different degrees of finer partitions by explicitly removing constraints yielding several approximate transformers with different speeds and precision. Reinforcement learning then obtains a policy that selects among different transformers based on the abstract elements. Our overall approach is presented in detail in Chapter 4.

This approach is helpful when the analysis is inherently non-decomposable or the partitions with online decomposition become too coarse causing slowdowns. Our results show that analysis performance improves significantly with up to 550x speedup without significant precision loss enabling precise Polyhedra analysis of large programs not possible otherwise. In our follow-up work [96] (not covered in this thesis), we improve upon this approach by leveraging structured prediction and also show that our concept of using machine learning for speeding up static analysis is more general by applying it also to speed up the Octagon domain by up to 28x over ELINA without losing significant precision.

### 1.3 FAST AND PRECISE NEURAL NETWORK CERTIFICATION

Neural networks are increasingly deployed for decision making in many real-world applications such as self-driving cars [30], medical diagnosis [6], and finance [74]. However, recent research [86] has shown that neural networks are susceptible to undesired behavior in many real-world scenarios posing a threat to their reliability. Thus there is a growing interest in ensuring they behave in a provably reliable man-

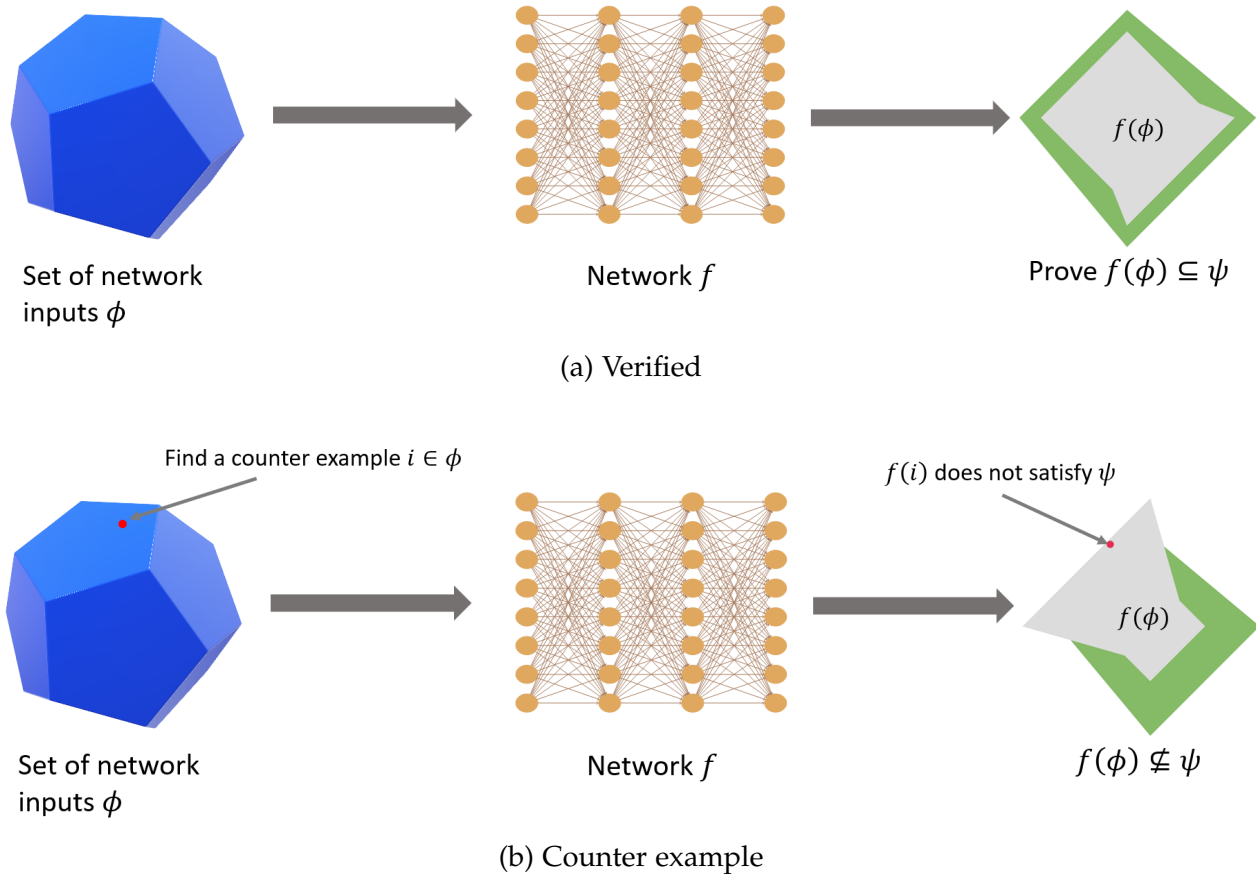


Figure 1.6: Neural network certification problem.

ner. To address this challenge, we designed scalable and precise methods based on numerical abstract interpretation for certifying the safety of deep neural networks.

**PROBLEM STATEMENT.** Fig. 1.6 shows the problem setting for neural network certification. We are given a set  $\phi$  of inputs to an already trained neural network  $f$  and a property  $\psi$  over the network outputs. Our goal is to prove whether  $f(\phi) \subseteq \psi$  holds, as in Fig. 1.6 (a) or produce an input  $i \in \phi$  for which the property is violated, as in Fig. 1.6 (b).  $\phi$  and  $\psi$  are usually chosen by a domain expert in such a way that a counter-example represents an undesired behavior. An example of  $\phi$  is the popular  $L_\infty$ -norm based region [40] for images.  $\phi$  here is constructed by considering all images that can be obtained by perturbing the intensity of each pixel in a correctly classified image by an amount of  $\epsilon \in \mathbb{R}$  independently.  $\psi$ , in this case, is usually classification robustness: all images in  $\phi$  should be classified correctly.

**DIMENSIONS OF THE PROBLEM.** Neural network certification is an emerging inter-disciplinary research direction that has been growing in the last three years. The problem space is rich and has many dimensions as shown in Fig. 1.7. We note that the dimensions in the figure are not exhaustive and represent only a subset.

For each dimension, the text in green represents cases that we handle in this thesis, those in blue represent cases handled by our work but not covered in this thesis. We do not handle the remaining cases currently.

The first dimension that we consider is that of the application domain. The computer vision domain is currently the most popular for neural network certification, but there is also growing interest in certifying models for natural language processing (NLP), speech, and aviation. To the best of our knowledge, there is no existing work on certifying neural networks for the remaining domains in Fig. 1.7.

The second dimension is of the specification being certified and has two components: the set of inputs  $\phi$  and the property over network outputs  $\psi$ . Our work focusses on  $\phi$  defined by changes to pixel or sound intensity, geometric transformations including rotation, translation, scaling on images, and changes to the sensor values. For a given  $\phi$ , the property  $\psi$  that we consider can be classification robustness, safety, or stability. We defined robustness above. Safety means that the network outputs do not satisfy a given error condition while stability signifies that the outputs are bounded within a given threshold. Both the error condition and the threshold are provided by a domain expert.

The particular architecture of the considered neural network is the third dimension. Our work considers fully-connected (FCN), convolutional (CNN), residual and recurrent neural networks (RNN) architectures. For such architectures, the non-linear functions in the hidden layers that we handle are ReLU, Sigmoid, Tanh, and Maxpool.

There are a variety of methods used in the literature for neural network certification such as SMT solvers [37, 69, 113, 114], mixed-integer linear programming [8, 32, 36, 49, 66, 135, 197], Lipschitz optimization [170], duality [67, 212], convex relaxations [7, 31, 68, 78, 127, 163, 175, 186, 188, 199, 211, 221], and combination of relaxations with solvers [187, 206]. Our work is based on custom convex relaxations for neural networks defined under the framework of abstract interpretation and the combination of such relaxations with MILP solvers.

The final dimension is of the type of formal guarantees. We provide deterministic guarantees meaning that we prove whether the property  $\psi$  holds for the entire set  $\phi$  or not. We note that there are models such as those in probabilistic forecasting [62] for which the outputs are probability distributions and probabilistic guarantees are a natural fit for these.

**ABSTRACT INTERPRETATION FOR NEURAL NETWORK CERTIFICATION** Unfortunately, complete certification as shown in Fig. 1.6 is infeasible for larger networks because of the large number of neurons and non-linear operations. For example, both the SMT and MILP solvers need to consider two branches per ReLU, which creates an exponential blowup for hundreds or thousands of ReLUs as is common. Thus, we focus on incomplete certification. As described later, we also combine both incomplete and complete certification for achieving state-of-the-art complete certification.

Applications	Vision Speech	NLP Code models		Finance Health care	Robotics Gaming	Aviation Manufacturing	
$\Phi$ / $\Psi$	Intensity Robustness	Geometric Safety	Patches Stability		Sensor values Fairness	Word substitution Temporal	
Architectures	FCN ReLU	CNN Sigmoid	Residual Tanh	RNN Maxpool	Transformers ELU	VAE Softplus	GAN DQN Softsign
Methods	MILP	SMT	Lipschitz constant	Duality	Convex relaxations	Solvers + Relaxations	
Guarantees	Deterministic			Probabilistic			

Figure 1.7: Different dimensions of the neural network certification problem. The text in green, blue, and black respectively represent cases included in this thesis, those that we consider in our work but not covered in this thesis, those that are not considered in our work.

As in Fig. 1.1, we overapproximate  $f(\phi)$  with  $g(\phi)$  using abstract interpretation. Because of the overapproximation, we cannot provide a counterexample as in Fig. 1.6 (b), i.e., if we fail to prove that  $f(\phi) \subseteq \psi$ , the status of the problem is unknown. It can be that either the property does not hold or  $g(\phi)$  is too imprecise. The first work to use abstract interpretation for neural network certification was that of Gehr et al. [78]. However, they used standard numerical domains used in program analysis, which are not well suited for neural networks. For example, even though the Polyhedra domain becomes practically feasible for program analysis via our work on online decomposition, it remains infeasible for neural network certification as the transformations in the networks create constraints between all neurons. As a result, the resulting instantiation of abstract interpretation for neural network certification is either too imprecise or it does not scale to larger networks.

**KEY IDEA: SPECIALIZED DOMAINS FOR NEURAL NETWORKS.** The main challenge in precisely certifying large neural networks is the fast and precise handling of the non-linearities employed in the networks. The approximations of these non-linearities in the implementations of the numerical domains commonly used in program analysis are either too imprecise or too expensive. For example, in the setting considered in this work, the input to a ReLU is always bounded while the ReLU approximations employed in program analysis assume unbounded input. Therefore to enable fast and precise certification of large neural networks, we designed relaxations tailored for exploiting the setting of neural network certification.

**CUSTOM ZONOTOPE RELAXATIONS.** In our first work in this direction [186], not included in this thesis, we designed new parallelizable Zonotope [81] relaxations tailored for handling the commonly used ReLU, Sigmoid, and Tanh non-linearities



in neural networks and provided theoretical guarantees on their optimality. Importantly, these approximations were sound with respect to floating-point arithmetic. This means that they always contain all results possible under different rounding modes and different orders of computations of floating-point operations. The resulting CPU-based analysis verified the robustness of a large image classification network with  $> 88\text{K}$  neurons against challenging  $L_\infty$ -norm based intensity perturbations in about 2 minutes.

**DEEPPOLY NUMERICAL DOMAIN.** Our next work [188] is a main contribution of this thesis. We designed a specialized numerical domain called DeepPoly with custom parallelizable transformers for handling the ReLU, Sigmoid, Tanh, and Max-pool non-linearities. We also provided theoretical guarantees on their optimality and soundness with respect to floating-point arithmetic. Further, we presented the first method for certifying the robustness of neural networks against geometric perturbations such as rotations. The resulting analysis yields more precise and scalable results than prior work. The DeepPoly domain is covered in detail in Chapter 5. In ongoing work with a master student, a custom GPU implementation GPUPoly with specialized GPU algorithms of DeepPoly has also been developed which can precisely certify a residual network with 34 layers and about 1M neurons against challenging  $L_\infty$ -norm based intensity perturbations in approximately 80 seconds.

**COMBINING RELAXATIONS WITH SOLVERS.** Building on DeepPoly, we designed a new approach in [187] combining the strengths of both approximation-based methods and precise solvers. Our key observation here is that as the analysis moves deeper into the network, the approximation error accumulates with each layer causing too much precision loss. We recover precision by calling the solver to obtain precise results and use those for refining our approximate analysis [186, 188]. To improve the scalability of the solver, we provide it with the bounds computed by our relaxations. Overall, this improves the precision of incomplete verification while maintaining sufficient scalability.

We also supply our approximations for refining the problem encoding of the solver for complete certification. Our approximations reduce the search area for the solver, e.g., by determining that certain branches are infeasible, thus improving its speed. For example, the complete certification of an ACAS Xu benchmarks [110] finished in 10 seconds with our approach, which is about 8x faster than the previous best.

The refinement method in [187] relies on the MILP-based exact encoding of the ReLU and does not scale for refining deeper layers in the networks. Refinement with the existing best convex relaxation of ReLU [175] scales but does not improve precision. In our most recent work [185], we designed a generic k-ReLU framework for obtaining more precise convex relaxations of the ReLU than possible with prior work. The generated relaxations are more scalable than the MILP-based encoding of ReLU, enabling more effective refinement.

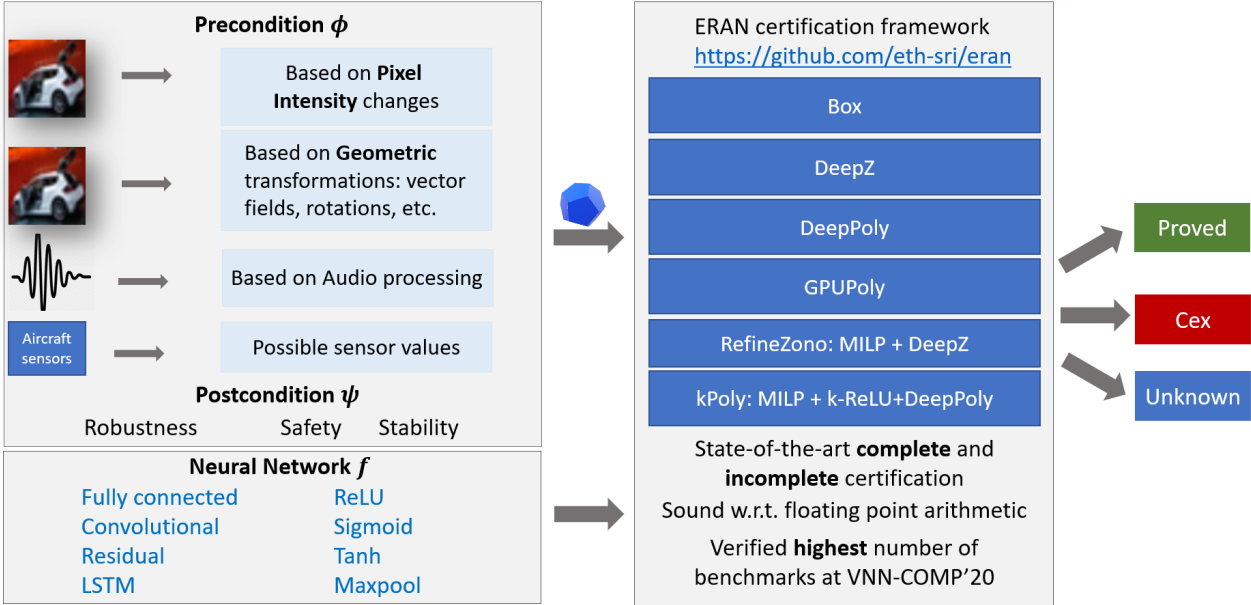


Figure 1.8: ERAN certification framework.

k-ReLU generates relaxations by considering multiple ReLUs jointly, a technique overlooked by all prior works, thus resulting in relaxations that are more precise than the prior single neuron based relaxations [175]. We also provide theoretical guarantees for the optimality of the k-ReLU relaxations. The cost and precision of the k-ReLU framework are tunable and the technique can be combined for improving the precision of all existing methods. Refining the results of DeepPoly with MILP and k-ReLU relaxation yields the most precise and scalable incomplete certification results reported to date. We describe this approach in greater detail in Chapter 6.

**ERAN CERTIFICATION FRAMEWORK.** All of the above-mentioned methods for neural network certification are implemented in the ERAN certification framework, publicly available at <https://github.com/eth-sri/eran>. Fig. 1.8 depicts its architecture. ERAN is capable of analyzing a wide range of neural network architectures (fully-connected, convolutional, residual, RNN), non-linearities (ReLU, Sigmoid, Tanh, Maxpool), datasets (image classification, audio classification, flight sensor data, drone speed), and  $\phi, \psi$  specified as polyhedral constraints over inputs and outputs respectively. We note that the input  $\phi$  capturing geometric transformations on images or after the transformations in the audio pre-processing pipeline are not polyhedral. A pre-analysis [15, 172, 188] is employed for computing a convex relaxation of such non-polyhedral input sets.

ERAN supports both complete and incomplete verification. ERAN provides a number of certification methods with varying degrees of precision and performance for incomplete certification. DeepPoly and DeepZ provide the most scalable and precise certification on a CPU. Certification with the following GPU-based extension of DeepPoly called GPUPoly yields the most scalable and precise results

overall. DeepPoly, DeepZ, and GPUPoly are sound with respect to floating-point arithmetic which is essential since otherwise the certification results may be wrong [107]. Both RefineZono and kPoly can be used to refine the results of DeepPoly, DeepZ, and GPUPoly by incurring an extra cost.

For each certification instance, analysis with ERAN yields one of three outcomes: (a) proves that the specification holds, (b) returns a counterexample (cex) when it can determine that a point in the precondition  $\phi$  violates the postcondition  $\psi$ , (c) otherwise the certification status is unknown which can happen when running incomplete certification or running complete certification with a time limit.

ERAN can be easily extended for other certification tasks and is currently the state-of-the-art tool for both complete and incomplete certification of neural networks. As a result, ERAN is widely used in many research projects [90, 133, 180]. Beyond certification, our methods have also been used to train state-of-the-art robust neural networks [148].



---

# PART I

---

## FAST AND PRECISE NUMERICAL PROGRAM ANALYSIS



---

## FAST POLYHEDRA ANALYSIS VIA ONLINE DECOMPOSITION

---

We start the thesis with our contributions for fast and precise numerical program analysis. In this chapter, we formally describe our theoretical framework and new algorithms for making the exponentially expensive Polyhedra domain practical for analyzing large real-world programs. We generalize the applicability of our methods to all subpolyhedra domains in Chapter 3 and provide theoretical guarantees on the analysis precision. Finally, in Chapter 4, we leverage reinforcement learning to further improve the performance of numerical program analysis. Overall, our methods yield many orders of magnitude speedup over prior approaches.

For almost 40 years, program analysis with the Polyhedra domain has been considered impractical for large real-world programs due to its exponential complexity [51, 122, 176]. In this chapter, we challenge this assumption and present a new approach that enables the application of Polyhedra for analyzing large, realistic programs, with speedups ranging between two to five orders of magnitude compared to the state-of-the-art. This allows us to analyze large real-world programs such as Linux device drivers beyond the reach of existing approaches within a few seconds. We note that our approach does not lose precision, i.e., it computes the same invariants as the original analysis.

The work in this chapter was published in [190].

**KEY IDEA: ONLINE DECOMPOSITION** Our key insight is based on the observation that the set of program variables can be partitioned with respect to the Polyhedra generated during the analysis into subsets, called blocks, such that linear constraints only exist between variables in the same subset [93, 94, 189]. We leverage this observation to decompose a large polyhedron into a set of smaller polyhedra, thus reducing the asymptotic complexity of the Polyhedra domain. However, maintaining decomposition online is challenging because over 40 Polyhedra transformers change the partitions dynamically and in non-trivial ways: blocks in the partitions can merge, split, grow, or shrink during analysis. Note that an exact partition cannot be computed a priori as it depends on the exact Polyhedra generated during the analysis. Therefore, static approaches for computing the partition lose significant precision [27].

To ensure our method does not lose precision, we develop a theoretical framework that asserts how partitions are modified during analysis. We then use our

theory to design new abstract transformers for Polyhedra. Our framework guarantees that the original polyhedron can be recovered exactly from the decomposed polyhedra at each step of the analysis. Thus our decomposed analysis produces the same fixpoint and has the same convergence rate as the original analysis. Interestingly as we will show in the next chapter, with a non-trivial extension our framework can be used for decomposing other numerical domains without losing precision, not only Polyhedra.

**MAIN CONTRIBUTIONS** Our main contributions are:

- A theoretical framework for decomposing Polyhedra analysis. This framework allows for efficient maintenance of decomposition throughout the analysis *without* losing precision.
- New algorithms for Polyhedra transformers that leverage decomposition based on the theory. The algorithms are further optimized using novel optimizations exploiting sparsity.
- A complete implementation of our Polyhedra transformers in the form of a state-of-the-art library ELINA publicly available at <https://github.com/ethsri/ELINA>.
- An evaluation of the effectiveness of our approach showing massive gains in both space and time over state-of-the-art approaches on a large number of benchmarks, including Linux device drivers. For instance, we obtain a 170x speedup on the largest benchmark containing  $> 50K$  lines of code. In many other cases, the analysis with our approach terminates whereas other implementations abort without result.

We note that our decomposed Polyhedra domain analysis can be seen as an instance of cofibred domains from [203].

## 2.1 BACKGROUND ON POLYHEDRA ANALYSIS

In this section, we first introduce the necessary background on Polyhedra domain analysis. We present two ways to represent polyhedra and define the Polyhedra domain including its transformers. We conclude the section by discussing their asymptotic complexity.

**NOTATION** Lower case letters ( $a, b, \dots$ ) represent column vectors and integers ( $g, k, \dots$ ). Upper case letters  $A, D$  represent matrices whereas  $O, P, Q$  are polyhedra. Greek ( $\alpha, \beta, \dots$ ) and calligraphic letters ( $\mathcal{P}, \mathcal{C}, \dots$ ) represent scalars and sets respectively.



### 2.1.1 Representation of Polyhedra

Let  $x = (x_1, x_2, \dots, x_n)^T$  be a column vector of program variables. A convex closed polyhedron  $P \subseteq \mathbb{Q}^n$  that captures linear constraints among variables in  $x$  can be represented in two equivalent ways: the *constraint representation* and the *generator representation* [151]. Both are introduced next.

**CONSTRAINT REPRESENTATION** This representation encodes a polyhedron  $P$  as an intersection of:

- A finite number of closed half spaces of the form  $a^T x \leq \beta$ .
- A finite number of subspaces of the form  $d^T x = \xi$ .

Collecting these yields matrices  $A, D$  and vectors of rational numbers  $b, e$  such that the polyhedron  $P$  can be written as:

$$P = \{x \in \mathbb{Q}^n \mid Ax \leq b \text{ and } Dx = e\}. \quad (2.1)$$

The associated *constraint set*  $\mathcal{C}$  of  $P$  is defined as  $\mathcal{C} = \mathcal{C}_P = \{Ax \leq b, Dx = e\}$ .

**GENERATOR REPRESENTATION** This representation encodes the polyhedron  $P$  as the convex hull of:

- A finite set  $\mathcal{V} \subset \mathbb{Q}^n$  of vertices  $v_i$ .
- A finite set  $\mathcal{R} \subseteq \mathbb{Q}^n$  representing rays.  $r_i \in \mathbb{R}$  are direction vectors of infinite edges of the polyhedron with one end bounded. The rays always start from a vertex in  $\mathcal{V}$ .
- A finite set  $\mathcal{Z} \subseteq \mathbb{Q}^n$  representing lines<sup>2</sup>.  $z_i \in \mathbb{Z}$  are direction vectors of infinite edges of the polyhedron with both ends unbounded. Each such line passes through a vertex in  $\mathcal{V}$ .

Thus, every  $x \in P$  can be written as:

$$x = \sum_{i=1}^{|\mathcal{V}|} \lambda_i v_i + \sum_{i=1}^{|\mathcal{R}|} \mu_i r_i + \sum_{i=1}^{|\mathcal{Z}|} \nu_i z_i, \quad (2.2)$$

where  $\lambda_i, \mu_i \geq 0$ , and  $\sum_{i=1}^{|\mathcal{V}|} \lambda_i = 1$ . The above vectors are the *generators* of  $P$  and are collected in the set  $\mathcal{G} = \mathcal{G}_P = \{\mathcal{V}, \mathcal{R}, \mathcal{Z}\}$ .

---

<sup>2</sup> one dimensional affine subspaces.

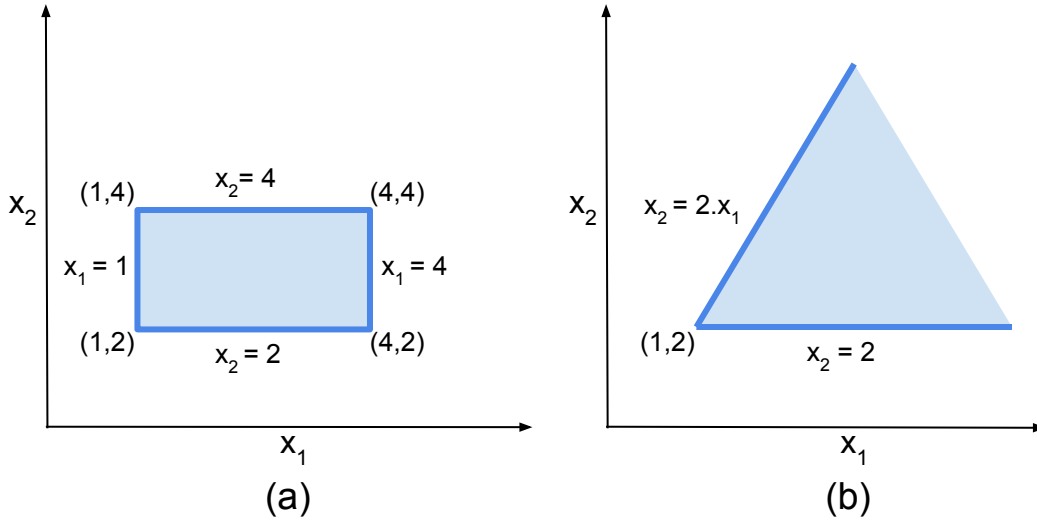


Figure 2.1: Two representations of polyhedron defined over variables  $x_1$  and  $x_2$ . (a) Bounded polyhedron; (b) unbounded polyhedron.

**Example 2.1.1.** Fig. 2.1 shows two examples of both representations for polyhedra. In Fig. 2.1(a) the polyhedron  $P$  is bounded and can be represented as either the intersection of four closed half spaces or as the convex hull of four vertices:

$$\mathcal{C} = \{-x_1 \leq -1, x_1 \leq 4, -x_2 \leq -2, x_2 \leq 4\}, \text{ or}$$

$$\mathcal{G} = \{\mathcal{V} = \{(1,2), (1,4), (4,2), (4,4)\}, \mathcal{R} = \emptyset, \mathcal{Z} = \emptyset\}.$$

Note that the sets of rays  $\mathcal{R}$  and lines  $\mathcal{Z}$  are empty in this case.

In Fig. 2.1(b), the polyhedron  $P$  is unbounded and can be represented either as the intersection of two closed half planes or as the convex hull of two rays starting at vertex  $(1,2)$ :

$$\mathcal{C} = \{-x_2 \leq -2, x_2 \leq 2 \cdot x_1\}, \text{ or}$$

$$\mathcal{G} = \{\mathcal{V} = \{(1,2)\}, \mathcal{R} = \{(1,2), (1,0)\}, \mathcal{Z} = \emptyset\}.$$

To reduce clutter, we abuse notation and often write  $P = (\mathcal{C}, \mathcal{G})$  since our algorithms, introduced later, maintain both representations. Both  $\mathcal{C}$  and  $\mathcal{G}$  represent *minimal* sets, i.e., they do not contain redundancy.

### 2.1.2 Polyhedra Domain

The Polyhedra domain is commonly used in static analysis to derive invariants that hold for all executions of the program starting from a given initial state. These invariants can be used to prove safety properties in programs like the absence of buffer overflow, division by zero and others [200, 201, 209]. The Polyhedra domain is a fully relational numerical domain, i.e., it can encode all possible linear constraints between program variables. Thus, it is more expressive than weakly relational domains such as Octagon [142], Pentagon [134] or Zone [140], which restrict

```

if(*) y:=2 · x-1; else y:=2 · x-2;
assert(y<=2 · x);

```

Figure 2.2: Code with assertion for static analysis.

the set of linear inequalities. The restrictions limit the set of assertions that can be proved using these domains. For example, the assertion in the code in Fig. 2.2 cannot be expressed using weakly relational domains whereas Polyhedra can express and prove the property. The expressivity of the Polyhedra domain comes at higher cost: it has asymptotic worst-case exponential complexity in both time and space.

The Polyhedra abstract domain consists of the *polyhedra lattice*  $(\mathcal{P}, \sqsubseteq, \sqcup, \sqcap, \perp, \top)$  and a set of transformers.  $\mathcal{P}$  is the set of convex closed polyhedra ordered by standard set inclusion:  $\sqsubseteq = \subseteq$ . The least upper bound ( $\sqcup$ ) of two polyhedra  $P$  and  $Q$  is the convex hull of  $P$  and  $Q$ , which, in general, is larger than the union  $P \cup Q$ . The greatest lower bound ( $\sqcap$ ) of  $P$  and  $Q$  is simply the intersection  $P \cap Q$ . The top element  $\top = \mathbb{Q}^n$  in the lattice is encoded by  $\mathcal{C} = \emptyset$  or generated by  $n$  linearly independent lines. The bottom element ( $\perp$ ) is represented by any unsatisfiable set of constraints in  $\mathcal{C}$  or with  $\mathcal{G} = \emptyset$ .

**TRANSFORMERS** The transformers used in the Polyhedra domain for program analysis model the effect of various program statements such as assignments and conditionals as well as control flow such as loops and branches on the program states approximated by polyhedra. There are also transformers for checking and accelerating analysis convergence towards a fixpoint. Overall, a standard implementation of the Polyhedra domain contains more than 40 transformers [12, 104]. We introduce the most frequently used transformers in Polyhedra domain:

*Inclusion test:* this transformer tests if  $P \sqsubseteq Q$  for the given polyhedra  $P$  and  $Q$ .

*Equality test:* this transformer tests if two polyhedra  $P$  and  $Q$  are equal by double inclusion.

*Join:* this transformer computes  $P \sqcup Q$ , i.e., the convex hull of  $P$  and  $Q$ .

*Meet:* this transformer computes  $P \sqcap Q = P \cap Q$ .

*Widening:* as the polyhedra lattice has infinite height, the analysis requires *widening* to accelerate convergence. The result of the widening transformer [13]  $P \nabla Q$  contains constraints from  $\mathcal{C}_Q$  that are either present in  $\mathcal{C}_P$  or that can replace a constraint in  $\mathcal{C}_P$  without changing  $P$ . Using the constraint representation it is defined as:

$$\mathcal{C}_{P \nabla Q} = \begin{cases} \mathcal{C}_Q, & \text{if } P = \perp; \\ \mathcal{C}'_P \cup \mathcal{C}'_Q, & \text{otherwise;} \end{cases} \quad (2.3)$$

where:

$$\begin{aligned} \mathcal{C}'_P &= \{c \in \mathcal{C}_P \mid \mathcal{C}_Q \models c\}, \\ \mathcal{C}'_Q &= \{c \in \mathcal{C}_Q \mid \exists c' \in \mathcal{C}_P, \mathcal{C}_P \models c \text{ and } ((\mathcal{C}_P \setminus c') \cup \{c\}) \models c'\}. \end{aligned}$$

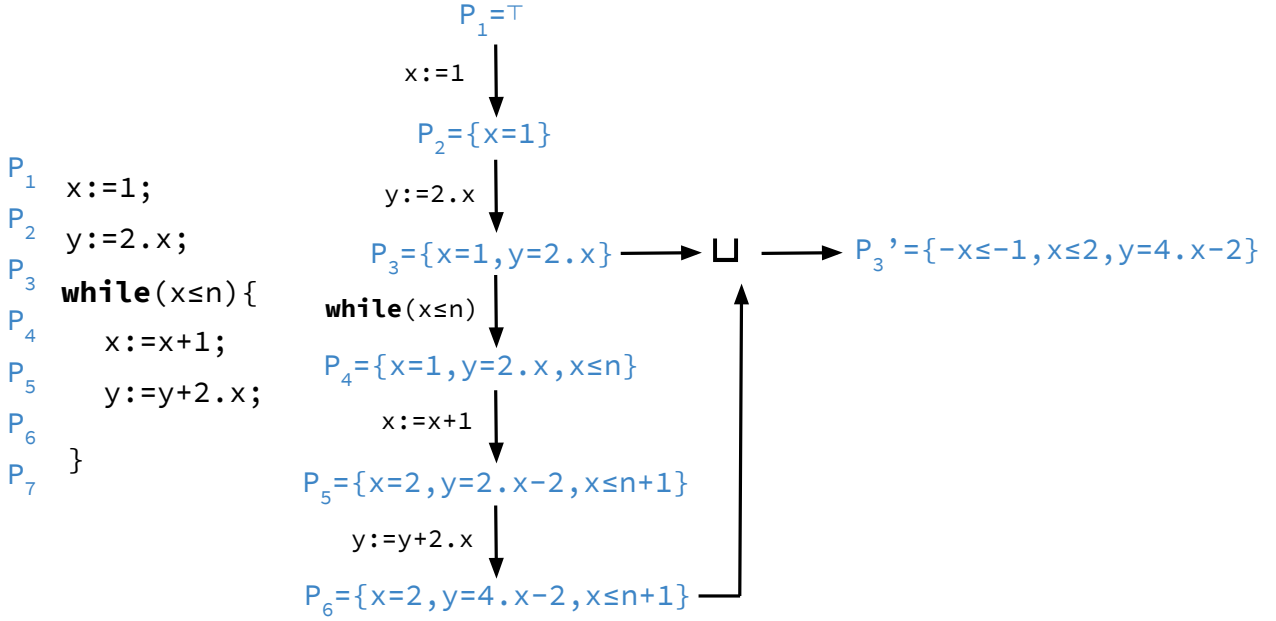


Figure 2.3: Polyhedra domain analysis (first iteration) on the example program on the left. The polyhedra are shown in constraint representation.

where  $\mathcal{C} \models c$  tests whether the constraint  $c$  can be *entailed* by the constraints in  $\mathcal{C}$ .

Next we introduce the transformers corresponding to program statements. For simplicity, we assume that the expression  $\delta$  on the right hand side of both conditional and assignment statements is affine i.e.,  $\delta = a^T \cdot x + \epsilon$ , where  $a \in \mathbb{Q}^n$ ,  $\epsilon \in \mathbb{Q}$  are constants. Non-linear expressions can be approximated by affine expressions using the techniques described in [141].

*Conditional:* Let  $\otimes \in \{\leq, =\}$ ,  $1 \leq i \leq n$ , and  $\alpha \in \mathbb{Q}$ , the conditional statement  $\alpha \cdot x_i \otimes \delta$  adds the constraint  $(\alpha - a_i)x_i \otimes \delta - a_i \cdot x_i$  to the constraint set  $\mathcal{C}$ .

*Assignment:* The transformer for an assignment  $x_i := \delta$  first adds a new variable  $x'_i$  to the polyhedron  $P$  and then augments  $\mathcal{C}$  with the constraint  $x'_i - \delta = 0$ . The variable  $x_i$  is then projected out [102] from the constraint set  $\mathcal{C} \cup \{x'_i - \delta = 0\}$ . Finally, the variable  $x'_i$  is renamed back to  $x_i$ .

### 2.1.3 Polyhedra Domain Analysis: Example

Fig. 2.3 shows a simple program that computes the sum of the first  $n$  even numbers where a polyhedron  $P_\ell$  is associated with each line  $\ell$  in the program. At the fixpoint, the polyhedron  $P_\ell$  represents invariants that hold for all executions of the program before executing the statement at line  $\ell$ . Here, we work only with the constraint representation of polyhedra. The analysis proceeds iteratively by selecting the polyhedron at a given line, say  $P_1$ , then applying the transformer for the statement at that program point ( $x := 1$  in this case) on that polyhedron, and producing a new polyhedron, in this case  $P_2$ . The analysis terminates when a fixpoint is reached, i.e., when further iterations do not add extra points to any polyhedra.

**FIRST ITERATION** The initial program state does not restrict possible values of the program variables  $x, y, n$ . Thus initially, polyhedron  $P_1$  is set to top ( $\top$ ). Next, the analysis applies the transformer for the assignment  $x:=1$  to  $P_1$ , producing  $P_2$ . The set  $\mathcal{C}_1$  is empty and the transformer adds constraint  $x = 1$  to obtain  $P_2$ . The next statement assigns to  $y$ . Since  $\mathcal{C}_2$  does not contain any constraint involving  $y$ , the transformer for the assignment  $y:=2 \cdot x$  adds  $y = 2 \cdot x$  to obtain  $P_3$ . Next, the conditional statement for the loop is processed: that transformer adds the constraint  $x \leq n$  to obtain polyhedron  $P_4$ . The assignment statement  $x:=x+1$  inside the loop assigns to  $x$  which is already present in the set  $\mathcal{C}_4$ . Thus, a new variable  $x'$  is introduced and constraint  $x' - x - 1 = 0$  is added to  $\mathcal{C}_4$  producing:

$$\mathcal{C}'_5 = \{x = 1, y = 2 \cdot x, x \leq n, x' - x - 1 = 0\}$$

The transformer then projects out  $x$  from  $\mathcal{C}'_5$  to produce:

$$\mathcal{C}''_5 = \{x' = 2, y = 2 \cdot x' - 2, x' \leq n + 1\}$$

Variable  $x'$  is then renamed to  $x$  to produce the final set for  $P_5$ :

$$\mathcal{C}_5 = \{x = 2, y = 2 \cdot x - 2, x \leq n + 1\}$$

The next assignment  $y:=y+2 \cdot x$  is handled similarly to produce  $P_6$ .

**NEXT ITERATIONS** The analysis then returns to the head of the while loop and propagates the polyhedron  $P_6$  to that point. To compute the new program state at the loop head, it now needs to compute the union of  $P_6$  with the previous polyhedron  $P_3$  at that point. Since the union of convex polyhedra is usually not convex, it is approximated using the join transformer ( $\sqcup$ ) to yield the polyhedron  $P'_3$ .

The analysis then checks if the new polyhedron  $P'_3$  at the loop head is included in  $P_3$  using inclusion testing ( $\sqsubseteq$ ). If yes, then no new information was added and the analysis terminates. However, here,  $P'_3 \not\sqsubseteq P_3$  and so the analysis continues. After several iterations, the widening transformer ( $\nabla$ ) may be applied at the loop head along with the join to accelerate convergence.

#### 2.1.4 Transformers and Asymptotic Complexity

The asymptotic time complexity of Polyhedra transformers depends on how a polyhedron is represented, as shown in Table 2.1. In the table,  $n$  is the number of variables,  $m$  is the number of constraints in  $\mathcal{C}$ ,  $g = |\mathcal{V}| + |\mathcal{R}| + |\mathcal{Z}|$  is the number of generators in  $\mathcal{G}$  and  $\text{LP}(m, n)$  is the complexity of solving a linear program with  $m$  constraints and  $n$  variables. For binary transformers like join, meet and others,  $m$  and  $g$  denote respectively the maximum of the number of constraints and generators in  $P$  and  $Q$ . The column *Constraint* shows the cost of computing the constraint

Table 2.1: Asymptotic complexity of Polyhedra operators with different representations.

Operator	Constraint	Generator	Both
Inclusion ( $\sqsubseteq$ )	$O(m \cdot \text{LP}(m, n))$	$O(g \cdot \text{LP}(g, n))$	$O(n \cdot g \cdot m)$
Join ( $\sqcup$ )	$O(n \cdot m^{2^{n+1}})$	$O(n \cdot g)$	$O(n \cdot g)$
Meet ( $\sqcap$ )	$O(n \cdot m)$	$O(n \cdot g^{2^{n+1}})$	$O(n \cdot m)$
Widening ( $\nabla$ )	$O(m \cdot \text{LP}(m, n))$	$O(g \cdot \text{LP}(g, n))$	$O(n \cdot g \cdot m)$
Conditional	$O(n)$	$O(n \cdot g^{2^{n+1}})$	$O(n)$
Assignment	$O(n \cdot m^2)$	$O(n \cdot g)$	$O(n \cdot g)$

set for the result starting from the input constraint set(s); the column *Generator* has similar meaning for the generators. The column *Both* shows the asymptotic cost of computing at least one of the representations for the result when both representations are available for the input(s).

**TRANSFORMERS VS. REPRESENTATIONS** Table 2.1 shows transformers, such as meet ( $\sqcap$ ), are considerably more efficient to compute using the constraint representation whereas other transformers, such as join ( $\sqcup$ ), are cheaper using the generator representation. Transformers such as inclusion testing ( $\sqsubseteq$ ) are most efficient when one of the two participating polyhedron is represented via constraints and the other via generators. As a result, popular libraries such as NewPolka [104] and PPL [12] maintain both representations of polyhedra during analysis. We follow the same approach here and thus each polyhedron  $P$  is represented as  $P = (\mathcal{C}, \mathcal{G})$ .

Maintaining both representations requires conversion. For example, the meet of two polyhedra can be efficiently computed by taking the union of the respective constraints. Conversion is then required to compute the corresponding generator representation of the result. As is common, we use Chernikova’s [50, 123] algorithm but with our own optimized implementation (Section 2.4.1) for converting from the constraint to the generator representation and vice-versa. The conversion algorithm also minimizes both representations. We note that other conversion algorithms [2, 10, 75] can also be used.

**CONVERSION BETWEEN REPRESENTATIONS** When both representations are available, all Polyhedra transformers become polynomial (last column of Table 2.1) and Chernikova’s algorithm becomes the bottleneck for the analysis as it has worst case exponential complexity for conversion in either direction. We refer the reader to [50, 123] for details of the algorithm. There are two approaches for reducing the cost of these conversions: *lazy* and *eager*.

The *lazy* approach computes the conversion only when required to amortize the cost over many operations. For example, in Fig. 2.3, there are a number of conditional checks and assignments in succession so one can keep working with the constraint representation and compute the generator one only when needed

(e.g., at the loop head when join is needed). The *eager* approach computes the conversion after every operation. Chernikova’s algorithm is incremental, which means that for transformers which add constraints or generators such as meet ( $\sqcap$ ), join ( $\sqcup$ ), conditional and others, the conversion needs to be computed only for the added constraints or generators. Because of this, in some cases eager can be faster than lazy. While our transformers and algorithms are compatible with both approaches, we use the eager approach in this work.

## 2.2 POLYHEDRA DECOMPOSITION

We next present our key insight and show how to leverage it to speed up program analysis using Polyhedra. Our observation is that the set of program variables can be partitioned into smaller subsets with respect to the polyhedra arising during analysis such that no constraints exist between variables in different subsets. This allows us to decompose a large polyhedron into a set of smaller polyhedra, which reduces the space complexity of the analysis. For example, the  $n$ -dimensional hypercube requires  $2^n$  generators whereas with decomposition only  $2n$  generators are required. The original polyhedron can be recovered exactly using the decomposed Polyhedra; thus, analysis precision is not affected. Further, the decomposition allows the expensive polyhedra transformers to operate on smaller polyhedra, thus reducing their time complexity without losing precision.

We first introduce our notation for partitions. Then, we introduce the theoretical underpinning of our work: the interaction between the Polyhedra domain transformers and the partitions.

### 2.2.1 Partitions

Let  $\mathcal{X} = \{x_1, x_2, \dots, x_n\}$  be the set of  $n$  variables. For a given polyhedron,  $\mathcal{X}$  can be partitioned into subsets  $\mathcal{X}_k$  we call *blocks* such that constraints only exist between variables in the same block. Each unconstrained variable  $x_i$  yields a singleton block  $\{x_i\}$ . We refer to this unique, finest partition as  $\pi = \pi_P = \{\mathcal{X}_1, \mathcal{X}_2, \dots, \mathcal{X}_r\}$ .

**Example 2.2.1.** Consider

$$\begin{aligned} \mathcal{X} &= \{x_1, x_2, x_3\} \text{ and} \\ P &= \{x_1 + 2 \cdot x_2 \leq 3\}. \end{aligned}$$

Here,  $\mathcal{X}$  is partitioned into two blocks:  $\mathcal{X}_1 = \{x_1, x_2\}$  and  $\mathcal{X}_2 = \{x_3\}$ . Now consider

$$P = \{x_1 + 2 \cdot x_2 \leq 3, 3 \cdot x_2 + 4 \cdot x_3 \leq 1\}.$$

Here, the partition of  $\mathcal{X}$  has only one block  $\mathcal{X}_1 = \{x_1, x_2, x_3\}$ .

The partition  $\pi_P$  decomposes the polyhedron  $P$  defined over  $\mathcal{X}$  into a set of smaller polyhedra  $P_k$  which we call *factors*. Each factor  $P_k$  is defined only over

the variables in  $\mathcal{X}_k$ . The polyhedron  $P$  can be recovered from the factors  $P_k$  by computing the union of the constraints  $\mathcal{C}_{P_k}$  and the Cartesian product of the generators  $\mathcal{G}_{P_k}$ . For this, we introduce the  $\bowtie$  transformer defined as:

$$\begin{aligned} P &= P_1 \bowtie P_2 \bowtie \dots \bowtie P_r \\ &= (\mathcal{C}_{P_1} \cup \mathcal{C}_{P_2} \dots \cup \mathcal{C}_{P_r}, \mathcal{G}_{P_1} \times \mathcal{G}_{P_2} \dots \times \mathcal{G}_{P_r}). \end{aligned} \quad (2.4)$$

**Example 2.2.2.** The polyhedron  $P$  in Fig. 2.1 (a) has no constraints between variables  $x_1$  and  $x_2$ . Thus,  $\mathcal{X} = \{x_1, x_2\}$  can be partitioned into blocks:  $\pi_P = \{\{x_1\}, \{x_2\}\}$  with corresponding factors  $P_1 = (\mathcal{C}_{P_1}, \mathcal{G}_{P_1})$  and  $P_2 = (\mathcal{C}_{P_2}, \mathcal{G}_{P_2})$  where:

$$\begin{aligned} \mathcal{C}_{P_1} &= \{-x_1 \leq -1, x_1 \leq 4\} & \mathcal{C}_{P_2} &= \{-x_2 \leq -2, x_2 \leq 4\} \\ \mathcal{G}_{P_1} &= \{\{(1), (4)\}, \emptyset, \emptyset\} & \mathcal{G}_{P_2} &= \{\{(2), (4)\}, \emptyset, \emptyset\} \end{aligned}$$

The original polyhedron can be recovered from  $P_1$  and  $P_2$  as  $P = P_1 \bowtie P_2 = (\mathcal{C}_{P_1} \cup \mathcal{C}_{P_2}, \mathcal{G}_{P_1} \times \mathcal{G}_{P_2})$ .

The set  $\mathcal{L}$  consisting of all partitions of  $\mathcal{X}$  forms a *partition lattice*  $(\mathcal{L}, \sqsubseteq, \sqcup, \sqcap, \perp, \top)$ . The elements  $\pi$  of the lattice are ordered as follows:  $\pi \sqsubseteq \pi'$ , if every block of  $\pi$  is included in some block of  $\pi'$  ( $\pi$  "is finer" than  $\pi'$ ). This lattice contains the usual transformers of *least upper bound* ( $\sqcup$ ) and *greatest lower bound* ( $\sqcap$ ). In the partition lattice,  $\top = \{\mathcal{X}\}$  and  $\perp = \{\{x_1\}, \{x_2\}, \dots, \{x_n\}\}$ .

**Example 2.2.3.** For example,

$$\{\{x_1, x_2\}, \{x_3\}, \{x_4\}, \{x_5\}\} \sqsubseteq \{\{x_1, x_2, x_3\}, \{x_4\}, \{x_5\}\}$$

Now consider,

$$\begin{aligned} \pi &= \{\{x_1, x_2\}, \{x_3, x_4\}, \{x_5\}\} \text{ and} \\ \pi' &= \{\{x_1, x_2, x_3\}, \{x_4\}, \{x_5\}\} \end{aligned}$$

Then,

$$\begin{aligned} \pi \sqcup \pi' &= \{\{x_1, x_2, x_3, x_4\}, \{x_5\}\} \text{ and} \\ \pi \sqcap \pi' &= \{\{x_1, x_2\}, \{x_3\}, \{x_4\}, \{x_5\}\} \end{aligned}$$

**Definition 2.2.1.** We call a partition  $\pi$  *permissible* for  $P$  if there are no variables  $x_i$  and  $x_j$  in different blocks of  $\pi$  related by a constraint in  $P$ , i.e., if  $\pi \sqsupseteq \pi_P$ .

Note, that the finest partition  $\pi_\top$  for the top ( $\top$ ) and the bottom ( $\perp$ ) polyhedra is the bottom element in the partition lattice, i.e.,  $\pi_\top = \pi_\perp = \perp$ . Thus, every partition is permissible for these.

### 2.2.2 Transformers and Partitions

The decomposed transformers require the computation of the output partition before the output is actually computed. We next show how the output partitions



are computed. The optimality of the computed partitions depends upon the degree to which the polyhedra are observed. The finest partition for the output of a Polyhedra transformer can always be computed from scratch by taking the output polyhedron and connecting the variables that occur in the same constraint in that polyhedron. However, this nullifies any performance gains as standard transformer needs to be applied for computing the output polyhedron. For efficiency, we compute the output partitions based on limited observation of the inputs. The partition for the output of transformers such as meet, conditionals, assignment, and widening is computed from the corresponding partitions of input polyhedra  $P$  and  $Q$ . For the join however, to ensure we do not end up with a trivial and imprecise partition, we need to examine  $P$  and  $Q$  (discussed later in the section). Our approach to handling the join partition is key to achieving significant analysis speedups.

We also note that the same polyhedron can have multiple constraint representations with different finest partitions as shown in the example below.

**Example 2.2.4.** The polyhedron  $P = \{x_1 = 0, x_2 = 0\}$  has the partition  $\pi_P = \{\{x_1\}, \{x_2\}\}$ . This polyhedron can also be represented as  $P' = \{x_1 = 0, x_2 = x_1\}$  with the associated partition  $\pi_{P'} = \{\{x_1, x_2\}\}$ . Here  $P = P'$  but  $\pi_P \neq \pi_{P'}$ .

We note that the conversion algorithm performs transformations to change the polyhedron representation. The exact output after such transformations depends on the polyhedron and cannot be determined statically. In this work, we do not model the effect of such transformations.

We next provide optimal partitions under our observation model. For polyhedron  $P$ , we denote the associated optimal partition in our model as  $\pi_P^{\text{obs}}$ . We will present a refinement of our observation model to obtain finer output partitions at small extra cost in Section 3.4. We will also present conditions when  $\pi_P^{\text{obs}} = \pi_P$ .

**MEET** The constraint set for the meet  $P \sqcap Q$  is the union  $\mathcal{C}_P \cup \mathcal{C}_Q$ . Thus, overlapping blocks  $\mathcal{X}_i \in \pi_P$  and  $\mathcal{X}_j \in \pi_Q$  will merge into one block in  $\pi_{P \sqcap Q}$ . This yields

**Lemma 2.2.1.** Let  $P$  and  $Q$  be two polyhedra with  $P \sqcap Q \neq \perp$ . Then  $\pi_{P \sqcap Q} \sqsubseteq \pi_{P \sqcap Q}^{\text{obs}} = \pi_P \sqcup \pi_Q$ .

**CONDITIONAL AND ASSIGNMENT** The conditional and assignment statements ( $x_i := \delta$  and  $\alpha \cdot x_i \otimes \delta$ ) create new constraints between program variables. Thus, to compute the partitions for the outputs of these transformers, we first compute a block  $\mathcal{B}$  which contains all variables affected by the statement. Let  $\mathcal{E}$  be the set of all variables  $x_j$  with  $a_j \neq 0$  in  $\delta = a^\top \cdot x + \epsilon$ , then  $\mathcal{B} = \mathcal{E} \cup \{x_i\}$ . To express the fusion incurred by  $\mathcal{B}$ , we introduce the following:

**Definition 2.2.2.** Let  $\pi$  be a partition of  $\mathcal{X}$  and  $\mathcal{B} \subseteq \mathcal{X}$ , then  $\pi \uparrow \mathcal{B}$  is the finest partition  $\pi'$  such that  $\pi \sqsubseteq \pi'$  and  $\mathcal{B}$  is a subset of an element of  $\pi'$ .

As discussed, the transformer for the conditional statement  $\alpha \cdot x_i \otimes \delta$  adds constraint  $(\alpha - a_i) \cdot x_i \otimes \delta - a_i \cdot x_i$  to  $\mathcal{C}_P$  to produce the set  $\mathcal{C}_O$  for the output  $O$ . Thus,

in  $\pi_O$ , all blocks  $\mathcal{X}_i \in \pi_P$  that overlap with  $\mathcal{B}$  will merge into one, whereas non-overlapping blocks remain independent. Thus, we get the following lemma for calculating  $\pi_O$ .

**Lemma 2.2.2.** Let  $P$  be the input polyhedra and let  $\mathcal{B}$  be the block corresponding to the conditional  $\alpha \cdot x_i \otimes \delta$ . If  $O \neq \perp$ , then  $\pi_O \sqsubseteq \pi_O^{\text{obs}} = \pi_P \uparrow \mathcal{B}$ .

$\pi_O$  for the output  $O$  of the transformer for the assignment  $x_i := \delta$  can be computed similarly to that of the conditional transformer.

**Lemma 2.2.3.** Let  $P$  be the input polyhedra and let  $\mathcal{B}$  be the block corresponding to an assignment  $x_i := \delta$ . Then  $\pi_O \sqsubseteq \pi_O^{\text{obs}} = \pi_P \uparrow \mathcal{B}$ .

**WIDENING** Like the join, the partition for widening ( $\nabla$ ) depends not only on partitions  $\pi_P$  and  $\pi_Q$ , but also on the exact form of  $P$  and  $Q$ . By definition, the constraint set for  $P \nabla Q$  contains only constraints from  $Q$ . Thus, the partition for  $P \nabla Q$  satisfies

**Lemma 2.2.4.** For polyhedra  $P$  and  $Q$ ,  $\pi_{P \nabla Q} \sqsubseteq \pi_{P \nabla Q}^{\text{obs}} = \pi_Q$ .

Note that the widening transformer can potentially remove all constraints containing a variable, making the variable unconstrained. Thus, in general,  $\pi_{P \nabla Q} \neq \pi_Q$ .

**JOIN** Let  $\mathcal{C}_P = \{A_1 \cdot x \leq b_1\}$  and  $\mathcal{C}_Q = \{A_2 \cdot x \leq b_2\}$ <sup>3</sup> and  $\mathcal{Y} = \{x'_1, x'_2, \dots, x'_n, \lambda\}$ , then the constraint set  $\mathcal{C}_{P \sqcup Q}$  for the join of  $P$  and  $Q$  can be computed by projecting out variables  $y_i \in \mathcal{Y}$  from the following set  $\mathcal{S}$  of constraints:

$$\mathcal{S} = \{A_1 \cdot x' \leq b_1 \cdot \lambda, A_2 \cdot (x - x') \leq b_2 \cdot (1 - \lambda), -\lambda \leq 0, \lambda \leq 1\}. \quad (2.5)$$

The Fourier-Motzkin elimination algorithm [102] is used for this projection. The algorithm starts with  $\mathcal{S}_0 = \mathcal{S}$  and projects out variables iteratively one after another so that  $\mathcal{C}_{P \sqcup Q} = \mathcal{S}_{n+1}$ . Let  $\mathcal{S}_{i-1}$  be the constraint set obtained after projecting out the first  $i - 1$  variables in  $\mathcal{Y}$ . Then  $y_i \in \mathcal{Y}$  is projected out to produce  $\mathcal{S}_i$  as follows:

$$\begin{aligned} \mathcal{S}_{y_i}^+ &= \{c \mid c \in \mathcal{S}_{i-1} \text{ and } a_i > 0\}, \\ \mathcal{S}_{y_i}^- &= \{c \mid c \in \mathcal{S}_{i-1} \text{ and } a_i < 0\}, \\ \mathcal{S}_{y_i}^0 &= \{c \mid c \in \mathcal{S}_{i-1} \text{ and } a_i = 0\}, \\ \mathcal{S}_{y_i}^\pm &= \{\mu \cdot c_1 + \nu \cdot c_2 \mid (c_1, c_2) \in \mathcal{S}_{y_i}^+ \times \mathcal{S}_{y_i}^- \text{ and } \mu \cdot a_{1_i} + \nu \cdot a_{2_i} = 0\}, \\ \mathcal{S}_i &= \mathcal{S}_{y_i}^0 \cup \mathcal{S}_{y_i}^\pm. \end{aligned} \quad (2.6)$$

Each iteration can potentially produce a quadratic number of new constraints, many of which are redundant. The redundant constraints are removed for efficiency.

The partition of  $P \sqcup Q$  depends in non-trivial ways on  $P$  and  $Q$ . In particular,  $\pi_{P \sqcup Q}$  has no general relationship to either  $\pi_P \sqcup \pi_Q$  or  $\pi_P \sqcap \pi_Q$ . The following example illustrates this:

<sup>3</sup> We assume equalities are encoded as symmetric pairs of opposing inequalities for simplicity.

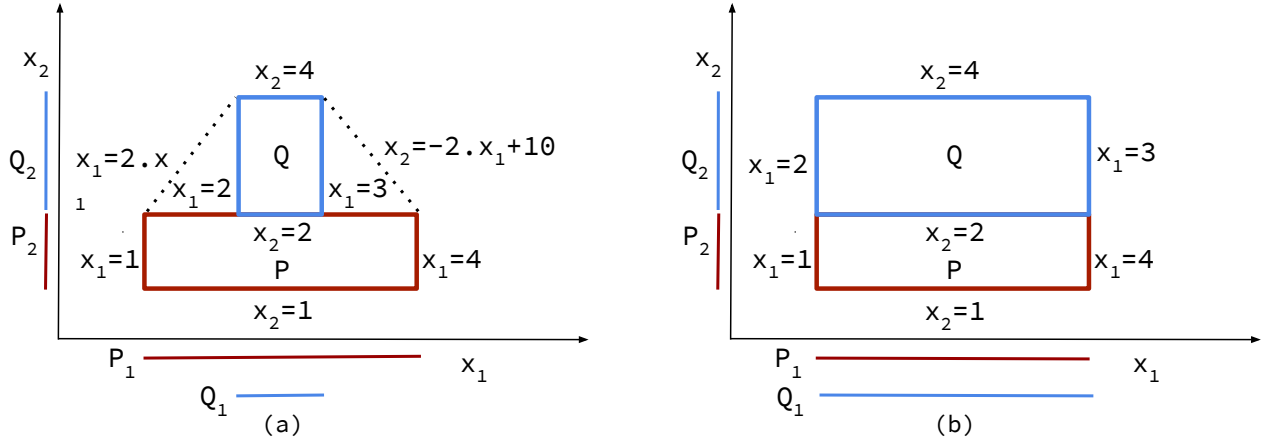


Figure 2.4: Two examples of  $P \sqcup Q$  with  $\pi_P = \pi_Q = \{\{x_1\}, \{x_2\}\}$ . (a)  $P_1 \neq Q_1, P_2 \neq Q_2$ ; (b)  $P_1 = Q_1, P_2 \neq Q_2$ .

**Example 2.2.5.** Let

$$\begin{aligned}
 P &= \{\{x_1 - x_2 \leq 0, x_1 \leq 0\}, \{x_3 = 1\}\} \text{ and} \\
 Q &= \{\{x_1 \leq 2\}, \{x_3 = 0\}\} \text{ with} \\
 \pi_P &= \{\{x_1, x_2\}, \{x_3\}\} \text{ and} \\
 \pi_Q &= \{\{x_1\}, \{x_2\}, \{x_3\}\}.
 \end{aligned}$$

In this case we have,

$$\begin{aligned}
 P \sqcup Q &= \{\{x_1 + 2 \cdot x_3 \leq 2, -x_3 \leq 0, x_3 \leq 1\}\} \text{ and} \\
 \pi_{P \sqcup Q} &= \{\{x_1, x_3\}, \{x_2\}\}.
 \end{aligned}$$

However,

$$\begin{aligned}
 \pi_P \sqcup \pi_Q &= \{\{x_1, x_2\}, \{x_3\}\} \text{ and} \\
 \pi_P \sqcap \pi_Q &= \{\{x_1\}, \{x_2\}, \{x_3\}\}.
 \end{aligned}$$

Thus, neither  $\pi_P \sqcup \pi_Q$  nor  $\pi_P \sqcap \pi_Q$  are permissible partitions for  $P \sqcup Q$ .

The theorem below identifies a case which enables us to compute a non-trivial permissible partition for  $P \sqcup Q$ . The theorem states that we can “transfer” a block from the input partitions to the output partition under certain conditions. It is a key enabler for the speedups shown later.

**Theorem 2.2.5.** *Let  $P$  and  $Q$  be two polyhedra with the same permissible partition  $\pi = \{X_1, X_2, \dots, X_r\}$  and let  $\pi'$  be a permissible partition for the join, that is,  $\pi_{P \sqcup Q} \sqsubseteq \pi'$ . If for any block  $X_k \in \pi$ ,  $P_k = Q_k$ , then  $X_k \in \pi'$ .*

*Proof.* Since both  $P$  and  $Q$  are partitioned according to  $\pi$ , the constraint set in (2.5) can be written for each  $X_k$  separately:

$$\{A_{1k} \cdot x'_k \leq b_{1k} \cdot \lambda, A_{2k} \cdot (x_k - x'_k) \leq b_{2k} \cdot (1 - \lambda), -\lambda \leq 0, \lambda \leq 1\}. \tag{2.7}$$

where  $x_k$  is column vector for the variables in  $\mathcal{X}_k$ .  $\lambda$  occurs in the constraint set for all blocks. For proving the theorem, we need to show that no variable in  $\mathcal{X}_k$  will have a constraint with a variable in  $\mathcal{X}_{k'} \in \pi$  after join. The variables in  $\mathcal{X}_k$  can have a constraint with the variables in  $\mathcal{X}_{k'}$  only by projecting out  $\lambda$ . Since  $P_k = Q_k$ ,  $\mathcal{C}_{P_k}$  and  $\mathcal{C}_{Q_k}$  are equivalent, we can assume  $A_{1k} = A_{2k}$  and  $b_{1k} = b_{2k}$ .<sup>4</sup> Inserting this into (2.7) we get

$$\{A_{1k} \cdot x'_k \leq b_{1k} \cdot \lambda, A_{1k} \cdot (x_k - x'_k) \leq b_{1k} \cdot (1 - \lambda), -\lambda \leq 0, \lambda \leq 1\}. \quad (2.8)$$

The result of the projection is independent of the order in which the variables are projected out. Thus, we can project out  $\lambda$  last. For proving the theorem, we need to show that it is possible to obtain all constraints for  $\mathcal{C}_{P_k \sqcup Q_k}$  before projecting out  $\lambda$  in (2.8). We add  $A_{1k} \cdot x'_k \leq b_{1k} \cdot \lambda$  and  $A_{1k} \cdot (x_k - x'_k) \leq b_{1k} \cdot (1 - \lambda)$  in (2.8) to project out all  $x'_k$  and obtain:

$$\{A_{1k} \cdot x_k \leq b_{1k}, -\lambda \leq 0, \lambda \leq 1\}. \quad (2.9)$$

Note that the constraint set in (2.9) does not contain all constraints generated by the Fourier-Motzkin elimination. Since  $P_k = P_k \sqcup P_k$ , we have  $\mathcal{C}_{P_k \sqcup Q_k} = \mathcal{C}_{P_k}$  and  $\mathcal{C}_{P_k}$  is included in the constraint set of (2.9); thus, the remaining constraints generated by the Fourier-Motzkin elimination are redundant. In (2.9), all constraints among the variables in  $\mathcal{X}_k$  are free from  $\lambda$ ; therefore, projecting out  $\lambda$  does not create new constraints for the variables in  $\mathcal{X}_k$ . Thus, there cannot be any constraint from a variable in  $\mathcal{X}_k$  to a variable in  $\mathcal{X}_{k'}$ .  $\square$

The proof of the theorem also yields the following result.

**Corollary 2.2.1.** If Theorem 2.2.5 holds, then  $P_k$  (and  $Q_k$ ) is a factor of  $P \sqcup Q$ .

**Example 2.2.6.** Fig. 2.4 shows two examples of  $P \sqcup Q$  where both  $P$  and  $Q$  have the same partition  $\pi_P = \pi_Q = \{\{x_1\}, \{x_2\}\}$ . In Fig. 2.4(a),

$$\begin{aligned} P &= \{\{x_1 = 1, x_1 = 4\}, \{x_2 = 1, x_2 = 2\}\}, \\ Q &= \{\{x_1 = 2, x_1 = 3\}, \{x_2 = 2, x_2 = 4\}\}. \end{aligned}$$

In this case,  $P_1 \neq Q_1$  and  $P_2 \neq Q_2$ ; thus,  $P \sqcup Q$  contains constraints  $x_2 = 2 \cdot x_1$  and  $x_2 = -2 \cdot x_1 + 10$  relating  $x_1$  and  $x_2$ , i.e.,  $\pi_{P \sqcup Q} = \{\{x_1, x_2\}\}$ .

In Fig. 2.4(b),

$$\begin{aligned} P &= \{\{x_1 = 1, x_1 = 4\}, \{x_2 = 1, x_2 = 2\}\}, \\ Q &= \{\{x_1 = 1, x_1 = 4\}, \{x_2 = 2, x_2 = 4\}\}. \end{aligned}$$

In this case,  $P_1 = Q_1$ . Thus, by Theorem 2.2.5,  $\{x_1\} \in \pi_{P \sqcup Q}$ , i.e.,  $\pi_{P \sqcup Q} = \{\{x_1\}, \{x_2\}\}$ .

<sup>4</sup> One can always perform a transformation so that  $A_{1k} = A_{2k}$  and  $b_{1k} = b_{2k}$  holds.

## 2.3 POLYHEDRA DOMAIN ANALYSIS WITH PARTITIONS

After presenting the theoretical background, we now discuss how we integrate partitioning in the entire analysis flow. The basic idea is to perform the analysis while maintaining the variable set partitioned, and thus the occurring polyhedra decomposed, as fine-grained as possible. The results from the previous section show that the main Polyhedra transformers can indeed maintain the partitions, even though these partitions change during the analysis. Crucially, under certain assumptions, even the join produces a non-trivial partitioned output. Note that there are no guarantees that for a given program, the partitions do not become trivial (i.e., equal to  $\{\mathcal{X}\}$ ); however, as our results later show, this is typically not the case and thus significant speedups are obtained. This should not be surprising: in complex programs, not all variables used are related to each other. For example, the individual conditional and assignment statements are usually defined over only a few variables. Similarly, the assumption for Theorem 2.2.5 usually holds as  $P$  could be the polyhedron at the loop head and  $Q$  the polyhedron at the loop exit. Since program loops modify only a few variables, the blocks of  $\mathcal{X}$  unaffected by the loop have equal factors in  $P$  and  $Q$ . However, there will be groups of variables that indeed develop relationships and these groups may change during execution. Our approach identifies and maintains such groups.

**MAINTAINING PRECISION** We emphasize that partitioning the variable set and thus decomposing polyhedra and transformers working on polyhedra, does not affect the overall precision of the result. That is, we neither lose nor gain precision in our analysis compared to prior approaches which do not use online partitioning. *The granularity of a partition only affects the cost, i.e., runtime and memory space, required for the analysis, but not the precision of its results.*

We now briefly discuss the data structures used for polyhedra and the maintenance of permissible partitions throughout the analysis. For the remainder of the chapter, permissible partitions will be denoted with  $\bar{\pi}_P \sqsupseteq \pi_P$ . The following sections then provide more details on the respective transformers.

### 2.3.1 Polyhedra Encoding

For a given polyhedron, NewPolka and PPL store both, the constraint set  $\mathcal{C}$  and the generator set  $\mathcal{G}$ , each represented as a matrix. We follow a similar approach adapted to our partitioned scenario. Specifically, assume a polyhedron  $P$  with permissible partition  $\bar{\pi}_P = \{\mathcal{X}_1, \mathcal{X}_2, \dots, \mathcal{X}_r\}$ , i.e., associated factors  $\{P_1, P_2, \dots, P_r\}$ , where  $P_k = (\mathcal{C}_{P_k}, \mathcal{G}_{P_k})$ . The blocks of  $\bar{\pi}_P$  are stored as a linked list of variables and the polyhedron as a linked list of factors. Alternatively, trees can also be used. Each factor is stored as two matrices. We do not explicitly store the factors and the blocks for the unconstrained variables. For example,  $\top$  is stored as  $\emptyset$ .

Table 2.2: Asymptotic time complexity of Polyhedra operators with decomposition.

Operator	Decomposed
Inclusion ( $\sqsubseteq$ )	$O(\sum_{i=1}^r n_i \cdot g_i \cdot m_i)$
Join ( $\sqcup$ )	$O(\sum_{i=1}^r n_i \cdot g_i \cdot m_i + n_{\max} \cdot g_{\max})$
Meet ( $\sqcap$ )	$O(\sum_{i=1}^r n_i \cdot m_i)$
Widening ( $\nabla$ )	$O(\sum_{i=1}^r n_i \cdot g_i \cdot m_i)$
Conditional	$O(n_{\max})$
Assignment	$O(n_{\max} \cdot g_{\max})$

### 2.3.2 Transformers and Permissible Partitions

The results in Section 2.2.2 calculated for each input polyhedra  $P, Q$  with partitions  $\pi_P, \pi_Q$  either the best (finest) or a permissible partition of the output polyhedron  $O$  of a transformer. Inspection shows that each result can be adapted to the case where the input partitions are only permissible. In this case, the output partition is likewise only permissible.

**Lemma 2.3.1.** Given permissible input partitions  $\bar{\pi}_P$  and  $\bar{\pi}_Q$ , Lemmas 2.2.1–2.2.4 and Theorem 2.2.5 yield permissible partitions for the outputs of transformers. Specifically, using prior notation:

- i) *Meet*:  $\bar{\pi}_{P \sqcap Q} = \bar{\pi}_P \sqcap \bar{\pi}_Q$  is permissible if  $P \sqcap Q \neq \perp$ , otherwise  $\perp$  is permissible.
- ii) *Conditional*:  $\bar{\pi}_P \uparrow \mathcal{B}$  is permissible if  $O \neq \perp$ , otherwise  $\perp$  is permissible.
- iii) *Assignment*:  $\bar{\pi}_P \uparrow \mathcal{B}$  is permissible.
- iv) *Widening*:  $\bar{\pi}_{P \nabla Q} = \bar{\pi}_Q$  is permissible.
- v) *Join*: Let  $\bar{\pi} = \bar{\pi}_P \sqcup \bar{\pi}_Q$  and  $\mathcal{U} = \{\mathcal{X}_k \mid P_k = Q_k, \mathcal{X}_k \in \bar{\pi}\}$ . Then the following is permissible:

$$\bar{\pi}_{P \sqcup Q} = \mathcal{U} \cup \bigcup_{\mathcal{A} \in \bar{\pi} \setminus \mathcal{U}} \mathcal{A}$$

Table 2.2 shows the asymptotic time complexity of the Polyhedra transformers decomposed with our approach. For simplicity, we assume that for binary transformers both inputs have the same partition. In the table,  $r$  is the number of blocks in the partition,  $n_i$  is the number of variables in the  $i$ -th block,  $g_i$  and  $m_i$  are the number of generators and constraints in the  $i$ -th factor respectively. It holds that  $n = \sum_{i=1}^r n_i$ ,  $m = \sum_{i=1}^r m_i$  and  $g = \prod_{i=1}^r g_i$ . We denote the number of variables and generators in the largest block by  $n_{\max}$  and  $g_{\max}$ , respectively. Since we follow the eager approach for conversion, both representations are available for inputs, i.e., the second column of Table 2.2 corresponds to column *Both* in Table 2.1. We do not show the cost of conversion.

```

 $\mathcal{P}_1 : \top$ 
  x:=5;
 $\mathcal{P}_2 : \{\{x = 5\}\}$ 
  u:=3;
 $\mathcal{P}_3 : \{\{x = 5\}, \{u = 3\}\}$ 
  if(x==y){
 $\mathcal{P}_4 : \{\{x = 5, x = y\}, \{u = 3\}\}$ 
    x:=2 · y;
 $\mathcal{P}_5 : \{\{y = 5, x = 2 \cdot y\}, \{u = 3\}\}$ 
  }
 $\mathcal{P}_6 : \{\{-x \leq -5, x \leq 10\}, \{u = 3\}\}$ 
  if(u==v){
 $\mathcal{P}_7 : \{\{-x \leq -5, x \leq 10\}, \{u = 3, u = v\}\}$ 
    u :=3 · v;
 $\mathcal{P}_8 : \{\{-x \leq -5, x \leq 10\}, \{v = 3, u = 3 \cdot v\}\}$ 
  }
 $\mathcal{P}_9 : \{\{-x \leq -5, x \leq 10\}, \{-u \leq -3, u \leq 9\}\}$ 
  z:=x + u;
 $\mathcal{P}_{10} : \{\{-x \leq -5, x \leq 10, -u \leq -3, u \leq 9, z = x + u\}\}$ 

```

Figure 2.5: Example of complexity reduction through decomposition for Polyhedra analysis on an example program.

Fig. 2.5 shows a representative program annotated with Polyhedra invariants at each program point. The program contains five variables  $u, v, x, y, z$  and has two conditional if-statements. It can be seen that the Polyhedra at different program points can be decomposed and thus the Polyhedra transformers benefit from the complexity reduction. For example, the assignment transformer for  $x:=2y$  and the conditional transformer for  $x==y$  need to operate only on the factor corresponding to the block  $\{x, y\}$ . The assignment transformer for  $u:=3v$  and the conditional transformer for  $u==v$  benefit similarly. Also note that the two blocks for the if statements modify only the variables  $\{x, y\}$  and  $\{u, v\}$  respectively. Thus the factors corresponding to the remaining variables remain equal in the inputs to the corresponding join. As a result, both join outputs  $\mathcal{P}_6$  and  $\mathcal{P}_9$  are also decomposed. We next discuss the algorithms for the core transformers using partitions.

## 2.4 POLYHEDRA TRANSFORMERS

In this section, we describe our algorithms for the main Polyhedra transformers. For each transformer, we first describe the base algorithm, followed by our adaptation of that algorithm to use partitions. We also discuss useful code optimizations for our algorithms. We follow an eager approach for the conversion; thus, the inputs and the output have both  $\mathcal{C}$  and  $\mathcal{G}$  available. Our choice allows us to always apply the conversion incrementally for the expensive meet, conditional, and join transformers while with the lazy approach it is possible that this cannot be done. Join is the most challenging transformer to adapt with partitions as the partition for the output depends on the exact form of the inputs. Our algorithms rely on two auxiliary transformers, *conversion* and *refactoring*, which we describe first.

### 2.4.1 Auxiliary Transformers

We apply code optimizations to leverage sparsity in the conversion algorithm which makes our conversion faster. Refactoring is frequently required by our algorithms to make the inputs conform to the same partition.

**CONVERSION TRANSFORMER** An expensive step in Chernikova’s algorithm is the computation of a matrix-vector product which is needed at each iteration of the algorithm. We observed that the vector is usually sparse, i.e., it contains mostly zeros; thus, we need to consider only those entries in the matrix which can be multiplied with the non-zero entries in the vector. Therefore at the start of each iteration, we compute an index for the non-zero entries of the vector. The index is discarded at the end of the iteration. This code optimization significantly reduces the cost of conversion.

We also vectorized the matrix-vector product using the single intrustion, multiple data (SIMD) based AVX intrinsics; however, it does not provide as much speedup compared to leveraging sparsity with the index.

**REFACTORIZING** Let  $P$  and  $Q$  be defined over the same set of variables  $\mathcal{X} = \{x_1, x_2, \dots, x_n\}$ , and let  $\bar{\pi}_P = \{x_{P_1}, x_{P_2}, \dots, x_{P_p}\}$ ,  $\bar{\pi}_Q = \{x_{Q_1}, x_{Q_2}, \dots, x_{Q_q}\}$  be permissible partitions for  $P$  and  $Q$  respectively and  $\mathcal{B} \subseteq \mathcal{X}$ . Usually  $\bar{\pi}_P \neq \bar{\pi}_Q$ ; thus, an important step for the transformers such as meet, inclusion testing, widening and join is refactoring the inputs  $P$  and  $Q$  so that the factors correspond to the same partition  $\bar{\pi}$  which is simply the least upper bound  $\bar{\pi}_P \sqcup \bar{\pi}_Q$ .

Similarly, usually  $\mathcal{B} \notin \bar{\pi}_P$  for the conditional and the assignment transformers. Thus,  $P$  is refactored according to  $\bar{\pi} = \bar{\pi}_P \uparrow \mathcal{B}$ .

$P$  is refactored by merging all factors  $P_i$  whose corresponding blocks  $x_{P_i}$  are included inside the same block  $x_j$  of  $\bar{\pi}$ . The merging is performed using the  $\bowtie$



**Algorithm 2.1** Refactor  $P$  with partition  $\bar{\pi}_P$  according to  $\bar{\pi}$ 


---

```

1: function REFACTOR( $P, \bar{\pi}_P, \bar{\pi}$ )
2:   Parameters:
3:      $P \leftarrow \{P_1, P_2, \dots, P_p\}$ 
4:      $\bar{\pi}_P \leftarrow \{\mathcal{X}_{P_1}, \mathcal{X}_{P_2}, \dots, \mathcal{X}_{P_p}\}$ 
5:      $\bar{\pi} \leftarrow \{\mathcal{X}_1, \mathcal{X}_2, \dots, \mathcal{X}_r\}$ 
6:   for  $k \in \{1, 2, \dots, r\}$  do
7:      $P'_k := \top$ 
8:   end for
9:   for  $i \in \{1, 2, \dots, p\}$  do
10:     $k := j$ , s.t.,  $\mathcal{X}_{P_i} \subseteq \mathcal{X}_j, \mathcal{X}_j \in \bar{\pi}$ 
11:     $P'_k := P'_k \bowtie P_i$ 
12:   end for
13:    $P' := \{P'_1, P'_2, \dots, P'_r\}$ 
14: return  $P'$ 
15: end function

```

---

transformer defined in (2.4). Refactoring is shown in Algorithm 2.1. We will use  $r$  to denote the number of blocks in  $\bar{\pi}$ .

**Example 2.4.1.** Consider<sup>5</sup>:

$$\begin{aligned}
\mathcal{X} &= \{x_1, x_2, x_3, x_4, x_5, x_6\}, \\
P &= \{\{x_1 = x_2, x_2 = 2\}, \{x_3 \leq 2\}, \{x_5 = 1\}, \{x_6 = 2\}\}, \\
Q &= \{\{x_1 = 2, x_2 = 2\}, \{x_3 \leq 2\}, \{x_5 = 2\}, \{x_6 = 3\}\}, \text{ with} \\
\bar{\pi}_P &= \{\{x_1, x_2\}, \{x_3, x_4\}, \{x_5\}, \{x_6\}\} \text{ and} \\
\bar{\pi}_Q &= \{\{x_1, x_2, x_4\}, \{x_3\}, \{x_5\}, \{x_6\}\}.
\end{aligned}$$

In this case,  $\bar{\pi}$  is:

$$\bar{\pi} = \bar{\pi}_P \sqcup \bar{\pi}_Q = \{\{x_1, x_2, x_3, x_4\}, \{x_5\}, \{x_6\}\}.$$

We find that both blocks  $\bar{\pi}_{P_1} = \{x_1, x_2\}$  and  $\bar{\pi}_{P_2} = \{x_3, x_4\}$  of  $\bar{\pi}_P$  are included in the first block of  $\bar{\pi}_P \sqcup \bar{\pi}_Q$ ; thus,  $P_1$  and  $P_2$  are merged using the  $\bowtie$  transformer. We merge  $Q_1$  and  $Q_2$  similarly. The resulting  $P'$  and  $Q'$  are shown below:

$$\begin{aligned}
P' &= \{P_1 \bowtie P_2, \{x_5 = 1\}, \{x_6 = 2\}\} \text{ and} \\
Q' &= \{Q_1 \bowtie Q_2, \{x_5 = 2\}, \{x_6 = 3\}\}
\end{aligned}$$

where,

$$\begin{aligned}
P_1 \bowtie P_2 &= \{x_1 = x_2, x_2 = 2, x_3 \leq 2\} \text{ and} \\
Q_1 \bowtie Q_2 &= \{x_1 = 2, x_2 = 2, x_3 \leq 2\}
\end{aligned}$$

After explaining refactoring, we now present our algorithms for the Polyhedra transformers with partitions.

---

<sup>5</sup> We show only constraints for simplicity.

### 2.4.2 Meet ( $\sqcap$ )

For the double representation, the constraint set  $\mathcal{C}_O$  of the output  $O = P \sqcap Q$  is the union of the constraints of the inputs  $P$  and  $Q$ , i.e.,  $\mathcal{C}_{P \sqcap Q} = \mathcal{C}_P \cup \mathcal{C}_Q$ .  $\mathcal{G}_O$  is obtained by incrementally adding the constraints in  $\mathcal{C}_Q$  to the polyhedron defined by  $\mathcal{G}_P$  through the conversion transformer. If the conversion returns  $\mathcal{G}_O = \emptyset$ , then  $\mathcal{C}_O$  is unsatisfiable and thus  $O = \perp$ .

**MEET WITH PARTITIONS** Our algorithm first computes the common partition  $\bar{\pi}_P \sqcap \bar{\pi}_Q$ .  $P$  and  $Q$  are then refactored according to this partition using Algorithm 2.1 to obtain  $P'$  and  $Q'$ . If  $P'_k = Q'_k$ , then  $\mathcal{C}_{P'_k} \cup \mathcal{C}_{Q'_k} = \mathcal{C}_{P'_k}$  and no conversion is required and we simply add  $P'_k$  to  $O$ . If  $P'_k \neq Q'_k$  we add  $\mathcal{C}_{P'_k} \cup \mathcal{C}_{Q'_k}$  to  $\mathcal{C}_O$ . Next the constraints in  $\mathcal{C}_{Q'_k}$  are incrementally added to the polyhedron defined by  $\mathcal{G}_{P'_k}$  through the conversion transformer obtaining  $\mathcal{G}_{P'_k \sqcap Q'_k}$ . If the conversion algorithm returns  $\mathcal{G}_{P'_k \sqcap Q'_k} = \emptyset$ , then we set  $O = \perp$ . We know from Section 2.3 that  $\bar{\pi}_O = \bar{\pi}_P \sqcap \bar{\pi}_Q$  if  $O \neq \perp$ , otherwise  $\bar{\pi}_O = \perp$ .

**CODE OPTIMIZATION**  $\mathcal{C}_{P'_k}$  and  $\mathcal{C}_{Q'_k}$  usually contain a number of common constraints. The generators in  $\mathcal{G}_{P'_k}$  already correspond to the constraints that occur in both  $\mathcal{C}_{P'_k}$  and  $\mathcal{C}_{Q'_k}$ . Thus, these constraints need not be considered for the conversion which reduces its cost.

The check for common constraints can create an overhead as in the worst case we have to compare each vector in  $\mathcal{C}_{Q'_k}$  with all vectors in  $\mathcal{C}_{P'_k}$ . To reduce this overhead, for a given vector in  $\mathcal{C}_{Q'_k}$ , we keep track of the vector index which caused the equality check to fail for the previous vector in  $\mathcal{C}_{P'_k}$ . For the next vector in  $\mathcal{C}_{P'_k}$ , we first compare the vector values at this index as the next vector, if not equal, is also likely to fail this check. The pseudo code for our meet transformer is shown in Algorithm 2.2. We omit tracking of the vector index in Algorithm 2.2 for simplicity.

### 2.4.3 Inclusion ( $\sqsubseteq$ )

For the double representation,  $P \sqsubseteq Q$  holds if all generators in  $\mathcal{G}_P$  satisfy all constraints in  $\mathcal{C}_Q$ . A vertex  $v \in \mathcal{V}_P$  satisfies the constraint set  $\mathcal{C}_Q$  if  $A \cdot v \leq b$  and  $D \cdot v = e$ . A ray  $r \in \mathcal{R}_P$  satisfies  $\mathcal{C}_Q$  if  $A \cdot r \leq 0$  and  $D \cdot r = 0$ . A line  $z \in \mathcal{Z}_P$  satisfies  $\mathcal{C}_Q$  if  $A \cdot z = 0$  and  $D \cdot z = 0$ .

**INCLUSION TESTING WITH PARTITIONS** In our algorithm, we refactor  $P$  and  $Q$  according to the same partition  $\bar{\pi}_P \sqcap \bar{\pi}_Q$ . We only refactor the generators of  $P$  and the constraints of  $Q$  according to  $\bar{\pi}_P \sqcap \bar{\pi}_Q$ , obtaining  $\mathcal{G}_{P'}$  and  $\mathcal{C}_{Q'}$  respectively. We then check for each block  $\mathcal{X}_k$  in  $\bar{\pi}_P \sqcap \bar{\pi}_Q$  if all generators in  $\mathcal{G}_{P'_k}$  satisfy  $\mathcal{C}_{Q'_k}$ .

**Algorithm 2.2** Decomposed Polyhedra meet

---

```

1: function MEET( $P, Q, \bar{\pi}_P, \bar{\pi}_Q$ )
2:   Parameters:
3:      $P \leftarrow \{P_1, P_2, \dots, P_p\}$ 
4:      $Q \leftarrow \{Q_1, Q_2, \dots, Q_q\}$ 
5:      $\bar{\pi}_P \leftarrow \{\mathcal{X}_{P_1}, \mathcal{X}_{P_2}, \dots, \mathcal{X}_{P_p}\}$ 
6:      $\bar{\pi}_Q \leftarrow \{\mathcal{X}_{Q_1}, \mathcal{X}_{Q_2}, \dots, \mathcal{X}_{Q_q}\}$ 
7:      $P' := \text{refactor}(P, \bar{\pi}_P, \bar{\pi}_P \sqcup \bar{\pi}_Q)$ 
8:      $Q' := \text{refactor}(Q, \bar{\pi}_Q, \bar{\pi}_P \sqcup \bar{\pi}_Q)$ 
9:      $O = \emptyset$ 
10:    for  $k \in \{1, 2, \dots, r\}$  do
11:      if  $P'_k = Q'_k$  then
12:         $O.\text{add}(P'_k)$ 
13:      else
14:         $\mathcal{C} := \text{remove\_common\_con}(\mathcal{C}_{P'_k} \cup \mathcal{C}_{Q'_k})$ 
15:         $\mathcal{G} := \text{incr\_chernikova}(\mathcal{C}, \mathcal{C}_{P'_k}, \mathcal{G}_{P'_k})$ 
16:        if  $\mathcal{G} = \emptyset$  then
17:           $O := \perp$ 
18:           $\bar{\pi}_O := \perp$ 
19:          return  $(O, \bar{\pi}_O)$ 
20:        end if
21:         $O.\text{add}((\mathcal{C}, \mathcal{G}))$ 
22:      end if
23:    end for
24:     $\bar{\pi}_O := \bar{\pi}_P \sqcup \bar{\pi}_Q$ 
25:    return  $(O, \bar{\pi}_O)$ 
26: end function

```

---

**CODE OPTIMIZATION** The result of the inclusion testing transformer is usually negative, so we first check the smaller factors for inclusion. Thus, the factors are sorted in the order given by the product of the number of generators in  $\mathcal{G}_{P'_k}$  and the number of constraints in  $\mathcal{C}_{Q'_k}$ . The pseudo code for our inclusion testing transformer is shown in Algorithm 2.3.

2.4.4 *Conditional*

For the double representation, the transformer for the conditional statement  $\alpha \cdot x_i \otimes \delta$  adds the constraint  $c = (\alpha - a_i) \cdot x_i \otimes \delta - a_i \cdot x_i$  to the constraint set  $\mathcal{C}_P$ , producing  $\mathcal{C}_O$ .  $\mathcal{G}_O$  is obtained by incrementally adding the constraint  $c$  to the polyhedron defined by  $\mathcal{G}_P$  through the conversion transformer. The conversion returns  $\mathcal{G}_O = \emptyset$ , if  $\mathcal{C}_O$  is unsatisfiable and thus we get  $O = \perp$ .

**CONDITIONAL TRANSFORMER WITH PARTITIONS** Our algorithm refactors  $P$  according to  $\bar{\pi}_P \uparrow \mathcal{B}$ , producing  $P'$ . The constraint  $c$  is added to the constraint set  $\mathcal{C}_{P'_k}$  of the factor corresponding to the block  $\mathcal{X}_k \in \bar{\pi}_P \uparrow \mathcal{B}$  containing  $\mathcal{B}$ , producing  $\mathcal{C}_{O_k}$ .  $\mathcal{G}_{O_k}$  is obtained by incrementally adding the constraint  $c$  to the polyhedron

**Algorithm 2.3** Decomposed inclusion testing for Polyhedra

---

```

1: function INCLUSION( $P, Q, \bar{\pi}_P, \bar{\pi}_Q$ )
2:   Parameters:
3:      $P \leftarrow \{P_1, P_2, \dots, P_p\}$ 
4:      $Q \leftarrow \{Q_1, Q_2, \dots, Q_q\}$ 
5:      $\bar{\pi}_P \leftarrow \{x_{P_1}, x_{P_2}, \dots, x_{P_p}\}$ 
6:      $\bar{\pi}_Q \leftarrow \{x_{Q_1}, x_{Q_2}, \dots, x_{Q_q}\}$ 
7:      $\mathcal{G}_{P'} := \text{refactor\_gen}(\mathcal{G}_P, \bar{\pi}_P, \bar{\pi}_P \sqcup \bar{\pi}_Q)$ 
8:      $\mathcal{C}_{Q'} := \text{refactor\_con}(\mathcal{C}_Q, \bar{\pi}_Q, \bar{\pi}_P \sqcup \bar{\pi}_Q)$ 
9:      $\text{sort\_by\_size}(\mathcal{G}_{P'}, \mathcal{C}_{Q'})$ 
10:    for  $k \in \{1, 2, \dots, r\}$  do
11:      if  $P'_k \not\subseteq Q'_k$  then
12:        return false
13:      end if
14:    end for
15:  return true
16: end function

```

---

**Algorithm 2.4** Decomposed conditional transformer for Polyhedra

---

```

1: function CONDITIONAL( $P, \bar{\pi}_P, \text{stmt}$ )
2:   Parameters:
3:      $P \leftarrow \{P_1, P_2, \dots, P_p\}$ 
4:      $\bar{\pi}_P \leftarrow \{x_{P_1}, x_{P_2}, \dots, x_{P_p}\}$ 
5:      $\text{stmt} \leftarrow \alpha \cdot x_i \otimes \delta$ 
6:      $\mathcal{B} := \text{extract\_block}(\text{stmt})$ 
7:      $P' := \text{refactor}(P, \bar{\pi}_P, \bar{\pi}_P \uparrow \mathcal{B})$ 
8:      $O := \emptyset$ 
9:      $\bar{\pi}_O := \bar{\pi}_P \uparrow \mathcal{B}$ 
10:    for  $k \in \{1, 2, \dots, r\}$  do
11:      if  $\mathcal{B} \subseteq \bar{\pi}_{O_k}$  then
12:         $\mathcal{C} := \mathcal{C}_{P'_k} \cup \{(\alpha - a_i) \cdot x_i \otimes \delta - a_i \cdot x_i\}$ 
13:         $\mathcal{G} := \text{incr\_chernikova}(\mathcal{C}, \mathcal{C}_{P'_k}, \mathcal{G}_{P'_k})$ 
14:        if  $\mathcal{G} = \emptyset$  then
15:           $O := \perp$ 
16:           $\bar{\pi}_O := \perp$ 
17:          return  $(O, \bar{\pi}_O)$ 
18:        end if
19:         $O.\text{add}((\mathcal{C}, \mathcal{G}))$ 
20:      else
21:         $O.\text{add}(P'_k)$ 
22:      end if
23:    end for
24:  return  $(O, \bar{\pi}_O)$ 
25: end function

```

---

defined by  $\mathcal{G}_{P'_k}$ . If the conversion algorithm returns  $\mathcal{G}_{O_k} = \emptyset$ , then we set  $O = \perp$ . As shown in Section 2.3,  $\bar{\pi}_O = \bar{\pi}_P \uparrow \mathcal{B}$  if  $O \neq \perp$ , otherwise  $\bar{\pi}_O = \perp$ . The pseudo code for our conditional transformer is shown in Algorithm 2.4.

### 2.4.5 Assignment

In Section 2.1.2, the transformer for the assignment  $x_i := \delta$ , where  $\delta = a^T \cdot x + \epsilon$ , was defined using the constraint set  $\mathcal{C}_P$  of  $P$ . For the double representation, the transformer works on the generator set  $\mathcal{G}_P = \{\mathcal{V}_P, \mathcal{R}_P, \mathcal{Z}_P\}$ . The generators  $\mathcal{G}_O = \{\mathcal{V}_O, \mathcal{R}_O, \mathcal{Z}_O\}$  for the output are given by:

$$\begin{aligned}\mathcal{V}_O &= \{v' \mid v'_i = a^T \cdot v + \epsilon, v \in \mathcal{V}_P\}, \\ \mathcal{R}_O &= \{r' \mid r'_i = a^T \cdot r, r \in \mathcal{R}_P\}, \\ \mathcal{Z}_O &= \{z' \mid z'_i = a^T \cdot z, z \in \mathcal{Z}_P\}.\end{aligned}\tag{2.10}$$

If the assignment is invertible, i.e., if  $a_i \neq 0$  (for example  $x := x + 1$ ), the constraint set  $\mathcal{C}_O$  can be calculated by backsubstitution. Let  $x'_i$  be the new value of  $x_i$  after assignment, then  $x'_i = a^T \cdot x + \epsilon$ . Thus, putting  $x_i = (x'_i - \sum_{j \neq i} a_j \cdot x_j - \epsilon) / a_i$  for  $x_i$  in all constraints of the set  $\mathcal{C}_P = \{A \cdot x \leq b, D \cdot x = e\}$  and renaming  $x'_i$  to  $x_i$ , we get the constraint set  $\mathcal{C}_O$ . For the non-invertible assignments, the conversion transformer is applied on all generators in  $\mathcal{G}_O$ .

**ASSIGNMENT TRANSFORMER WITH PARTITIONS** In our algorithm, we refactor  $P$  according to  $\bar{\pi}_P \uparrow \mathcal{B}$ , producing  $P'$ . We compute the new generators using (2.10) only for the factor  $P'_k$  corresponding to the block  $\mathcal{X}_k \in \bar{\pi}_P \uparrow \mathcal{B}$  containing  $\mathcal{B}$ . The constraints are computed only for  $P'_k$  for both invertible and non-invertible assignments. This results in a large reduction of the operation count. As shown in Section 2.3,  $\bar{\pi}_O = \bar{\pi}_P \uparrow \mathcal{B}$ . We will present a refinement of  $\bar{\pi}_O$  in Chapter 3. The pseudo code for our assignment transformer is shown in Algorithm 2.5. The `handle_assign` function applies (2.10) on  $\mathcal{G}_{P'_k}$ .

### 2.4.6 Widening ( $\nabla$ )

For the double representation, the widening transformer requires the generators and the constraints of  $P$  and the constraints of  $Q$ . A given constraint  $a \cdot x \otimes b$ , where  $\otimes \in \{\leq, =\}$ , *saturates* a vertex  $v \in \mathcal{V}$  if  $a \cdot v = b$ , a ray  $r \in \mathcal{R}$  if  $a \cdot r = 0$ , and a line  $z \in \mathcal{Z}$  if  $a \cdot z = 0$ .

For a given constraint  $c$  and  $\mathcal{G}$ , the set  $\mathcal{S}_{c,\mathcal{G}}$  is defined as:

$$\mathcal{S}_{c,\mathcal{G}} = \{g \mid g \in \mathcal{G} \text{ and } c \text{ saturates } g\}.\tag{2.11}$$

The standard widening transformer computes for each constraint  $c_p \in \mathcal{C}_P$ , the set  $\mathcal{S}_{c_p,\mathcal{G}_P}$  and for each constraint  $c_q \in \mathcal{C}_Q$ , the set  $\mathcal{S}_{c_q,\mathcal{G}_P}$ . If  $\mathcal{S}_{c_q,\mathcal{G}_P} = \mathcal{S}_{c_p,\mathcal{G}_P}$  for any  $c_p$ , then  $c_q$  is added to the output constraint set  $\mathcal{C}_O$ . The widening transformer removes the constraints from  $\mathcal{C}_Q$ , so the conversion is not incremental in the standard implementations. Recent work [184] allows incremental conversion when constraints or generators are removed.

**Algorithm 2.5** Decomposed assignment transformer for Polyhedra

---

```

1: function ASSIGNMENT( $P, \bar{\pi}_P, \text{stmt}$ )
2:   Parameters:
3:      $P \leftarrow \{P_1, P_2, \dots, P_p\}$ 
4:      $\bar{\pi}_P \leftarrow \{\mathcal{X}_{P_1}, \mathcal{X}_{P_2}, \dots, \mathcal{X}_{P_p}\}$ 
5:      $\text{stmt} \leftarrow x_i := a^T \cdot x + e$ 
6:      $\mathcal{B} := \text{extract\_block}(\text{stmt})$ 
7:      $P' := \text{refactor}(P, \bar{\pi}_P, \bar{\pi}_P \uparrow \mathcal{B})$ 
8:      $O := \emptyset$ 
9:      $\bar{\pi}_O := \bar{\pi}_P \uparrow \mathcal{B}$ 
10:    for  $k \in \{1, 2, \dots, r\}$  do
11:      if  $\mathcal{B} \subseteq \bar{\pi}_{O_k}$  then
12:         $\mathcal{G} := \text{handle\_assign}(\mathcal{G}_{P'_k}, \text{stmt})$ 
13:        if  $a_i = 0$  then
14:           $\mathcal{C} := \text{backsubstitute}(\mathcal{G}, \text{stmt})$ 
15:        else
16:           $\mathcal{C} := \text{chernikova}(\mathcal{G})$ 
17:        end if
18:         $O.\text{add}((\mathcal{C}, \mathcal{G}))$ 
19:      else
20:         $O.\text{add}(P'_k)$ 
21:      end if
22:    end for
23:    return  $(O, \bar{\pi}_O)$ 
24: end function

```

---

**WIDENING WITH PARTITIONS** In our algorithm, we refactor  $P$  according to  $\bar{\pi}_P \sqcup \bar{\pi}_Q$ , producing  $P'$ . For a given constraint  $c_q \in \mathcal{C}_{Q_i}$ , we access the block  $\mathcal{X}_k \in \bar{\pi}_P \sqcup \bar{\pi}_Q$  containing  $\mathcal{X}_{Q_i}$  and compute  $\mathcal{S}_{c_q, \mathcal{G}_{P'_k}}$ . If this set is equal to  $\mathcal{S}_{c_p, \mathcal{G}_{P'_k}}$  for any  $c_p \in \mathcal{C}_{P'_k}$ , then  $c_q$  is added to  $\mathcal{C}_{O_i}$ . If  $\mathcal{C}_{O_i} = \mathcal{C}_{Q_i}$ , then the conversion transformer is not required and  $Q_i$  is added to  $O$ , otherwise it is applied on all constraints in  $\mathcal{C}_{O_i}$ . As shown in Section 2.3,  $\bar{\pi}_O = \bar{\pi}_Q$ . The pseudo code for our widening transformer is shown in Algorithm 2.6. The saturate function applies (2.11) on given  $c$  and  $\mathcal{G}$ .

To possibly improve the granularity for  $\bar{\pi}_O$ , we check if for any block  $\mathcal{X}_k \in \bar{\pi}_{P \nabla Q}$ ,  $\mathcal{C}_{O_k} = \emptyset$ ; if yes, then  $\mathcal{X}_k$  is removed from  $\bar{\pi}_{P \nabla Q}$  and replaced by a set of singleton blocks with each block corresponding to a variable in  $\mathcal{X}_k$ .

2.4.7 *Join* ( $\sqcup$ )

For the double representation, the generators  $\mathcal{G}_O$  of the output  $O = P \sqcup Q$  of the join are simply the union of the generators of the input polyhedra, i.e.,  $\mathcal{G}_O = \mathcal{G}_P \cup \mathcal{G}_Q$ .  $\mathcal{C}_O$  is obtained by incrementally adding the generators in  $\mathcal{G}_Q$  to the polyhedron defined by  $\mathcal{C}_P$ .

**Algorithm 2.6** Decomposed polyhedra widening

---

```

1: function WIDENING( $P, Q, \bar{\pi}_P, \bar{\pi}_Q$ )
2:   Parameters:
3:      $P \leftarrow \{P_1, P_2, \dots, P_p\}$ 
4:      $Q \leftarrow \{Q_1, Q_2, \dots, Q_q\}$ 
5:      $\bar{\pi}_P \leftarrow \{X_{P_1}, X_{P_2}, \dots, X_{P_p}\}$ 
6:      $\bar{\pi}_Q \leftarrow \{X_{Q_1}, X_{Q_2}, \dots, X_{Q_q}\}$ 
7:      $P' := \text{refactor}(P, \bar{\pi}_P, \bar{\pi}_P \sqcup \bar{\pi}_Q)$ 
8:      $O := \emptyset$ 
9:     for  $k \in \{1, 2, \dots, r\}$  do
10:      for  $c_p \in \mathcal{C}_{P'_k}$  do
11:         $\mathcal{S}_{c_p, \mathcal{G}_{P'_k}} := \text{saturate}(c_p, \mathcal{G}_{P'_k})$ 
12:      end for
13:    end for
14:    for  $i \in \{1, 2, \dots, q\}$  do
15:       $\mathcal{C}_{O_i} := \emptyset$ 
16:       $k := j$ , s.t.,  $X_{Q_i} \subseteq X_j, X_j \in \bar{\pi}_P \sqcup \bar{\pi}_Q$ 
17:      for  $c_q \in \mathcal{C}_{Q_i}$  do
18:         $\mathcal{S}_{c_q, \mathcal{G}_{P'_k}} := \text{saturate}(c_q, \mathcal{G}_{P'_k})$ 
19:        if  $\exists c_p \in \mathcal{C}_{P'_k}$ , s.t.,  $\mathcal{S}_{c_q, \mathcal{G}_{P'_k}} = \mathcal{S}_{c_p, \mathcal{G}_{P'_k}}$  then
20:           $\mathcal{C}_{O_i} := \mathcal{C}_{O_i} \cup \{c_q\}$ 
21:        end if
22:      end for
23:      if  $\mathcal{C}_{O_i} = \mathcal{C}_{Q_i}$  then
24:         $O.\text{add}(Q_i)$ 
25:      else
26:         $\mathcal{G}_{O_i} := \text{chernikova}(\mathcal{C}_{O_i})$ 
27:         $O.\text{add}((\mathcal{C}_{O_i}, \mathcal{G}_{O_i}))$ 
28:      end if
29:    end for
30:    return  $(O, \bar{\pi}_Q)$ 
31: end function

```

---

**JOIN WITH PARTITIONS** In our join transformer shown in Algorithm 2.8, we first refactor  $P$  and  $Q$  according to  $\bar{\pi}_P \sqcup \bar{\pi}_Q$ , obtaining  $P'$  and  $Q'$  respectively. The join transformer can create constraints between the variables in different blocks of  $\bar{\pi}_P \sqcup \bar{\pi}_Q$ . In the worst case, the join can merge all blocks into one to produce the  $\top$  partition, which blows up the number of generators due to the Cartesian product in (2.4). However, in many cases common in the program analysis setting, the blocks of  $\bar{\pi}_P \sqcup \bar{\pi}_Q$  need not be combined without sacrificing precision. Identifying such cases is key in our work for avoiding the exponential blowup observed by prior libraries [12, 104]. Theorem 2.2.5 identifies such cases.

**COMPUTING THE GENERATORS FOR THE JOIN** If  $P'_k = Q'_k$  holds, then  $P'_k$  can be added to  $O$  by Corollary 2.2.1. Since no new generators are added, the conver-

**Algorithm 2.7** Compute generators for the join

---

```

1: function COMPUTE_GEN_JOIN( $P', Q', \bar{\pi}_P \sqcup \bar{\pi}_Q$ )
2:   Parameters:
3:      $P' \leftarrow \{P'_1, P'_2, \dots, P'_r\}$ 
4:      $Q' \leftarrow \{Q'_1, Q'_2, \dots, Q'_r\}$ 
5:      $\bar{\pi}_P \sqcup \bar{\pi}_Q \leftarrow \{x_1, x_2, \dots, x_r\}$ 
6:      $\mathcal{U} := \emptyset$ 
7:      $\bar{\pi}_O := \emptyset$ 
8:      $P'_N := Q'_N := \top$ 
9:      $O := \emptyset$ 
10:    for  $k \in \{1, 2, \dots, r\}$  do
11:      if  $P'_k = Q'_k$  then
12:         $\mathcal{U}.add(x_k)$ 
13:         $O.add(P'_k)$ 
14:      else
15:         $\bar{\pi}_O := \bar{\pi}_O \cup x_k$ 
16:         $P'_N := P'_N \bowtie P'_k$ 
17:         $Q'_N := Q'_N \bowtie Q'_k$ 
18:      end if
19:    end for
20:     $\bar{\pi}_O := \mathcal{U} \cup \bar{\pi}_O$ 
21:    return  $\bar{\pi}_O, O, P'_N, Q'_N$ 
22:  end function

```

---

sion transformer is not required for these. This results in a large reduction of the operation count for the conversion transformer.

As in Section 2.3, we compute  $\bar{\pi} = \bar{\pi}_P \sqcup \bar{\pi}_Q$  and  $\mathcal{U} = \{x_k \in \bar{\pi} \mid P'_k = Q'_k\}$ . The factors in  $P'$  and  $Q'$  corresponding to the blocks  $\mathcal{A} \in \bar{\pi} \setminus \mathcal{U}$  are merged using the  $\bowtie$  transformer to produce  $P'_N$  and  $Q'_N$  respectively. Next, we compute  $\mathcal{G}_O = \{\mathcal{G}_{P'_{u_1}}, \mathcal{G}_{P'_{u_2}}, \dots, \mathcal{G}_{P'_{u_u}}, \mathcal{G}_{P'_N} \cup \mathcal{G}_{Q'_N}\}$  where  $u = |\mathcal{U}|$ . The pseudo code for this step is shown in Algorithm 2.7.

**COMPUTING THE CONSTRAINTS FOR THE JOIN** We know the constraint set for all factors corresponding to the blocks in  $\mathcal{U}$ .  $\mathcal{C}_{P'_N \cup Q'_N}$  is obtained by incrementally adding the generators in  $\mathcal{G}_{Q'_N}$  to the polyhedron defined by  $\mathcal{C}_{P'_N}$ . Similar to the meet transformer in Section 2.4.2, we apply our code optimization of not computing the constraints for the generators common in both  $\mathcal{G}_{P'_N}$  and  $\mathcal{G}_{Q'_N}$ .

**Example 2.4.2.** Consider

$$\begin{aligned}
\mathcal{X} &= \{x_1, x_2, x_3, x_4, x_5, x_6\}, \\
P &= \{\{x_1 = x_2, x_2 = 2\}, \{x_3 \leq 2\}, \{x_5 = 1\}, \{x_6 = 2\}\}, \\
Q &= \{\{x_1 = 2, x_2 = 2\}, \{x_3 \leq 2\}, \{x_5 = 2\}, \{x_6 = 3\}\} \text{ with} \\
\bar{\pi}_P &= \{\{x_1, x_2\}, \{x_3, x_4\}, \{x_5\}, \{x_6\}\} \text{ and} \\
\bar{\pi}_Q &= \{\{x_1, x_2, x_4\}, \{x_3\}, \{x_5\}, \{x_6\}\}
\end{aligned}$$



**Algorithm 2.8** Decomposed polyhedra join

---

```

1: function JOIN( $P, Q, \bar{\pi}_P, \bar{\pi}_Q$ )
2:   Parameters:
3:      $P \leftarrow \{P_1, P_2, \dots, P_p\}$ 
4:      $Q \leftarrow \{Q_1, Q_2, \dots, Q_q\}$ 
5:      $\bar{\pi}_P \leftarrow \{x_{P_1}, x_{P_2}, \dots, x_{P_p}\}$ 
6:      $\bar{\pi}_Q \leftarrow \{x_{Q_1}, x_{Q_2}, \dots, x_{Q_q}\}$ 
7:      $P' := \text{refactor}(P, \bar{\pi}_P, \bar{\pi}_P \sqcup \bar{\pi}_Q)$ 
8:      $Q' := \text{refactor}(Q, \bar{\pi}_Q, \bar{\pi}_P \sqcup \bar{\pi}_Q)$ 
9:      $(\bar{\pi}_O, O, P'_N, Q'_N) := \text{compute\_gen\_join}(P', Q', \bar{\pi}_P \sqcup \bar{\pi}_Q)$ 
10:     $\mathcal{G} := \text{remove\_common\_gen}(\mathcal{G}_{P'_N} \cup \mathcal{G}_{Q'_N})$ 
11:     $\mathcal{C} := \text{incr\_chernikova}(\mathcal{G}, \mathcal{G}_{P'_N}, \mathcal{C}_{P'_N})$ 
12:     $O.\text{add}((\mathcal{C}, \mathcal{G}))$ 
13:    return  $(O, \bar{\pi}_O)$ 
14: end function

```

---

In this case, the refactoring gives us,

$$\begin{aligned} \bar{\pi}_P \sqcup \bar{\pi}_Q &= \{\{x_1, x_2, x_3, x_4\}, \{x_5\}, \{x_6\}\}, \\ P' &= \{\{x_1 = x_2, x_2 = 2, x_3 \leq 2\}, \{x_5 = 1\}, \{x_6 = 2\}\}, \\ Q' &= \{\{x_1 = 2, x_2 = 2, x_3 \leq 2\}, \{x_5 = 2\}, \{x_6 = 3\}\}. \end{aligned}$$

We observe that only  $P'_1 = Q'_1$ ; thus, we add  $P'_1$  to the join  $O$  and  $\{x_1, x_2, x_3, x_4\}$  to  $\mathcal{U}$ . Applying Algorithm 2.7 we get,

$$\begin{aligned} \mathcal{N} &= \{\{x_5\}, \{x_6\}\}, \\ P'_N &= \{x_5 = 1, x_6 = 2\}, \\ Q'_N &= \{x_5 = 2, x_6 = 3\}, \\ O &= \{\{x_1 = x_2, x_2 = 2, x_3 \leq 2\}\}, \\ \bar{\pi}_O &= \{\{x_1, x_2, x_3, x_4\}, \{x_5, x_6\}\}. \end{aligned}$$

$\mathcal{G}_{Q'_N}$  contains only one vertex  $(2, 3)$ . The conversion transformer incrementally adds this vertex to the polyhedron defined by  $\mathcal{C}_{P'_N}$ . Thus, the factors  $O_1$  and  $O_2$  of  $O = \{O_1, O_2\}$  are given by,

$$\begin{aligned} O_1 &= \{x_1 = x_2, x_2 = 2, x_3 \leq 2\} \text{ and} \\ O_2 &= \{-x_5 \leq -1, x_5 \leq 2, x_6 = x_5 + 1\}. \end{aligned}$$

As shown in Section 2.3,  $\bar{\pi}_{P \sqcup Q} = \mathcal{U} \cup \bigcup_{A \in \bar{\pi} \setminus \mathcal{U}} A$ . Note that we can have  $\pi_O \neq \bar{\pi}_O$  even though  $\bar{\pi}_P = \pi_P$  and  $\bar{\pi}_Q = \pi_Q$ . This is because the join transformer will not have a constraint involving a variable  $x_i$  if either  $P$  or  $Q$  does not contain any constraint involving  $x_i$ . We illustrate this with an example below:

**Example 2.4.3.** Consider

$$\begin{aligned} P &= \{\{x_1 = 0\}, \{x_2 - x_3 = 2, x_3 - x_4 = 3\}\} \text{ and} \\ Q &= \{\{x_1 = 0\}, \{x_2 - x_4 = 5\}\} \text{ with} \\ \bar{\pi}_P &= \pi_P = \{\{x_1\}, \{x_2, x_3, x_4\}\} \text{ and} \\ \bar{\pi}_Q &= \pi_Q = \{\{x_1\}, \{x_2, x_4\}, \{x_3\}\}. \end{aligned}$$

For this example, Algorithm 2.8 returns

$$\begin{aligned} O &= \{\{x_1 = 0\}, \{x_2 - x_4 = 5\}\} \text{ and} \\ \bar{\pi}_O &= \{\{x_1\}, \{x_2, x_3, x_4\}\}. \end{aligned}$$

whereas  $\pi_O = \{\{x_1\}, \{x_2, x_4\}, \{x_3\}\}$ . Thus  $\pi_O \sqsubseteq \bar{\pi}_O$ .

**IMPROVING THE GRANULARITY OF  $\bar{\pi}_O$**  We lose performance since  $\bar{\pi}_O$  is usually not the finest partition for  $O$ . To possibly improve the partition obtained, we perform a preprocessing step before applying Algorithm 2.7 in our join transformer. If all variables of a block  $\mathcal{X}_k \in \bar{\pi}_P \sqcup \bar{\pi}_Q$  are unconstrained in either  $P$  or  $Q$ , then the join does not require any constraints involving these variables. We replace  $\mathcal{X}_k$  in  $\bar{\pi}_P \sqcup \bar{\pi}_Q$  with a set of singleton blocks. This set has one block for each variable in  $\mathcal{X}_k$ .  $P'_k$  and  $Q'_k$  are not considered for the join.

If only a subset of variables of  $\mathcal{X}_k \in \bar{\pi}_P \sqcup \bar{\pi}_Q$  are unconstrained in either  $P$  or  $Q$ , then we cannot remove the unconstrained variables from  $\mathcal{X}_k$  as the join may require constraints involving the unconstrained variables. For example  $x_3$  is unconstrained in  $Q$  in example 2.4.3. However, the constraints involving  $x_3$  are required for the join or else we lose precision. In Chapter 3 we will present a refinement which will produce a finer output partition for the join.

It is important to note that the key to keeping the cost of the join down is to reduce the application of the  $\bowtie$  transformer as it increases the number of generators exponentially, which in turn, increases the cost of the expensive conversion. The  $\bowtie$  transformer is applied in Algorithm 2.8 during refactoring and while merging factors corresponding to  $\mathcal{A}$ . In practice,  $\bar{\pi}_P$  and  $\bar{\pi}_Q$  are usually similar so the  $\bowtie$  transformer adds a small number of generators while refactoring.

**WHY THEOREM 2.2.5 WORKS IN PRACTICE** In the program analysis setting, the join is applied at the loop head or where the branches corresponding to if-else statements merge. In case of a loop head,  $P$  represents the polyhedron before executing the loop body and  $Q$  represents the polyhedron after executing the loop. The loop usually modifies only a small number of variables. The factors corresponding to the blocks containing only the unmodified variables are equal, thus  $|\bigcup_{\mathcal{A} \in \bar{\pi} \setminus \mathcal{U}} \mathcal{A}|$  is small. Hence, the application of the  $\bowtie$  transformer while merging factors corresponding to  $\mathcal{A}$  does not create an exponential number of new generators. Similarly for the if-else, the branches modify only a small number of variables and thus  $|\bigcup_{\mathcal{A} \in \bar{\pi} \setminus \mathcal{U}} \mathcal{A}|$  remains small.

```

x:=0;
y:=0;
if (*){
    x++;
    y++;
}
z:=x;

```

Figure 2.6: Precision loss for static partitioning.

**COMPARISON WITH STATIC PARTITIONING** It is also worth noting that determining unmodified blocks before running the analysis requires knowledge of the partition at the start of the loop. Partitions computed based on the dependence relation between program variables may not be permissible as the abstract semantics of the Polyhedra transformers may relate more variables, resulting in precision loss. This is illustrated by the code in Fig. 2.6.

Here an analysis based on the dependence relation [27] will yield the partition  $\{\{x, z\}, \{y\}\}$  after the assignment  $z:=x$ , since the variables  $x$  and  $y$  are unrelated. However, the join due to the conditional if-statement creates a constraint between  $x$  and  $y$ ; thus,  $\pi_p = \{\{x, y, z\}\}$  which is computed by our analysis.

**COMPLEXITY** The performance of the join transformer is dominated by the cost of the conversion transformer. The conversion incrementally adds the generators corresponding to  $\mathcal{G}_Q$  to the polyhedron defined by the constraints in  $\mathcal{C}_p$ . The worst case complexity of the conversion is exponential in the number of generators. For the join without partitioning, the number of generators can be  $O(2^n)$  in the worst case. The join transformer in Algorithm 2.8 applies the conversion only on the generators in  $\mathcal{G}_{Q'_N}$ . Using the notation from Section 2.3, let  $\mathcal{N} = \bigcup_{\mathcal{A} \in \bar{\pi} \setminus \mathcal{U}} \mathcal{A}$  be the union of all blocks in  $\bar{\pi}$  for which the corresponding factors are not equal, then the number of generators in  $\mathcal{G}_{Q'_N}$  can be  $O(2^{|\mathcal{N}|})$  in the worst case. In practice, usually  $2^{|\mathcal{N}|} \ll 2^n$  resulting in a huge reduction in operation count.

An alternative approach for computing  $\mathcal{C}_O$  could be to use (2.5), however this is more expensive than applying the conversion. This is because the Fourier-Motzkin elimination can generate a quadratic number of new constraints for each variable that it projects out. Many of the generated constraints are redundant and should be removed to keep the algorithm efficient. Redundancy checking is performed by calling a linear solver for every constraint which slows down the computation. We note that recent work [137, 138, 218] makes the redundancy removal more efficient thereby improving its feasibility.

## 2.5 EXPERIMENTAL EVALUATION

In this section, we evaluate the effectiveness of our decomposition approach for analyzing realistic programs. We implemented all of our algorithms in the form of a library for numerical domains which we call ELINA [1]. The source code of ELINA is publicly available at <http://elina.ethz.ch>.

We compare the performance of ELINA against NewPolka [104] and PPL [12], both widely used state-of-the-art libraries for Polyhedra domain analysis. PPL uses the same basic algorithms as NewPolka, but uses a lazy approach for the conversion whereas NewPolka uses an eager approach. PPL is faster than NewPolka for some transformers and slower for others. Like NewPolka, ELINA uses an eager approach. The experimental results of our evaluation show that the polyhedra arising during analysis can indeed be kept partitioned using our approach. We demonstrate dramatic savings in both time and memory across all benchmarks.

### 2.5.1 *Experimental Setup*

In ELINA we use rational numbers encoded using 64-bit integers as in NewPolka and PPL. In the case of an integer overflow, all libraries set the polyhedron to  $\top$ .

**PLATFORM** All of our experiments were carried out on a 3.5 GHz Intel Quad Core i7-4771 Haswell CPU. The sizes of the L1, L2, and L3 caches are 256 KB, 1024 KB, and 8192 KB, respectively, and the main memory has 16 GB. Turbo boost was disabled for consistency of measurements. All libraries were compiled with gcc 5.2.1 using the flags `-O3 -m64 -march=native`.

**ANALYZER** We use the *crab-llvm* analyzer which is part of the SeaHorn [91] verification framework. The analyzer is written in C++ and analyzes LLVM bitcode for C programs. It generates polyhedra invariants which are then checked for satisfiability with an SMT-solver. The analysis is intra-procedural and the time for analyzing different functions in the analyzed program varies.

### 2.5.2 *Experimental Results*

We measured the time and memory consumed for the Polyhedra analysis by NewPolka, PPL, and ELINA on more than 1500 benchmarks. We used a time limit of 4 hours and a memory limit of 12 GB for our experiments.

**BENCHMARKS** We tested the analyzer on the benchmarks of the popular software verification competition [24]. The competition provides benchmarks in different categories. We chose three categories which are suited for the analysis with a numerical domain: (a) Linux Device Drivers (LD), (b) Control Flow (CF), and (c)

Table 2.3: Speedup of Polyhedra domain analysis for ELINA over NewPolka and PPL.

Benchmark	Category	LOC	NewPolka		PPL		ELINA		Speedup ELINA vs.	
			time(s)	memory(GB)	time(s)	memory(GB)	time(s)	memory(GB)	NewPolka	PPL
firewire_firedtv	LD	14506	1367	1.7	331	0.9	0.4	0.2	3343	828
net_fddi_skfp	LD	30186	5041	11.2	6142	7.2	9.2	0.9	547	668
mtd_ubi	LD	39334	3633	7	M0	M0	4	0.9	908	$\infty$
usb_core_main0	LD	52152	11084	2.7	4003	1.4	65	2	170	62
tty_synclinkmp	LD	19288	T0	T0	M0	M0	3.4	0.1	>4235	$\infty$
scsi_advansys	LD	21538	T0	T0	T0	T0	4	0.4	>3600	>3600
staging_vt6656	LD	25340	T0	T0	T0	T0	2	0.4	>7200	>7200
net_ppp	LD	15744	T0	T0	10530	0.15	924	0.3	>16	11.4
p10_l00	CF	592	841	4.2	121	0.9	11	0.8	76	11
p16_l40	CF	1783	M0	M0	M0	M0	11	3	$\infty$	$\infty$
p12_l57	CF	4828	M0	M0	M0	M0	14	0.8	$\infty$	$\infty$
p13_l53	CF	5816	M0	M0	M0	M0	54	2.7	$\infty$	$\infty$
p19_l59	CF	9794	M0	M0	M0	M0	70	1.7	$\infty$	$\infty$
ddv_all	HM	6532	710	1.4	85	0.5	0.05	0.1	12772	1700

Heap Manipulation (HM). Each of these categories contains hundreds of benchmarks and invariants that cannot be expressed using weaker domains such as Octagon, Zone, or others.

Table 2.3 shows the time (in seconds) and the memory (in GB) consumed for Polyhedra analysis with NewPolka, PPL, and ELINA on 14 large benchmarks. In the table, the entry T0 means that the analysis did not finish within 4 hours. Similarly, the entry M0 means that the analysis exceeded the memory limit. The benchmarks in the table were selected based on the following criteria:

- The analysis ran for  $> 10$  minutes with NewPolka.
- There was no integer overflow during the analysis for the most time consuming function in the analyzed program.

At each step of the analysis, our algorithms obtain mathematically/semantically the same polyhedra as NewPolka and PPL, just represented differently (decomposed). In the actual implementation, since our representation contains different numbers, ELINA may produce an integer overflow before NewPolka or PPL. However, on the benchmarks shown in Table 2.3, NewPolka overflowed 296 times whereas ELINA overflowed 13 times. We also never overflowed on the procedures in the benchmarks that are most expensive to analyze (neither did NewPolka and PPL). Thus ELINA does not benefit from faster convergence due to integer overflows which sets the corresponding polyhedra to  $\top$ .

We show the speedups for ELINA over NewPolka and PPL which range from one to at least four orders of magnitude. In the case of a time out, we provide a lower bound on the speedup, which is very conservative. Whenever there is memory overflow, we show the corresponding speedup as  $\infty$ , because the analysis can never finish on the given machine even if given arbitrary time.

Table 2.3 also shows the number of lines of code for each benchmark. The largest benchmark is `usb_core_main0` with 52K lines of code. ELINA analyzes it in 65 seconds whereas NewPolka takes  $> 3$  hours and PPL requires  $> 1$  hour. PPL performs

Table 2.4: Partition statistics for Polyhedra analysis with ELINA.

Benchmark	$\mathcal{X}$		$\mathcal{N}$		nb		trivial/total
	max	avg	max	avg	max	avg	
firewire_firedtv	159	80	24	5	31	6	10/577
net_fddi_skfp	589	111	89	24	89	15	76/5163
mtd_ubi	528	60	111	10	57	12	27/2518
usb_core_main0	365	72	267	29	61	15	80/14594
tty_synclinkmp	332	47	48	8	34	10	23/3862
scsi_advansys	282	67	117	11	82	19	11/2315
staging_vt6656	675	53	204	10	62	6	35/1330
net_ppp	218	59	112	33	19	5	1/2350
p10_l00	303	184	234	59	38	29	0/601
p16_l40	188	125	86	39	53	38	4/186
p12_l57	921	371	461	110	68	28	4/914
p13_l53	1631	458	617	149	78	28	5/1325
p19_l59	1272	476	867	250	65	21	9/1754
ddv_all	45	22	7	2	14	8	5/124

better than NewPolka on 5 benchmarks whereas NewPolka has better performance than PPL on 2 benchmarks. Half of the benchmarks in the Linux Device Drivers category do not finish within the time and memory limit with NewPolka and PPL. `net_ppp` takes the longest to finish with ELINA ( $\approx 15$  minutes).

All benchmarks in the Control Flow category run out of memory with both NewPolka and PPL except for `p10_l00` which is also the smallest. This is because all benchmarks in this category contain a large number of join points which creates an exponential number of generators for both libraries. With our approach, we are able to analyze all benchmarks in this category in  $\leq 3$  GB. There are  $> 500$  benchmarks in this category not shown in Table 2.3 that run out of memory with both libraries whereas ELINA is able to analyze them.

There is only one large benchmark in the Heap Manipulation category. For it we get a 12722x speedup and also save 14x in memory over NewPolka. The gain over PPL is 1700x in time and 5x in memory.

We gathered statistics on the number of variables ( $|\mathcal{X}|$ ), the size of the largest block ( $|\mathcal{N}|$ ) in the respective partition, and its number of blocks (nb) after each join for all benchmarks. Table 2.4 shows *max* and *average* of these quantities. It can be seen that the number of variables in  $\mathcal{N}$  is significantly smaller than in  $\mathcal{X}$  resulting in complexity gains. The last column shows the fraction of the times the partition is trivial (equal to  $\{\mathcal{X}\}$ ). It is very low and happens only when the number of variables is very small.

The bottleneck for the analysis is the conversion applied on  $\mathcal{G}_{P \sqcup Q}$  during the join transformer. ELINA applies conversion on  $\mathcal{G}_{P'_N \sqcup Q'_N}$  which contains variables from the set  $\mathcal{N} = \bigcup_{\mathcal{A} \in \pi \setminus \mathcal{U}} \mathcal{A}$  whereas NewPolka and PPL apply conversion for all

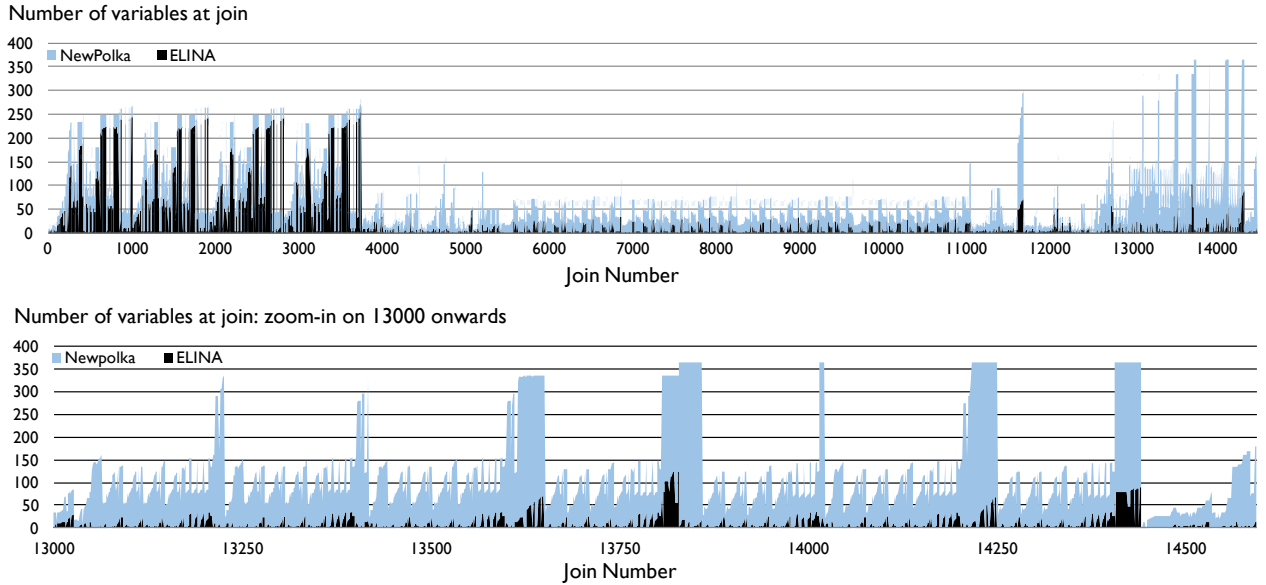


Figure 2.7: The join transformer during the analysis of the `usb_core_main0` benchmark. The x-axis shows the join number and the y-axis shows the number of variables in  $\mathcal{N} = \bigcup_{A \in \pi \setminus \mathcal{U}} \mathcal{A}$  (subset of variables affected by the join) and in  $\mathcal{X}$ . The first figure shows these values for all joins whereas the second figure shows it for one of the expensive regions of the analysis.

variables in the set  $\mathcal{X}$ . The first part of Fig. 2.7 plots the number of variables in  $\mathcal{N}$  and in  $\mathcal{X}$  for all joins during the analysis of the `usb_core_main0` benchmark.  $|\mathcal{X}|$  varies for all joins at different program points. It can be seen that the number of variables in  $\mathcal{N}$  is close to the number of variables in  $\mathcal{X}$  till join number 5000. Although the number of variables is large in this region, it is not the bottleneck for NewPolka and PPL as the number of generators is linear in the number of variables. We get a speedup of 4x mainly due to our conversion transformer which leverages sparsity. The most expensive region of the analysis for both NewPolka and PPL is after join number 5000 where the number of generators grows exponentially. In this region,  $\mathcal{N}$  contains 9 variables on average whereas  $\mathcal{X}$  contains 54. The second part of Fig. 2.7 zooms in one of these expensive regions. Since the cost of conversion depends exponentially on the number of generators which in turn depends on the number of variables, we get a large speedup.

We also measured the effect of optimizations not related to partitioning on the overall speedup. The maximum difference was on the `net_ppp` benchmark which was 2.4x slower without the optimizations.

**REMAINING BENCHMARKS** Above we presented the results for 14 large benchmarks. The remaining benchmarks either finish or run out of memory in  $< 10$  minutes with NewPolka or the analysis produces an integer overflow in the most time consuming function. The bound on the speedup for these benchmarks ranges from 2x to 76x.

## 2.6 DISCUSSION

Program analysis with the exponentially expensive Polyhedra domain was believed to be intractable for analyzing large real-world programs for around 40 years. In this chapter, we presented a theoretical framework, and its implementation, for speeding up the Polyhedra domain analysis by orders of magnitude without losing precision. Our key idea is to decompose the analysis and its transformers to work on sets of smaller polyhedra, thus reducing its asymptotic time and space complexity. This is possible because the statements in real-world programs affect only a few variables in the polyhedra. As a result, the variable set can be partitioned into independent subsets. The challenge in maintaining these partitions is in handling their continuous change during the analysis. These changes cannot be predicted statically in advance. Our partition computations leverage dynamic analysis state and the semantics of the Polyhedra transformers. These computations are fast and produce sufficiently fine partitions, which enables significant speedups. Precision-wise, our decomposed analysis computes polyhedra semantically equivalent to those produced by the original non-decomposed analysis at each step. Overall our analysis computes the same invariants as the original one, but significantly faster.

We provided a complete end-to-end implementation of the Polyhedra domain analysis within ELINA [1]. Benchmarking against two state-of-the-art libraries for the Polyhedra analysis, namely, NewPolka and PPL, on real-world programs including Linux device drivers and heap manipulating programs showed orders of magnitude speedup or successful completion where the others time-out or exceed memory. We believe that our framework presents a significant step forward in making Polyhedra domain analysis practical for real-world use.

In the next chapter, we will show that our theoretical framework of online decomposition is generic and can be extended to any numerical domain that maintains linear constraints between program variables.



---

GENERALIZING ONLINE DECOMPOSITION

---

In Chapter 2, we presented our theoretical framework for decomposing the standard implementation of the Polyhedra domain that is based on the most precise Polyhedra transformers. However, the Polyhedra domain can also be implemented differently, with other, less precise transformers. Further, there are other existing popular numerical domains such as Octahedron [51], TVPI [183], Octagon [142], and Zone [140] and these too can be implemented in a variety of ways. So the basic question is how to apply the idea of online decomposition to all numerical domains and potentially different implementations.

The online decomposition for the Polyhedra [190] domain (Chapter 2) is specialized for the particular implementations of the domain. In [189], we developed an online decomposition for the Octagon domain. In both cases, the decomposition was manually designed from scratch for the standard transformers of the particular domain. The downside of this approach is that the substantial effort invested in decomposing the transformers of the specific implementation of the domain cannot be reused and needs to be repeated for every new implementation. This task is difficult and error-prone as it requires devising new algorithms and data structures from scratch each time.

To illustrate the issue, consider an element  $\mathcal{J} = \{-x_1 - x_2 \leq 0, -x_3 \leq 0, -x_4 \leq 0\}$  in the Octagon domain (which captures constraints of the form  $\pm x_i \pm x_j \leq c, c \in \mathbb{R}$ , between the program variables) and the conditional expression  $x_2 + x_3 + x_4 \leq 1$ . There are multiple ways to define a sound conditional transformer in the Octagon domain for the given conditional expression. One may define a sound conditional transformer  $T_1$  that adds the non-redundant constraint  $-x_1 + x_4 \leq 1$  to  $\mathcal{J}$  resulting in the output  $\mathcal{J}' = \{-x_1 - x_2 \leq 0, -x_3 \leq 0, -x_4 \leq 0, -x_1 + x_4 \leq 1\}$  whereas another transformer  $T_2$  may add  $x_2 + x_3 \leq 1$  to  $\mathcal{J}$  resulting in  $\mathcal{J}'' = \{-x_1 - x_2 \leq 0, -x_3 \leq 0, -x_4 \leq 0, x_2 + x_3 \leq 1\}$ . The specialized decomposition for the Octagon domain [189] requires access to the exact definition of the transformer, i.e., it will produce different decompositions for  $T_1$  and  $T_2$  as the set of variables in the constraints added by the two transformers are disjoint.

**THIS CHAPTER** Our key objective is to bring the power of decomposition to all sub-polyhedra domains without requiring complex manual effort from the domain designer. This enables domain designers to achieve speed-ups without requiring

them to rewrite all abstract transformers from scratch each time. More formally, our goal is to provide a systematic correct-by-construction method that, given a sound abstract transformer  $T$  in a sub-polyhedra domain (e.g., Zone), generates a sound decomposed version of  $T$  that is faster than  $T$  and *does not* require any change to the internals of  $T$ . In this chapter, we present a construction that achieves this objective under certain conditions. We provide theoretical guarantees on the convergence, monotonicity, and precision of the decomposed analysis with respect to the non-decomposed analysis. We also show that the obtained decomposed transformers are faster than the prior, hand-tuned decomposed domains from [189, 190].

The work in this chapter was published in [191].

**MAIN CONTRIBUTIONS** We make the following contributions:

- We introduce a general construction for obtaining decomposed transformers from given non-decomposed transformers of existing numerical domains. Our construction is “black-box:” it does not require changes to the underlying algorithms implemented in the original non-decomposed transformers.
- We provide conditions on the non-decomposed transformers under which our decomposition maintains precision and equivalence at fixpoint.
- We apply our method to decompose standard transformers of three popular and expensive domains: Polyhedra, Octagon, and Zone. For these, we provide complete end-to-end implementations as part of ELINA [1].
- We evaluate the effectiveness of our decomposed analysis against state-of-the-art implementations on large real-world benchmarks including Linux device drivers. Our evaluation shows up to 6x and 2x speedups compared to state-of-the-art manually decomposed domain implementations and orders of magnitude speedups compared to non-decomposed Polyhedra and Octagon implementations. For Zone, we achieve speedups of up to 6x compared to our own, non-decomposed implementation.

### 3.1 GENERIC MODEL FOR NUMERICAL ABSTRACT DOMAINS

In this section, we introduce a generic model for the abstract domains to which our theory applies. An abstract domain consists of a set of abstract elements and a set of transformers that model the effect of program statements (assignment, conditionals, etc.) and control flow (join etc.) on the abstract elements. Let  $\mathcal{X} = \{x_1, x_2, \dots, x_n\}$  be a set of program variables. We consider sub-polyhedra domains, i.e., numerical abstract domains  $\mathcal{D}$  that encode linear relationships between the variables in  $\mathcal{X}$  of the form:

$$\sum_{i=1}^n a_i \cdot x_i \otimes c, \quad \text{where } x_i \in \mathcal{X}, a_i \in \mathbb{Z}, \otimes \in \{\leq, =\}, c \in \mathcal{C}. \quad (3.1)$$

Typical choices for  $\mathcal{C}$  include  $\mathbb{Q}$  (rationals) and  $\mathbb{R}$  (reals). As with any abstraction, the design of a numerical domain is guided by the cost vs. precision tradeoff. For instance, the Polyhedra domain [57] is the most precise domain yet it is also the most expensive. On the other hand, the Interval domain is cheap but also very imprecise as it does not preserve relational information between variables. Between these two sit a number of domains with varying degrees of precision and cost: examples include Two Variables Per Inequality (TVPI) [143], Octagon [142], and Zone [140].

**REPRESENTING DOMAIN CONSTRAINTS** We introduce notation for describing the set of constraints a given domain  $\mathcal{D}$  can express for the variables in  $\mathcal{X}$ . This set of constraints is referred to as  $\mathcal{L}_{\mathcal{X},\mathcal{D}}$  and is determined by four components  $(n, \mathcal{R}, \mathcal{T}, \mathcal{C})$ :

- The size  $n$  of the variable set  $\mathcal{X}$ .
- A relation  $\mathcal{R} \subseteq \mathcal{R}_1 \times \mathcal{R}_2 \times \dots \times \mathcal{R}_n$  to describe the universe of possible coefficients. Each  $\mathcal{R}_i \subseteq \mathbb{Z}$  is a set of integers defining the allowed values for the coefficient  $a_i$ . Typical examples for  $\mathcal{R}_i$  include  $\mathbb{Z}$ ,  $\mathbb{U} = \{-1, 0, 1\}$ , and  $\mathbb{L} = \{-2^k, 0, 2^k \mid k \in \mathbb{Z}\}$ .
- The set  $\mathcal{T} \subseteq \{\leq, =\}$  determining equality/inequality constraints.
- The set  $\mathcal{C}$  containing the allowed values for the constant  $c$  in (3.1). Typical examples include  $\mathbb{Q}$  and  $\mathbb{R}$ .

Table 3.1 shows common constraints in the above notation allowed by different numerical domains. The set of constraints  $\mathcal{L}_{\mathcal{X},\mathcal{D}}$  representable by a domain  $\mathcal{D}$  contains all constraints of the form  $\sum_{i=1}^n a_i \cdot x_i \otimes c$  where: (i) the coefficient list of each expression  $\sum_{i=1}^n a_i \cdot x_i$  is a permutation of a tuple in  $\mathcal{R}$ , (ii)  $\otimes \in \mathcal{T}$ , and (iii) the constant  $c \in \mathcal{C}$ . For instance, the possible constraints  $\mathcal{L}_{\mathcal{X},\text{Octagon}}$  for the Octagon domain over real numbers are described via the tuple  $(n, \mathbb{U}^2 \times \{0\}^{n-2}, \{\leq, =\}, \mathbb{R})$ .

**Example 3.1.1.** Consider a program with four variables and a fictive domain that can relate at most two:

$$\mathcal{X} = \{x_1, x_2, x_3, x_4\} \text{ and } \mathcal{L}_{\mathcal{X},\mathcal{D}} : (4, \mathbb{U}^2 \times \{0\}^2, \{\leq, =\}, \{1, 2\}).$$

Here, the constraint  $2x_1 + 3x_4 \leq 2 \notin \mathcal{L}_{\mathcal{X},\mathcal{D}}$  as no permutation of tuples in  $\mathbb{U}^2 \times \{0\}^2$  can produce  $(2, 0, 0, 3)$ . Similarly,  $x_2 - x_3 \leq 3 \notin \mathcal{L}_{\mathcal{X},\mathcal{D}}$  even though there exists a permutation of tuples in  $\mathbb{U}^2 \times \{0\}^2$  that can produce  $(0, 1, -1, 0)$ , but  $3 \notin \mathcal{C}$ . However, the constraints  $x_2 - x_3 \leq 1$  and  $x_2 - x_3 = 2$  are in  $\mathcal{L}_{\mathcal{X},\mathcal{D}}$ .

Table 3.1: Instantiation of constraints expressible in various numerical domains.

Domain	$\mathcal{R}$	$\mathcal{T}$	$\mathcal{C}$	Reference
Polyhedra	$\mathbb{Z}^n$	$\{\leq, =\}$	$\mathbb{Q}, \mathbb{R}$	[57]
Linear equality	$\mathbb{Z}^n$	$\{=\}$	$\mathbb{Q}, \mathbb{R}$	[112]
Octahedron	$\mathbb{U}^n$	$\{\leq, =\}$	$\mathbb{Q}, \mathbb{R}$	[51]
Stripes	$\{(a, a, -1, 0, \dots, 0) \mid a \in \mathbb{Z}\} \cup \{(0, a, -1, 0, \dots, 0) \mid a \in \mathbb{Z}\}$	$\{\leq, =\}$	$\mathbb{Q}, \mathbb{R}$	[71]
TVPI	$\mathbb{Z}^2 \times \{0\}^{n-2}$	$\{\leq, =\}$	$\mathbb{Q}, \mathbb{R}$	[183]
Octagon	$\mathbb{U}^2 \times \{0\}^{n-2}$	$\{\leq, =\}$	$\mathbb{Q}, \mathbb{R}$	[142]
Logahedra	$\mathbb{L}^2 \times \{0\}^{n-2}$	$\{\leq, =\}$	$\mathbb{Q}, \mathbb{R}$	[101]
Zone	$\{1, 0\} \times \{0, -1\} \times \{0\}^{n-2}$	$\{\leq, =\}$	$\mathbb{Q}, \mathbb{R}$	[140]
Upper bound	$\{1\} \times \{-1\} \times \{0\}^{n-2}$	$\{\leq\}$	$\{0\}$	[134]
Interval	$\{1, -1\} \times \{0\}^{n-1}$	$\{\leq, =\}$	$\mathbb{Q}, \mathbb{R}$	[54]

**DEFINING AN ABSTRACT DOMAIN** An abstract element  $\mathcal{J}$  in a domain  $\mathcal{D}$  is a conjunction of a finite number of constraints from  $\mathcal{L}_{\mathcal{X}, \mathcal{D}}$ . By abuse of notation we will represent  $\mathcal{J}$  as a set of constraints (interpreted as a conjunction of the constraints in the set). We ignore the equivalent generator representation of an abstract element where the element is encoded as a collection of vertices, rays, and lines (Section 2.1.1) as the constraint representation leads to a clearer exposition of the ideas. However, our technical results are also valid with the generator representation. The set of all possible abstract elements in  $\mathcal{D}$  is denoted with  $\mathcal{P}_{\mathcal{D}}$  and typically forms a lattice  $(\mathcal{P}_{\mathcal{D}}, \sqsubseteq, \sqcup, \sqcap, \top, \perp)$  with respect to the defined domain order  $\sqsubseteq$ . Given abstract elements  $\mathcal{J}$  and  $\mathcal{J}'$ ,  $\mathcal{J} \sqcup \mathcal{J}'$  is the smallest element in the domain covering both  $\mathcal{J}$  and  $\mathcal{J}'$  and is computed or approximated by the join transformer. Similarly  $\mathcal{J} \sqcap \mathcal{J}'$  is the meet, computed, e.g., as  $\mathcal{J} \cup \mathcal{J}'$ . While our theory handles all transformers in a given domain  $\mathcal{D}$ , our presentation focuses on the core transformers (as in Chapter 2), namely: conditional containing a linear constraint, assignment with a linear expression, meet ( $\sqcap$ ), join ( $\sqcup$ ), and widening ( $\nabla$ ). We chose these because they are the most expensive domain transformers and thus their design shows the most variation, i.e., they can be implemented in multiple ways.

As is standard, we use the meet-preserving concretization function  $\gamma$  to denote with  $\gamma(\mathcal{J})$  the concrete element (polyhedron) represented by the abstract element  $\mathcal{J}$ . We note that it is possible for  $\mathcal{J}$  to include redundant constraints, that is, removing a constraint from  $\mathcal{J}$  may not change the represented concrete element  $\gamma(\mathcal{J})$ . Further, the minimal (without any redundancy) representation of a concrete element  $\gamma(\mathcal{J})$  need not be unique, i.e., there could be two distinct abstract elements  $\mathcal{J}$  and  $\mathcal{J}'$  with  $\gamma(\mathcal{J}) = \gamma(\mathcal{J}')$ :

**Example 3.1.2.**  $\mathcal{J} = \{x_1 = 0, x_2 = 0\}$  and  $\mathcal{J}' = \{x_1 = 0, x_2 = 0, x_1 = x_2\}$  represent the same concrete element  $\gamma(\mathcal{J})$  in the Polyhedra domain. However,  $\mathcal{J}'$  contains the redundant constraint  $x_1 = x_2$ .  $\mathcal{J}$  is not the only minimal representation as  $\mathcal{J}'' = \{x_1 = 0, x_1 = x_2\}$  is also minimal for  $\gamma(\mathcal{J})$ .

We next define what it means for an abstract transformer to be sound.<sup>1</sup>

**Definition 3.1.1.** A given abstract transformer  $T$  is sound w.r.t to its concrete transformer  $T^\#$  iff for any element  $J \in \mathcal{D}$ ,  $T^\#(\gamma(J)) \subseteq \gamma(T(J))$ .

The soundness criterion above is naturally extended to transformers with multiple arguments.

**Definition 3.1.2.** We say an abstract domain  $\mathcal{D}$  is *closed* (also called forward complete in [83, 164]) for a concrete transformer  $T^\#$  (e.g., conditional, meet) iff it can be done precisely in the domain, i.e., if there exists an abstract transformer  $T$  corresponding to that concrete transformer such that for any abstract element  $J$  in  $\mathcal{D}$ ,  $\gamma(T(J)) = T^\#(\gamma(J))$ .

The Polyhedra domain is closed for conditional, assignment, and meet, but not for the join. All other domains in Table 3.1 are only closed for the meet. Indeed, a crucial aspect of abstract interpretation is to permit sound approximations of transformers for which the domain is not closed.

**Example 3.1.3.** The Octagon domain is not closed for the conditional transformer. For example, if the condition is  $x_1 - 2x_2 \leq 0$  and the abstract element is  $J = \{x_1 \leq 1, x_2 \leq 0\}$ , then the concrete element  $T^\#(\gamma(J)) = \{x_1 \leq 1, x_2 \leq 0, x_1 - 2x_2 \leq 0\}$  is not representable exactly in the Octagon domain (because the constraint  $x_1 - 2x_2 \leq 0$  is not exactly representable).

A useful concept in analysis (and one we refer to throughout the chapter) is that of a best abstract transformer.

**Definition 3.1.3.** A (unary) abstract transformer  $T$  in  $\mathcal{D}$  is best iff for any other sound unary abstract transformer  $T'$  (corresponding to the same concrete transformer  $T^\#$ ) it holds that for any element  $J$  in  $\mathcal{D}$ ,  $T$  always produces a more precise result (in the concrete), that is,  $\gamma(T(J)) \subseteq \gamma(T'(J))$ . The definition is naturally lifted to multiple arguments.

In Example 3.1.3, a possible sound approximation for the output in the Octagon domain is  $J'' = J$  while a best transformer would produce  $\{x_1 \leq 0, x_2 \leq 0, x_1 - x_2 \leq 0\}$ . Since there can be multiple abstract elements with the same concretization, there can be multiple best abstract transformers in  $\mathcal{D}$ . We require the sub-polyhedra abstract domains to be equipped with a best transformer and also be closed under meet. Due to these restrictions, our theory does not apply to the Zonotope [81, 82, 191] and DeepPoly [188] domains.

<sup>1</sup> Throughout the chapter we will simply use the term transformer to mean a sound abstract transformer.

## 3.2 DECOMPOSING ABSTRACT ELEMENTS

In this section, we introduce the needed notation and concepts for decomposing abstract elements and transformers. We extend the terminology of *partitions*, *blocks*, and *factors* as introduced in Section 2.2.1 for handling the decomposition of the elements of the abstract domain  $\mathcal{D}$ . We write  $\pi_{\mathcal{J}}$  for referring to the unique finest partition for an element  $\mathcal{J}$  in  $\mathcal{D}$ . Each factor  $\mathcal{J}_k \subseteq \mathcal{J}$  is defined by the constraints that exist between the variables in the corresponding block  $\mathcal{X}_k \in \pi_{\mathcal{J}}$ .  $\mathcal{J}$  can be recovered from the set of factors by taking the union of the constraint sets  $\mathcal{J}_k$ .

**Example 3.2.1.** Consider the element  $\mathcal{J} = \{x_1 - x_2 \leq 1, x_3 \leq 0, x_4 \leq 0\}$  in the TVPI domain

$$\mathcal{X} = \{x_1, x_2, x_3, x_4\} \text{ and } \mathcal{L}_{\mathcal{X}, \text{TVPI}} : (4, \mathbb{Z}^2 \times \{0\}^2, \{\leq, =\}, \mathbb{Q}).$$

Here  $\mathcal{X}$  can be partitioned into three blocks with respect to  $\mathcal{J}$  resulting in three factors:

$$\pi_{\mathcal{J}} = \{\{x_1, x_2\}, \{x_3\}, \{x_4\}\}, \quad \mathcal{J}_1 = \{x_1 - x_2 \leq 1\}, \quad \mathcal{J}_2 = \{x_3 \leq 0\}, \quad \text{and } \mathcal{J}_3 = \{x_4 \leq 0\}.$$

For a given  $\mathcal{D}$ ,  $\pi_{\perp} = \pi_{\top} = \perp = \{\{x_1\}, \{x_2\}, \dots, \{x_n\}\}$ . More generally, note that  $\mathcal{J} \sqsubseteq \mathcal{J}'$  does not imply that  $\pi_{\mathcal{J}'}$  is finer, coarser, or comparable to  $\pi_{\mathcal{J}}$ .

**DIFFERENT PARTITIONS FOR EQUIVALENT ELEMENTS** To gain a deeper understanding of partitions for abstract elements, there are two interesting points worth noting. First, it is possible that two semantically equivalent abstract elements  $\mathcal{J}, \mathcal{J}'$  in the domain have different partitions. That is, even if  $\gamma(\mathcal{J}) = \gamma(\mathcal{J}')$ , it may be the case that  $\pi_{\mathcal{J}} \neq \pi_{\mathcal{J}'}$  or  $\pi_{\mathcal{J}} \sqsubset \pi_{\mathcal{J}'}$ :

**Example 3.2.2.** Consider  $\mathcal{J} = \{x_1 \leq x_2, x_2 = 0, x_3 = 0\}$  with the finest partition  $\pi_{\mathcal{J}} = \{\{x_1, x_2\}, \{x_3\}\}$ ,  $\mathcal{J}' = \{x_1 \leq 0, x_2 = 0, x_3 = 0\}$  with  $\pi_{\mathcal{J}'} = \{\{x_1\}, \{x_2\}, \{x_3\}\}$  and  $\mathcal{J}'' = \{x_1 \leq x_3, x_2 = 0, x_3 = 0\}$  with  $\pi_{\mathcal{J}''} = \{\{x_1, x_3\}, \{x_2\}\}$  in the Polyhedra domain. Here  $\gamma(\mathcal{J}) = \gamma(\mathcal{J}') = \gamma(\mathcal{J}'')$ , but the partitions are pairwise different.

Second, it is possible that for a given abstract element  $\mathcal{J}$ , there exists an equivalent element  $\mathcal{J}'$  with finer partition but  $\mathcal{J}'$  is not representable in the domain. This shows a potential limitation of syntactic partitions used in our framework.

**Example 3.2.3.** Consider the Stripes domain

$$\mathcal{X} = \{x_1, x_2, x_3\}, \quad \mathcal{L}_{\mathcal{X}, \text{Stripes}} : \{3, \{(a, a, -1) \mid a \in \mathbb{Z}\} \cup \{(0, a, -1) \mid a \in \mathbb{Z}\}, \{\leq, =\}, \mathbb{Q}\},$$

and the abstract element

$$\mathcal{J} = \{x_1 + x_2 - x_3 = 0, -x_2 + x_3 = 0\} \text{ with } \pi_{\mathcal{J}} = \{x_1, x_2, x_3\}.$$

This domain cannot represent the equivalent element  $\mathcal{J}' = \{x_1 = 0, x_2 - x_3 = 0\}$  with partition  $\pi_{\mathcal{J}'} = \{\{x_1\}, \{x_2, x_3\}\}$ , which is finer than  $\pi_{\mathcal{J}}$ . This is because the constraint  $x_1 = 0$  is not representable in the Stripes domain.

It is important we guarantee that regardless of how approximate a given transformer  $T$  is, the partition we end up computing for  $T$  is always sound (permissible) for the output abstract element  $\mathcal{J}$  produced by  $T$ . Next, we extend Definition 2.2.1:

**Definition 3.2.1.** A partition  $\bar{\pi}$  is permissible for an abstract element  $\mathcal{J}$  in  $\mathcal{D}$  if it is *coarser* than  $\pi_{\mathcal{J}}$ , that is, if  $\bar{\pi} \sqsupseteq \pi_{\mathcal{J}}$ .

The variables related in  $\pi_{\mathcal{J}}$  are also related in any permissible partition of  $\mathcal{J}$ , but not vice-versa. In Example 3.2.1,  $\{\{x_1, x_2\}, \{x_3, x_4\}\}$  is permissible for  $\mathcal{J}$  while  $\{\{x_1\}, \{x_2, x_3, x_4\}\}$  is not. We will use  $\bar{\pi}_{\mathcal{J}}$  to denote a permissible partition for  $\mathcal{J}$ .

### 3.3 RECIPE FOR DECOMPOSING TRANSFORMERS

A primary objective of this work is to define a mechanical recipe which takes as input a sound abstract transformer and produces as output a sound and decomposed variant of that transformer, thus resulting in better analysis performance. In this section we describe the general recipe and illustrate its actual use.

At first glance the above challenge appears fundamentally difficult because there are many ways to define a sound transformer in a domain  $\mathcal{D}$ . Standard implementations of popular numerical domains like Octagon, Zone, TVPI, and others, do not necessarily implement the best transformers as they can be expensive; instead the domains often approximate them. Interestingly, as pointed out earlier, such an approximation can make the associated partition both coarser or finer. That is, the partitioning function is not monotone. Here is an example illustrating this point:

**Example 3.3.1.** Consider the elements  $\mathcal{J} = \{x_1 \leq 0, x_2 \leq 0, x_1 - x_2 \leq 0\}$  with  $\pi_{\mathcal{J}} = \{\{x_1, x_2\}\}$  and  $\mathcal{J}' = \{x_1 \leq 0, x_2 \leq 0\}$  with  $\pi_{\mathcal{J}'} = \{\{x_1\}, \{x_2\}\}$  in the Octagon domain. Here,  $\gamma(\mathcal{J}) \subset \gamma(\mathcal{J}')$  and  $\pi_{\mathcal{J}} \sqsupseteq \pi_{\mathcal{J}'}$ . Now consider the element  $\mathcal{J}'' = \{x_1 + x_2 \leq 0\}$  with  $\pi_{\mathcal{J}''} = \{\{x_1, x_2\}\}$ . Here,  $\gamma(\mathcal{J}') \subset \gamma(\mathcal{J}'')$  and  $\pi_{\mathcal{J}'} \sqsubseteq \pi_{\mathcal{J}''}$ .

**Definition 3.3.1.** A transformer  $T$  in  $\mathcal{D}$  is decomposable for input  $\mathcal{J}$  iff the output  $\mathcal{J}_O = T(\mathcal{J})$  results in a partition  $\pi_{\mathcal{J}_O} \neq \top$ . For binary transformers, the definition is analogous.

There are many ways to define sound approximations of the best transformers in  $\mathcal{D}$ . As a consequence, it is possible to have two transformers  $T_1, T_2$  in  $\mathcal{D}$  on the same input  $\mathcal{J}$  such that one produces the  $\top$  partition for the output while the other does not. There are two principal ways to obtain a decomposable transformer: (a) white box: here, one designs the transformer from scratch, maintaining the (changing) partitions during the analysis, and (b) black box: here, one provides a construction for decomposing existing transformers without knowing their internals. In the next section, we pursue the second approach and show that it is possible and, under certain conditions there is no loss of precision. As a preview, we describe the high-level steps that one needs to perform dynamically in our black-box decomposition.

A CONSTRUCTION FOR ONLINE TRANSFORMER DECOMPOSITION There are three main steps for decomposing a given transformer:

1. compute (if needed) partitions for the input(s) of the transformer,
2. compute a partition for the output based on the program statement/expression and input partition(s) from step 1,
3. refactor the inputs according to the partition from step 2,
4. apply the transformer on one or more factors of the inputs from step 3.

We next describe these steps in greater detail.

In an ideal setting, one would always work with the finest partition for the inputs and the output to (optimally) reduce the cost of the transformer. The finest partition for the inputs of a given transformer can always be computed from scratch by taking the abstract element and connecting the variables that occur in the same constraint in that element. The downside is that this computation may incur significant overhead. For example, computing the finest partition for an element in the Octagon domain from scratch has the same quadratic complexity (in the number of variables) as the standard conditional, meet, and assignment transformers. Thus, it may nullify potential performance gains from decomposing these transformers. In our approach we iteratively maintain permissible partitions.

To compute the output partition, a naive way is to first run the transformer, obtain an abstract element as a result, and then compute the partition for that element. Of course, this approach is useless since running the standard transformer prevents performance gains. Thus, the challenge is to determine first a permissible output partition that is not too coarse at little cost so that then the transformers can be applied only to relevant factors. Indeed, in our construction we compute a permissible partition for the output based on permissible partitions of the input, the program statement, and possibly additional information that is cheaply available. In practice, our computed partitions are reasonably fine so that we get significant performance gains. We provide conditions under which computed partitions are optimal in Section 3.4. One can also refine the partitions by computing them from scratch occasionally in case they become too coarse.

Next we refactor the input(s) so that the partition corresponds to the output partition according to Algorithm 2.1. In the last step, once the output partition is obtained, the associated abstract element is computed directly in decomposed form by applying the original transformer to one or more factor(s) of the input(s). Applying this transformer on smaller factors reduces its complexity and results in increased performance. In certain cases, the permissible partition for the output can be further refined after applying the transformer and without adding significant overhead. We identify such cases in Section 3.4.

Our approach is generic in nature and can decompose the standard transformers of the existing sub-polyhedra numerical abstract domains. We implemented our



recipe and applied it to Polyhedra, Octagon, and Zone. Using a set of large Linux device drivers, we show later in Section 3.5 the performance of our generated decomposed transformers vs. transformers obtained via state-of-the-art hand-tuned decomposition [189, 190]. Our approach yields up to 6x speed-ups for Polyhedra and up to 2x speed-ups for Octagon. This speed-up is due to our decomposition theorems (discussed next) that enable, in certain cases, finer decomposition of abstract elements than previously possible. Speedups compared to the original transformers without decomposition are orders of magnitude larger. Further, we also decompose the Zone domain using our approach (for which no previous decomposition exists) without changing the existing domain transformers. We obtain a speedup of up to 6x over the non-decomposed implementation of the Zone domain. In summary, our recipe is generic in nature yet leads to state-of-the-art performance for classic abstract transformers.

### 3.4 DECOMPOSING DOMAIN TRANSFORMERS

In this section we show a construction that takes as input a sound and monotone transformer in a given domain  $\mathcal{D}$  and produces a decomposed variant of the same transformer that operates on part(s) of the input(s). The resulting decomposed transformer is always sound. We define classes of transformers for which the output produced by the decomposed transformer has the same concretization as the original non-decomposed transformer, i.e., there is no loss of precision. Although our results apply to all transformers, we focus on the conditional, assignment, meet, join, and widening transformers. We also show how to obtain finer partitions than the manually decomposed transformers for Polyhedra (Chapter 2) and Octagon considered in prior work [189, 190].

**PARTITIONED ABSTRACT ELEMENTS AND FACTORS** Given an abstract element  $\mathcal{J}$  with permissible partition  $\bar{\pi}_{\mathcal{J}} = \{\mathcal{X}_1, \dots, \mathcal{X}_r\}$ , we denote the associated factors with  $\mathcal{J}(\mathcal{X}_k)$ ,  $1 \leq k \leq r$ . We will write a decomposed abstract element  $\mathcal{J}$  as a set of constraints (and not as set of factors) just as we did before, but always explicitly mention the associated partition.

**COMMON PARTITION AND REFACTORING** We assume that inputs  $\mathcal{J}, \mathcal{J}'$  for binary transformers are partitioned according to a common permissible partition  $\bar{\pi}_{\text{common}}$ . This partition can always be computed as  $\bar{\pi}_{\text{common}} = \bar{\pi}_{\mathcal{J}} \sqcup \bar{\pi}_{\mathcal{J}'}$ , where  $\bar{\pi}_{\mathcal{J}}, \bar{\pi}_{\mathcal{J}'}$  are permissible partitions for  $\mathcal{J}, \mathcal{J}'$  respectively. For all transformers, we assume that the inputs get refactored as per the output partition whenever needed by the corresponding transformer based on Algorithm 2.1.

**ABSTRACT ORDERING** Ordering in the decomposed abstract domain is defined as  $\mathcal{J} \sqsubseteq \mathcal{J}' \equiv \gamma(\mathcal{J}) \subseteq \gamma(\mathcal{J}')$ .

**PRECISION OF DECOMPOSED TRANSFORMERS** We define classes of conditional, assignment, meet, join and widening transformers for which the outputs of both the given non-decomposed  $T$  and our associated decomposed  $T^d$  have the same concretization, i.e.,  $\gamma(T(\mathcal{J})) = \gamma(T^d(\mathcal{J}))$  for all  $\mathcal{J}$  in  $\mathcal{D}$ , which implies that  $T^d$  inherits monotonicity from  $T$ . If the transformer is not in these classes, then both  $\gamma(T(\mathcal{J})) \subset \gamma(T^d(\mathcal{J}))$  or  $\gamma(T(\mathcal{J})) \supset \gamma(T^d(\mathcal{J}))$  are possible and monotonicity may not hold.

If in addition to  $\gamma(T(\mathcal{J})) = \gamma(T^d(\mathcal{J}))$ , the given transformers satisfy

$$\gamma(\mathcal{J}) = \gamma(\mathcal{J}') \Rightarrow \gamma(T(\mathcal{J})) = \gamma(T(\mathcal{J}')),$$

then the analysis with our associated decomposed transformers produces the same semantic invariants at fixpoint (fixpoint equivalence).

### 3.4.1 Conditional

We consider conditional statements of the form  $e \otimes c$  where  $e = \sum_{i=1}^n a_i \cdot x_i$  with  $a_i \in \mathbb{Z}$ ,  $\otimes \in \{\leq, =\}$ , and  $c \in \mathbb{Q}, \mathbb{R}$ , on an abstract element  $\mathcal{J}$  with an associated permissible partition  $\bar{\pi}_{\mathcal{J}}$  in domain  $\mathcal{D}$ . The conditional transformer computes the effect of adding the constraint  $e \otimes c$  to  $\mathcal{J}$ . As discussed in Section 3.1, most existing domains are not closed for the conditional transformer. Moreover, computing the best transformers is expensive in these domains and thus the transformer is usually approximated to strike a balance between precision and cost. The example below illustrates two sound conditional transformers on the same input: the first transformer produces a decomposable output whereas the output of the second results in the  $\top$  partition.

**Example 3.4.1.** Consider

$$\begin{aligned} \mathcal{X} &= \{x_1, x_2, x_3, x_4, x_5, x_6\}, \mathcal{L}_{\mathcal{X}, \text{Polyhedra}} : (6, \mathbb{Z}^6, \{\leq, =\}, \mathbb{Q}), \\ \mathcal{J} &= \{x_1 + x_2 \leq 0, x_3 + x_4 \leq 5\} \text{ with } \bar{\pi}_{\mathcal{J}} = \pi_{\mathcal{J}} = \{\{x_1, x_2\}, \{x_3, x_4\}, \{x_5\}, \{x_6\}\}. \end{aligned}$$

For the conditional statement  $x_5 + x_6 \leq 0$ , a best transformer  $T_1$  may return:

$$\mathcal{J}_O = \{x_1 + x_2 \leq 0, x_3 + x_4 \leq 5, x_5 + x_6 \leq 0\} \text{ with } \bar{\pi}_{\mathcal{J}_O} = \pi_{\mathcal{J}_O} = \{\{x_1, x_2\}, \{x_3, x_4\}, \{x_5, x_6\}\},$$

which is decomposable. However, another sound transformer  $T_2$  may return the non-decomposable output:

$$\mathcal{J}'_O = \{x_1 + x_2 + x_3 + x_4 + x_5 + x_6 \leq 5\} \text{ with } \bar{\pi}_{\mathcal{J}'_O} = \pi_{\mathcal{J}'_O} = \top.$$

Let  $\mathcal{B}_{\text{cond}} = \{x_i \mid a_i \neq 0\}$  be the set of variables with non-zero coefficients in the constraint  $\sum_{i=1}^n a_i \cdot x_i \otimes c$ . The block  $\mathcal{B}_{\text{cond}}^* = \bigcup_{\mathcal{X}_k \cap \mathcal{B}_{\text{cond}} \neq \emptyset} \mathcal{X}_k$  fuses all blocks  $\mathcal{X}_k \in \bar{\pi}_{\mathcal{J}}$  that have non-empty intersection with  $\mathcal{B}_{\text{cond}}$ .

**Example 3.4.2.** Consider  $\mathcal{X} = \{x_1, x_2, x_3, x_4, x_5, x_6\}$  and an element  $\mathcal{J}$  in the Polyhedra domain with  $\bar{\pi}_{\mathcal{J}} = \{\{x_1, x_2, x_3\}, \{x_4, x_5\}, \{x_6\}\}$ . For the conditional  $x_3 + x_6 \leq 0$ ,  $\mathcal{B}_{\text{cond}} = \{x_3, x_6\}$  and  $\mathcal{B}_{\text{cond}}^* = \{x_1, x_2, x_3, x_6\}$ .

**Algorithm 3.1** Decomposed conditional transformer  $T_{\text{cond}}^{\text{d}}$ 


---

```

1: function CONDITIONAL( $(\mathcal{J}, \bar{\pi}_{\mathcal{J}}), \text{stmt}, T_{\text{cond}}$ )
2:    $\mathcal{B}_{\text{cond}}^* := \text{compute\_block}(\text{stmt}, \bar{\pi}_{\mathcal{J}})$ 
3:    $\mathcal{J}_{\text{cond}} := \mathcal{J}(\mathcal{B}_{\text{cond}}^*)$ 
4:    $\bar{\pi}_{\mathcal{J}_O} := \{\mathcal{A} \in \bar{\pi}_{\mathcal{J}} \mid \mathcal{A} \cap \mathcal{B}_{\text{cond}}^* = \emptyset\} \cup \{\mathcal{B}_{\text{cond}}^*\}$ 
5:    $\mathcal{J}_{\text{rest}} := \mathcal{J}(\mathcal{X} \setminus \mathcal{B}_{\text{cond}}^*)$ 
6:    $\mathcal{J}_O := T_{\text{cond}}(\mathcal{J}_{\text{cond}}) \cup \mathcal{J}_{\text{rest}}$ 
7:   return  $(\mathcal{J}_O, \bar{\pi}_{\mathcal{J}_O})$ 
8: end function

```

---

**CONSTRUCTION FOR CONDITIONAL** Algorithm 3.1 shows our construction for decomposing a given conditional transformer  $T_{\text{cond}}$ . Given an input element  $\mathcal{J}$  with a permissible partition  $\bar{\pi}_{\mathcal{J}}$  in domain  $\mathcal{D}$ , the algorithm first extracts the block  $\mathcal{B}_{\text{cond}}^*$  based on the conditional statement and the permissible partition  $\bar{\pi}_{\mathcal{J}}$  as described above. The block  $\mathcal{B}_{\text{cond}}^*$  coarsens the input partition to yield the output partition. Finally, the original transformer is applied to the abstract element  $\mathcal{J}_{\text{cond}}$  associated with the block  $\mathcal{B}_{\text{cond}}^*$ ; the remaining constraints are kept as is in the result.

In Algorithm 3.1 the output of the decomposed transformer  $T_{\text{cond}}^{\text{d}}$  on input  $\mathcal{J}$  is computed as  $T_{\text{cond}}^{\text{d}}(\mathcal{J}) = T_{\text{cond}}(\mathcal{J}_{\text{cond}}) \cup \mathcal{J}_{\text{rest}}$ . One can show that  $T_{\text{cond}}^{\text{d}}$  is sound but we focus on also maintaining precision and thus monotonicity. Thus, we define a class  $\text{Cond}(\mathcal{D})$  of conditional transformers  $T_{\text{cond}}$  where  $\gamma(T_{\text{cond}}(\mathcal{J})) = \gamma(T_{\text{cond}}^{\text{d}}(\mathcal{J}))$  (this is one of the two conditions discussed earlier that ensure fixpoint equivalence).

**Definition 3.4.1.** A (sound and monotone) transformer  $T_{\text{cond}}$  for the conditional expression  $e \otimes c$  is in  $\text{Cond}(\mathcal{D})$  iff for any element  $\mathcal{J}$  and any associated permissible partition  $\bar{\pi}_{\mathcal{J}}$ , the output  $T_{\text{cond}}(\mathcal{J})$  satisfies:

- $T_{\text{cond}}(\mathcal{J}) = \mathcal{J} \cup \mathcal{J}' \cup \mathcal{J}''$  where  $\mathcal{J}'$  contains non-redundant constraints between the variables from  $\mathcal{B}_{\text{cond}}^*$  only and  $\mathcal{J}''$  is a set of redundant constraints between the variables in  $\mathcal{X}$ .
- $\gamma(T_{\text{cond}}(\mathcal{J}_{\text{cond}})) = \gamma(\mathcal{J}' \cup \mathcal{J}_{\text{cond}})$ .

**Theorem 3.4.1.** If  $T_{\text{cond}} \in \text{Cond}(\mathcal{D})$ , then  $\gamma(T_{\text{cond}}(\mathcal{J})) = \gamma(T_{\text{cond}}^{\text{d}}(\mathcal{J}))$  for all inputs  $\mathcal{J}$  in  $\mathcal{D}$ . In particular,  $T_{\text{cond}}^{\text{d}}$  is sound and monotone.

*Proof.*

$$\begin{aligned}
\gamma(T_{\text{cond}}(\mathcal{J})) &= \gamma(\mathcal{J} \cup \mathcal{J}' \cup \mathcal{J}'') && \text{(by Definition 3.4.1)} \\
&= \gamma(\mathcal{J} \cup \mathcal{J}') && (\mathcal{J}'' \text{ is redundant}) \\
&= \gamma(\mathcal{J}_{\text{rest}} \cup (\mathcal{J}_{\text{cond}} \cup \mathcal{J}')) && \text{(as } \mathcal{J} = \mathcal{J}_{\text{rest}} \cup \mathcal{J}_{\text{cond}}) \\
&= \gamma(\mathcal{J}_{\text{rest}}) \cap \gamma(\mathcal{J}_{\text{cond}} \cup \mathcal{J}') && (\gamma \text{ is meet-preserving}) \\
&= \gamma(\mathcal{J}_{\text{rest}}) \cap \gamma(T_{\text{cond}}(\mathcal{J}_{\text{cond}})) && \text{(by Definition 3.4.1)} \\
&= \gamma(\mathcal{J}_{\text{rest}} \cup T_{\text{cond}}(\mathcal{J}_{\text{cond}})) && (\gamma \text{ is meet-preserving}) \\
&= \gamma(T_{\text{cond}}^{\text{d}}(\mathcal{J})).
\end{aligned}$$

□

Note that we can strengthen the condition in Definition 3.4.1 by replacing  $\mathcal{B}_{\text{cond}}^*$  with  $\mathcal{B}_{\text{cond}}$ . This makes it independent of permissible partitions but would reduce the size of the class  $\text{Cond}(\mathcal{D})$ .

In Example 3.4.1,  $T_1 \in \text{Cond}(\mathcal{D})$  whereas  $T_2 \notin \text{Cond}(\mathcal{D})$  since  $T_2$  does not keep the original constraints. Most standard transformers used in practice satisfy the two conditions and can thus be decomposed with our construction without losing any precision. The following example illustrates our construction for decomposing the standard conditional transformer  $T_{\text{cond}}$  in the Octagon domain.

**Example 3.4.3.** Consider

$$\begin{aligned} \mathcal{X} &= \{x_1, x_2, x_3\}, \mathcal{L}_{\mathcal{X}, \text{Octagon}} : (3, \mathbb{U}^2 \times \{0\}, \{\leq, =\}, \mathbb{Q}), \\ \mathcal{J} &= \{x_1 \leq 0, x_2 + x_3 \leq 0\} \text{ with } \bar{\pi}_{\mathcal{J}} = \pi_{\mathcal{J}} = \{\{x_1\}, \{x_2, x_3\}\}. \end{aligned}$$

Consider the conditional statement  $x_3 \leq 0$  with  $\mathcal{B}_{\text{cond}} = \{x_3\}$ .  $T_{\text{cond}}$  adds the constraint  $x_3 \leq 0$  to  $\mathcal{J}$  and then applies Octagon closure on the resulting element to produce the output:

$$T_{\text{cond}}(\mathcal{J}) = \{x_1 \leq 0, x_2 + x_3 \leq 0, x_3 \leq 0, x_1 + x_3 \leq 0\},$$

which matches Definition 3.4.1 with  $\mathcal{J}' = \{x_3 \leq 0\}$  and  $\mathcal{J}'' = \{x_1 + x_3 \leq 0\}$ .

Algorithm 3.1 computes  $\mathcal{B}_{\text{cond}}^* = \{x_2, x_3\}$ ,  $\bar{\pi}_O = \{\{x_1\}, \{x_2, x_3\}\}$ ,  $\mathcal{J}_{\text{cond}} = \{x_2 + x_3 \leq 0\}$  and  $\mathcal{J}_{\text{rest}} = \{x_1 \leq 0\}$ . The algorithm applies  $T_{\text{cond}}$  on  $\mathcal{J}_{\text{cond}}$  and keeps  $\mathcal{J}_{\text{rest}}$  untouched to produce:

$$T_{\text{cond}}^d(\mathcal{J}) = \{x_1 \leq 0, x_2 + x_3 \leq 0, x_3 \leq 0\} \text{ with } \bar{\pi}_O.$$

Since  $\mathcal{J}' \cup \mathcal{J}_{\text{cond}} = T_{\text{cond}}(\mathcal{J}_{\text{cond}})$ ,  $T_{\text{cond}}$  satisfies the conditions for  $\text{Cond}(\mathcal{D})$  in this case and there is no change in precision.

Note that best transformers are not necessarily in  $\text{Cond}(\mathcal{D})$ . This is due to constraints on the coefficient set  $\mathcal{R}$  or the constant set  $\mathcal{C}$  in  $\mathcal{D}$ . We provide an example of a domain  $\mathcal{D}$  which does not have any best transformer in  $\text{Cond}(\mathcal{D})$ .

**Example 3.4.4.** We consider a fictive domain

$$\mathcal{X} = \{x_1, x_2\} \text{ and } \mathcal{L}_{\mathcal{X}, \mathcal{D}} : (2, \mathbb{Z}^2, \{\leq, =\}, \{0, 1, 1.5\}).$$

We assume  $\mathcal{J} = \{x_1 \leq 1, x_2 \leq 1\}$  with permissible partition  $\{\{x_1\}, \{x_2\}\}$  and the conditional  $x_2 \leq 0.5$ . In this case  $\mathcal{B}_{\text{cond}}^* = \{x_2\}$ . Using only the constraints with variables in  $\mathcal{B}_{\text{cond}}^*$  yields  $T_{\text{cond}}(\mathcal{J}_{\text{cond}}) = \{x_2 \leq 1\}$  as  $0.5 \notin \mathcal{C}$ . This means that the most precise result we can express which fits our conditions in the definition will be semantically equivalent to  $\mathcal{J}$ . However, a best transformer would produce an abstract element semantically equivalent to  $\{x_1 \leq 1, x_2 \leq 1, x_1 + x_2 \leq 1.5\}$ , which is more precise than  $\mathcal{J}$ . Thus, no best transformer is in  $\text{Cond}(\mathcal{D})$ .

We next identify domains for which a best conditional transformer is in  $\text{Cond}(\mathcal{D})$ .

**Theorem 3.4.2.** *A best conditional transformer  $T_{\text{cond}}^{\text{best}}$  of domain  $\mathcal{D}$  is in  $\text{Cond}(\mathcal{D})$  if for  $\mathcal{D} = (n, \mathcal{R}, \mathcal{T}, \mathcal{C})$  we have that  $\mathcal{R} = \mathcal{R}_1 \times \mathcal{R}_r \times \dots \times \mathcal{R}_n$  where each  $\mathcal{R}_i \subseteq \mathbb{Z}$  and  $0 \in \mathcal{R}_i$ ,  $\mathcal{T} = \{\leq, =\}$  and  $\mathcal{C} = \mathbb{R}$  or  $\mathbb{Q}$ .*

*Proof.* We show the proof for  $\mathcal{C} = \mathbb{R}$ . The extension to  $\mathcal{C} = \mathbb{Q}$  is straightforward. We first present an inefficient but correct construction of  $T_{\text{cond}}^{\text{best}}(\mathcal{J})$  for an element  $\mathcal{J} \in \mathcal{D}$  with the associated partition  $\bar{\pi}_{\mathcal{J}}$ . We note that the result of the concrete conditional transformer is  $\mathcal{J} \cup \{e \otimes c\}$ . We compute its abstraction in  $\mathcal{D}$  by initializing  $T_{\text{cond}}^{\text{best}}(\mathcal{J}) = \emptyset$  and computing an upper bound  $c_{\text{max}}$  for all representable expressions  $\sum_{i=1}^n \alpha'_i \cdot x_i$  in  $\mathcal{D}$  constrained under the set  $\mathcal{J} \cup \{e \otimes c\}$  using LP. If  $c_{\text{max}} \neq \infty$ , then  $\sum_{i=1}^n \alpha'_i \cdot x_i \leq c_{\text{max}}$  is added to  $T_{\text{cond}}^{\text{best}}(\mathcal{J})$ .

We now show that  $T_{\text{cond}}^{\text{best}}(\mathcal{J})$  satisfies Definition 3.4.1. We note that our construction will not remove any existing constraint from  $\mathcal{J}$  by adding a bound of  $\infty$ ; thus,  $\mathcal{J} \subseteq T_{\text{cond}}^{\text{best}}(\mathcal{J})$ . Since the constraint  $e \otimes c$  only affects the variables in  $\mathcal{B}_{\text{cond}}^*$  in the concrete, any constraint in  $T_{\text{cond}}^{\text{best}}(\mathcal{J})$  involving only the variables of  $\mathcal{X} \setminus \mathcal{B}_{\text{cond}}^*$  is either already in  $\mathcal{J}$  or is redundant in  $T_{\text{cond}}^{\text{best}}(\mathcal{J})$ .

Let us now consider a constraint  $\iota := \sum_{i=1}^{n_1} \alpha_i \cdot b_i + \sum_{i=1}^{n_2} \mu_i \cdot u_i \leq c$  in  $T_{\text{cond}}^{\text{best}}(\mathcal{J})$  where  $b_i \in \mathcal{B}_{\text{cond}}^*$ ,  $u_i \in \mathcal{X} \setminus \mathcal{B}_{\text{cond}}^*$ ,  $n_1, n_2 > 0$ ,  $n_1 + n_2 \leq n$  involving the variables of  $\mathcal{B}_{\text{cond}}^*$  and  $\mathcal{X} \setminus \mathcal{B}_{\text{cond}}^*$ . Since  $\mathcal{B}_{\text{cond}}^*$  and  $\mathcal{X} \setminus \mathcal{B}_{\text{cond}}^*$  are separate in  $\bar{\pi}_{\mathcal{J} \cup \{e \otimes c\}}$ ,  $\iota$  is redundant in  $\mathcal{J} \cup \{e \otimes c\}$ . There must exist constraints  $\iota_1 := \sum_{i=1}^{n_1} \alpha_i \cdot b_i \leq c_1$  and  $\iota_2 := \sum_{i=1}^{n_2} \mu_i \cdot u_i \leq c_2$  in  $\mathcal{J} \cup \{e \otimes c\}$  with  $c_1 + c_2 \leq c$ . Since  $\iota$  is representable in  $\mathcal{D}$ , the tuple  $(\alpha_1, \dots, \alpha_{n_1}, \mu_1, \dots, \mu_{n_2})$  is in  $\mathcal{R}$ . Since  $0 \in \mathcal{R}_i$ , the tuples  $(\alpha_1, \dots, \alpha_{n_1}, 0, \dots, 0)$  and  $(\mu_1, \dots, \mu_{n_2}, 0, \dots, 0)$  are also in  $\mathcal{R}$ . Since  $c_1, c_2 \in \mathcal{C} = \mathbb{R}$ , the constraints  $\iota_1$  and  $\iota_2$  are representable in  $\mathcal{D}$  and are therefore part of  $T_{\text{cond}}^{\text{best}}(\mathcal{J})$  which makes the constraint  $\iota$  redundant. Therefore, all non-redundant constraints added by  $T_{\text{cond}}^{\text{best}}$  are only between the variables in block  $\mathcal{B}_{\text{cond}}^*$ . Further, our construction ensures that these can be recovered by applying  $T_{\text{cond}}^{\text{best}}$  on  $\mathcal{J}_{\text{cond}}$ .  $\square$

We note that the algorithm for  $T_{\text{cond}}^{\text{best}}$  in Theorem 3.4.2 will not terminate if  $\mathcal{R}_i$  is infinite. Existing domains including Polyhedra, Octagon and Zone satisfy the conditions of Theorem 3.4.2 and thus a best conditional transformer in these domains is in  $\text{Cond}(\mathcal{D})$ . The following obvious corollary provides a condition under which the output partition  $\bar{\pi}_{\mathcal{J}_O}$  computed by Algorithm 3.1 is finest, i.e.,  $\bar{\pi}_{\mathcal{J}_O} = \pi_{\mathcal{J}_O}$ .

**Corollary 3.4.1.** For the conditional  $e \otimes c$ ,  $\bar{\pi}_{\mathcal{J}_O} = \pi_{\mathcal{J}_O}$ , if  $\bar{\pi}_{\mathcal{J}} = \pi_{\mathcal{J}}$  and  $\mathcal{J}_O = \mathcal{J} \cup \{e \otimes c\}$ .

### 3.4.2 Assignment

As in Section 2.1.2, we consider linear assignments of the form  $x_j := \delta$  on an abstract element  $\mathcal{J}$  with an associated permissible partition  $\bar{\pi}_{\mathcal{J}}$  in  $\mathcal{D}$  where  $\delta := \sum_{i=1}^n \alpha_i \cdot$

$x_i + c$  with  $a_i \in \mathbb{Z}$  and  $c \in \mathbb{Q}, \mathbb{R}$ . An assignment is *invertible* if  $a_j \neq 0$  (for example  $x_1 := x_1 + x_2$ ). We write  $\mathcal{J}_{x_j} \subseteq \mathcal{J}$  for the subset of constraints containing  $x_j$ .

As discussed in Section 3.1, a number of existing domains are not closed under assignment. As for the conditional, the best assignment transformers are usually expensive and may need to be approximated. We provide an example, very similar to Example 3.4.1, that shows how approximation can affect decomposition.

**Example 3.4.5.** Consider

$$\mathcal{X} = \{x_1, x_2, x_3, x_4, x_5, x_6\}, \mathcal{L}_{\mathcal{X}, \text{Polyhedra}} : (6, \mathbb{Z}^6, \{\leq, =\}, \mathbb{Q}),$$

$$\mathcal{J} = \{x_1 + x_2 = 0, x_3 + x_4 = 5, x_5 - x_3 = 0\} \text{ with } \bar{\pi}_{\mathcal{J}} = \pi_{\mathcal{J}} = \{\{x_1, x_2\}, \{x_3, x_4, x_5\}, \{x_6\}\}.$$

For the assignment  $x_5 := -x_6$ , a best sound assignment transformer  $T_1$  may return the decomposable output:

$$\mathcal{J}_O = \{x_1 + x_2 = 0, x_3 + x_4 = 5, x_5 + x_6 = 0\} \text{ with } \bar{\pi}_{\mathcal{J}_O} = \pi_{\mathcal{J}_O} = \{\{x_1, x_2\}, \{x_3, x_4\}, \{x_5, x_6\}\}.$$

However, another sound transformer  $T_2$  may return the non-decomposable output:

$$\mathcal{J}'_O = \{x_1 + x_2 + x_3 + x_4 + x_5 + x_6 = 5\} \text{ with } \bar{\pi}_{\mathcal{J}'_O} = \pi_{\mathcal{J}'_O} = \top.$$

Let  $\mathcal{B}_{\text{assign}} = \{x_i \mid a_i \neq 0\} \cup \{x_j\}$  be the set of variables affected by  $\delta := \sum_{i=1}^n a_i x_i + c$  (we also include  $x_j$ ). The block  $\mathcal{B}_{\text{assign}}^* = \bigcup_{\mathcal{X}_k \cap \mathcal{B}_{\text{assign}} \neq \emptyset} \mathcal{X}_k$  fuses all blocks  $\mathcal{X}_k \in \bar{\pi}_{\mathcal{J}}$  having non-empty intersection with  $\mathcal{B}_{\text{assign}}$ .

**Example 3.4.6.** Consider  $\mathcal{X} = \{x_1, x_2, x_3, x_4, x_5, x_6\}$  and an element  $\mathcal{J}$  in the Polyhedra domain with  $\bar{\pi}_{\mathcal{J}} = \{\{x_1, x_2\}, \{x_3, x_4\}, \{x_5, x_6\}\}$ . For the assignment  $x_3 := x_1 + x_2$ ,  $\mathcal{B}_{\text{assign}} = \{x_1, x_2, x_3\}$  and  $\mathcal{B}_{\text{assign}}^* = \{x_1, x_2, x_3, x_4\}$ .

We briefly explain the standard assignment transformer as background to motivate the later definition of  $\text{Assign}(\mathcal{D})$  (the class of assignment transformers that do not lose precision with our decomposition).

**STANDARD TRANSFORMER: INVERTIBLE ASSIGNMENT** The standard assignment transformer first removes all constraints in  $\mathcal{J}_{x_j}$  from  $\mathcal{J}$ . It then computes a set of constraints  $\mathcal{J}_{\text{inv}}$  by substituting, in all constraints in  $\mathcal{J}_{x_j}$ ,  $x_j$  with  $(x_j - \sum_{i \neq j} a_i x_i - c)/a_j$ . Finally,  $\mathcal{J}_{\text{inv}}$  may be approximated by a set  $\mathcal{J}'_{\text{inv}}$  of representable constraints (in  $\mathcal{D}$ ) over the same variable set. The result is  $\mathcal{J}_O = (\mathcal{J} \setminus \mathcal{J}_{x_j}) \cup \mathcal{J}'_{\text{inv}}$ .

**STANDARD TRANSFORMER: NON-INVERTIBLE ASSIGNMENT** For a non-invertible assignment, the transformer also first removes  $\mathcal{J}_{x_j}$  from  $\mathcal{J}$ . Next, it computes a set of constraints  $\mathcal{J}_{\text{non-inv}}$  by *projecting out*  $x_j$  from all constraints in  $\mathcal{J}_{x_j}$  using variable elimination [102]. Then it adds  $\{x_j - e = 0\}$  to  $\mathcal{J}_{\text{non-inv}}$ . Finally,  $\mathcal{J}_{\text{non-inv}}$  may be approximated by  $\mathcal{J}'_{\text{non-inv}}$  to make it representable in  $\mathcal{D}$  over the same variable set. The result is  $\mathcal{J}_O = (\mathcal{J} \setminus \mathcal{J}_{x_j}) \cup \mathcal{J}'_{\text{non-inv}}$ .

**Algorithm 3.2** Decomposed assignment transformer  $T_{\text{assign}}^d$ 


---

```

1: function ASSIGNMENT( $(\mathcal{J}, \bar{\pi}_{\mathcal{J}}), \text{stmt}, T_{\text{assign}}$ )
2:    $\mathcal{B}_{\text{assign}}^* := \text{compute\_block}(\text{stmt}, \bar{\pi}_{\mathcal{J}})$ 
3:    $\mathcal{J}_{\text{assign}} := \mathcal{J}(\mathcal{B}_{\text{assign}}^*)$ 
4:    $\bar{\pi}_{\mathcal{J}_O} := \{\mathcal{A} \in \bar{\pi}_{\mathcal{J}} \mid \mathcal{A} \cap \mathcal{B}_{\text{assign}}^* = \emptyset\} \cup \{\mathcal{B}_{\text{assign}}^*\}$ 
5:    $\mathcal{J}_{\text{rest}} := \mathcal{J}(\mathcal{X} \setminus \mathcal{B}_{\text{assign}}^*)$ 
6:    $\mathcal{J}_O := T_{\text{assign}}(\mathcal{J}_{\text{assign}}) \cup \mathcal{J}_{\text{rest}}$ 
7:    $\bar{\pi}_{\mathcal{J}_O} := \text{refine}(\bar{\pi}_{\mathcal{J}_O})$ 
8:   return  $(\mathcal{J}_O, \bar{\pi}_{\mathcal{J}_O})$ 
9: end function

```

---

**CONSTRUCTION FOR ASSIGNMENT** Algorithm 3.2 shows our construction for decomposing a given assignment transformer  $T_{\text{assign}}$ . It operates analogous to the decomposed conditional transformer, except for the partition refinement in line 7, which we explain at the end of this section. The output of the decomposed transformer  $T_{\text{assign}}^d$  on  $\mathcal{J}$  is  $T_{\text{assign}}^d(\mathcal{J}) = T_{\text{assign}}(\mathcal{J}_{\text{assign}}) \cup \mathcal{J}_{\text{rest}}$ . Next we define a class  $\text{Assign}(\mathcal{D})$  of assignment transformers  $T_{\text{assign}}$  where  $\gamma(T_{\text{assign}}(\mathcal{J})) = \gamma(T_{\text{assign}}^d(\mathcal{J}))$  for all  $\mathcal{J}$ . Again, this will ensure soundness and monotonicity.

**Definition 3.4.2.** A (sound and monotone) assignment transformer  $T$  in  $\mathcal{D}$  for the statement  $x_j := e$  is in  $\text{Assign}(\mathcal{D})$  iff for any element  $\mathcal{J}$  and any associated permissible partition  $\bar{\pi}_{\mathcal{J}}$  in  $\mathcal{D}$ , the output  $T_{\text{assign}}(\mathcal{J})$  satisfies the following conditions:

- $T_{\text{assign}}(\mathcal{J}) = (\mathcal{J} \setminus \mathcal{J}_{x_j}) \cup \mathcal{J}' \cup \mathcal{J}''$  where  $\mathcal{J}'$  contains non-redundant constraints between the variables from  $\mathcal{B}_{\text{assign}}^*$  only, and  $\mathcal{J}''$  is a set of redundant constraints between the variables in  $\mathcal{X}$ .
- $\gamma(T_{\text{assign}}(\mathcal{J}_{\text{assign}})) = \gamma((\mathcal{J}_{\text{assign}} \setminus \mathcal{J}_{x_j}) \cup \mathcal{J}')$ .

**Theorem 3.4.3.** If  $T_{\text{assign}} \in \text{Assign}(\mathcal{D})$ , then  $\gamma(T_{\text{assign}}(\mathcal{J})) = \gamma(T_{\text{assign}}^d(\mathcal{J}))$  for all inputs  $\mathcal{J}$  in  $\mathcal{D}$ . In particular,  $T_{\text{assign}}^d$  is sound and monotone.

*Proof.*  $\mathcal{B}_{\text{assign}}^*$  contains  $x_j$  by definition, thus  $\mathcal{J} \setminus \mathcal{J}_{x_j} = \mathcal{J}_{\text{rest}} \cup (\mathcal{J}_{\text{assign}} \setminus \mathcal{J}_{x_j})$ . We have,

$$\begin{aligned}
\gamma(T_{\text{assign}}(\mathcal{J})) &= \gamma((\mathcal{J} \setminus \mathcal{J}_{x_j}) \cup \mathcal{J}' \cup \mathcal{J}'') && \text{(by Definition 3.4.2)} \\
&= \gamma((\mathcal{J} \setminus \mathcal{J}_{x_j}) \cup \mathcal{J}') && (\mathcal{J}'' \text{ is redundant}) \\
&= \gamma(\mathcal{J}_{\text{rest}} \cup (\mathcal{J}_{\text{assign}} \setminus \mathcal{J}_{x_j}) \cup \mathcal{J}') && \text{(from above)} \\
&= \gamma(\mathcal{J}_{\text{rest}}) \cap \gamma((\mathcal{J}_{\text{assign}} \setminus \mathcal{J}_{x_j}) \cup \mathcal{J}') && (\gamma \text{ is meet-preserving}) \\
&= \gamma(\mathcal{J}_{\text{rest}}) \cap \gamma(T_{\text{assign}}(\mathcal{J}_{\text{assign}})) && \text{(by Definition 3.4.2)} \\
&= \gamma(\mathcal{J}_{\text{rest}} \cup T_{\text{assign}}(\mathcal{J}_{\text{assign}})) && (\gamma \text{ is meet-preserving}) \\
&= \gamma(T_{\text{assign}}^d(\mathcal{J}))
\end{aligned}$$

□

As for  $\text{Cond}(\mathcal{D})$ , Definition 3.4.2 can be tightened to make it independent of permissible partitions at the price of a smaller  $\text{Assign}(\mathcal{D})$ .

In Example 3.4.5,  $T_1 \in \text{Assign}(\mathcal{D})$  whereas  $T_2 \notin \text{Assign}(\mathcal{D})$  as  $T_2$  does not keep the constraints in  $\mathcal{J} \setminus \mathcal{J}_{x_5}$ . Most standard assignment transformers used in practice satisfy the two conditions and can thus be decomposed with our construction without losing any precision. The following example illustrates our construction for decomposing the standard assignment transformer  $T_{\text{assign}}$  in the TVPI domain.

**Example 3.4.7.** Consider

$$\begin{aligned} \mathcal{X} &= \{x_1, x_2, x_3\}, \mathcal{L}_{\mathcal{X}, \text{TVPI}} : (3, \mathbb{Z}^2 \times \{0\}, \{\leq, =\}, \mathbb{Q}), \\ \mathcal{J} &= \{x_1 \leq 0, x_2 + x_3 \leq 0, x_3 \leq 3\} \text{ with } \bar{\pi}_{\mathcal{J}} = \pi_{\mathcal{J}} = \{\{x_1\}, \{x_2, x_3\}\}. \end{aligned}$$

Consider the non-invertible assignment  $x_2 := 2 \cdot x_3$  with  $\mathcal{B}_{\text{assign}} = \{x_2, x_3\}$ .  $T_{\text{assign}}$  determines that  $\mathcal{J}_{x_2} = \{x_2 + x_3 \leq 0\}$ , projects out  $x_2$ , which yields the empty set, and then adds  $x_2 - 2 \cdot x_3 = 0$  to obtain  $\mathcal{J}_{\text{non-inv}}$ , which is representable. Overall this results in  $\{x_1 \leq 0, x_2 - 2 \cdot x_3 = 0, x_3 \leq 3\}$ .

Next, the transformer applies TVPI completion to produce the final output:

$$T_{\text{assign}}(\mathcal{J}) = \{x_1 \leq 0, x_2 - 2 \cdot x_3 = 0, x_3 \leq 3, x_2 \leq 6, x_1 + x_2 \leq 6, x_1 + x_3 \leq 3, x_2 + x_3 \leq 9\},$$

which has the form of Definition 3.4.2 with

$$\mathcal{J}' = \{x_2 - 2 \cdot x_3 = 0\} \text{ and } \mathcal{J}'' = \{x_2 \leq 6, x_1 + x_2 \leq 6, x_1 + x_3 \leq 3, x_2 + x_3 \leq 9\}.$$

Algorithm 3.2 computes  $\mathcal{B}_{\text{assign}}^* = \{x_2, x_3\}$ ,  $\mathcal{J}_{\text{assign}} = \{x_2 + x_3 \leq 0, x_3 \leq 3\}$ ,  $\bar{\pi}_{\mathcal{J}_0} = \{\{x_1\}, \{x_2, x_3\}\}$ , and  $\mathcal{J}_{\text{rest}} = \{x_1 \leq 0\}$ . The algorithm applies  $T_{\text{assign}}$  on  $\mathcal{J}_{\text{assign}}$  and keeps  $\mathcal{J}_{\text{rest}}$  untouched to produce:

$$\mathcal{J}_0 = T_{\text{assign}}^d(\mathcal{J}) = \{x_1 \leq 0, x_2 - 2 \cdot x_3 = 0, x_3 \leq 3, x_2 \leq 6, x_2 + x_3 \leq 9\} \text{ with } \bar{\pi}_{\mathcal{J}_0}.$$

Here  $\mathcal{J}'$  contains non-redundant constraints between variables from  $\mathcal{B}_{\text{assign}}^*$  only and we have  $\gamma((\mathcal{J}_{\text{assign}} \setminus \mathcal{J}_{x_j}) \cup \mathcal{J}') = \gamma(\{x_2 - 2 \cdot x_3 = 0, x_2 \leq 3\}) = \gamma(T_{\text{assign}}(\mathcal{J}_{\text{assign}}))$ . Thus  $T_{\text{assign}}$  satisfies the conditions for  $\text{Assign}(\mathcal{D})$  in this case and there is no loss of precision.

We next identify domains for which a best assignment transformer is in  $\text{Assign}(\mathcal{D})$ .

**Theorem 3.4.4.** *A best assignment transformer  $T_{\text{cond}}^{\text{best}}$  of domain  $\mathcal{D}$  is in  $\text{Assign}(\mathcal{D})$  if for  $\mathcal{D} = (n, \mathcal{R}, \mathcal{T}, \mathcal{C})$  we have that  $\mathcal{R} = \mathcal{R}_1 \times \mathcal{R}_r \times \dots \times \mathcal{R}_n$  where each  $\mathcal{R}_i \subseteq \mathbb{Z}$  and  $0 \in \mathcal{R}_i$ ,  $\mathcal{T} = \{\leq, =\}$  and  $\mathcal{C} = \mathbb{R}$  or  $= \mathbb{Q}$ .*

*Proof.* The result of the concrete assignment transformer is  $(\mathcal{J} \setminus \mathcal{J}_{x_j}) \cup \mathcal{J}_{\mathcal{B}_{\text{assign}}^*}$  where  $\mathcal{J}_{\mathcal{B}_{\text{assign}}^*}$  contains non-redundant constraints only between the variables in  $\mathcal{B}_{\text{assign}}^*$  added by the concrete transformer. Since  $\mathcal{J} \setminus \mathcal{J}_{x_j}$  is representable in  $\mathcal{D}$ ,  $T_{\text{assign}}^{\text{best}}$  can



be implemented by adding constraints from  $\mathcal{J}_{\mathcal{B}_{\text{assign}}^*}$  to  $\mathcal{J} \setminus \mathcal{J}_{x_j}$  via  $\mathsf{T}_{\text{cond}}^{\text{best}}$ . From Theorem 3.4.2, we know that  $\mathsf{T}_{\text{cond}}^{\text{best}}$  for  $\mathcal{D}$  is in  $\text{Cond}(\mathcal{D})$ . Thus  $\mathsf{T}_{\text{cond}}^{\text{best}}$  will add non-redundant constraints only between the variables in the block  $\mathcal{B}_{\text{assign}}^*$  to  $\mathcal{J} \setminus \mathcal{J}_{x_j}$ . These constraints can be obtained by applying  $\mathsf{T}_{\text{cond}}^{\text{best}}$  on  $\mathcal{J}_{\text{assign}} \setminus \mathcal{J}_{x_j}$ . Thus  $\mathsf{T}_{\text{assign}}^{\text{best}}$  is in  $\text{Assign}(\mathcal{D})$ .  $\square$

We note that most existing domains such as Polyhedra, Octagon, Zone, Octahedron satisfy the conditions of Theorem 3.4.4 and thus a best assignment transformer in these domains is in  $\text{Assign}(\mathcal{D})$ .

**REFINEMENT** So far we have assumed that line 7 of Algorithm 3.2 is the identity (that is, refinement does not affect  $\bar{\pi}_{\mathcal{J}_O}$ ). We now discuss refinement in more detail.

**Definition 3.4.3** (Refinement condition). The output partition  $\bar{\pi}_{\mathcal{J}_O}$  of a non-invertible assignment transformer  $\mathsf{T}_{\text{assign}}$  satisfying Definition 3.4.2 is a candidate for refinement if  $\mathcal{X}_t \cap \mathcal{B}_{\text{assign}} = \{x_j\}$  where  $\mathcal{X}_t$  is the block of  $\bar{\pi}_{\mathcal{J}}$  containing  $x_j$ . Here,  $\mathcal{J}$  is the abstract element upon which the transformer is applied and  $\bar{\pi}_{\mathcal{J}}$  is a permissible partition for  $\mathcal{J}$ .

If the above condition holds during analysis (it can be checked efficiently), then refinement can split  $\mathcal{B}_{\text{assign}}^*$  from  $\bar{\pi}_{\mathcal{J}_O}$  into two blocks  $\mathcal{X}_t \setminus \{x_j\}$  and  $\mathcal{B}_{\text{assign}}^* \setminus (\mathcal{X}_t \setminus \{x_j\})$ , provided no redundant constraint ( $\mathcal{J}''$  in Definition 3.4.2) fuses these two blocks. The result is a finer partition  $\bar{\pi}_{\mathcal{J}_O}$ . Note that our observation model for computing the output partition is more refined compared to Section 2.2.2 where we do not distinguish between invertible and non-invertible assignments and also do not check whether the condition  $\mathcal{X}_t \cap \mathcal{B}_{\text{assign}} = \{x_j\}$  holds.

**Example 3.4.8.** Consider

$$\begin{aligned} \mathcal{X} &= \{x_1, x_2, x_3, x_4, x_5\}, \mathcal{L}_{\mathcal{X}, \text{Zone}} : (5, \{1, 0\} \times \{0, -1\} \times \{0\}^3, \{\leq, =\}, \mathbb{Q}), \\ \mathcal{J} &= \{x_1 \leq x_2, x_2 \leq x_3, x_4 \leq x_5\} \text{ with } \bar{\pi}_{\mathcal{J}} = \pi_{\mathcal{J}} = \{\{x_1, x_2, x_3\}, \{x_4, x_5\}\}. \end{aligned}$$

Consider the non-invertible assignment  $x_2 := x_4$  with  $\mathcal{B}_{\text{assign}} = \{x_2, x_4\}$  and the standard Zone assignment transformer  $\mathsf{T}_{\text{assign}}$ . Without refinement, we obtain the partition  $\bar{\pi}_{\mathcal{J}_O} = \{\mathcal{X}\} = \top$ . However, our refinement condition enables us to obtain a finer output partition. We have that  $\mathcal{X}_t = \{x_1, x_2, x_3\}$  and  $\mathcal{X}_t \cap \mathcal{B}_{\text{assign}} = \{x_2\}$  and thus the dynamic refinement condition applies, splitting the block  $\mathcal{B}_{\text{assign}}^* = \mathcal{X}$  into two blocks:  $\mathcal{X}_t \setminus \{x_2\} = \{x_1, x_3\}$  and  $\mathcal{B}_{\text{assign}}^* \setminus (\mathcal{X}_t \setminus \{x_2\}) = \{x_2, x_4, x_5\}$ . This produces a finer partition for the output:

$$\mathcal{J}_O = \{x_1 \leq x_3, x_2 - x_4 = 0, x_4 \leq x_5, x_2 \leq x_5\} \text{ with } \bar{\pi}_{\mathcal{J}_O} = \pi_{\mathcal{J}_O} = \{\{x_1, x_3\}, \{x_2, x_4, x_5\}\}.$$

As with the conditional,  $\bar{\pi}_{\mathcal{J}_O} \neq \pi_{\mathcal{J}_O}$  in general even if  $\bar{\pi}_{\mathcal{J}} = \pi_{\mathcal{J}}$ . The following corollary provides conditions under which  $\bar{\pi}_{\mathcal{J}_O} = \pi_{\mathcal{J}_O}$  after applying Algorithm 3.2.

**Corollary 3.4.2.** For the assignment  $x_j := \delta$ ,  $\bar{\pi}_{\mathcal{J}_O} = \pi_{\mathcal{J}_O}$  holds if  $\bar{\pi}_{\mathcal{J}} = \pi_{\mathcal{J}}$  and, in the invertible case  $\mathcal{J}_O = (\mathcal{J} \setminus \mathcal{J}_{x_j}) \cup \mathcal{J}_{\text{inv}}$  or, in the non-invertible case  $\mathcal{J}_O = (\mathcal{J} \setminus \mathcal{J}_{x_j}) \cup \mathcal{J}_{\text{non-inv}}$ .

3.4.3 Meet ( $\sqcap$ )

As discussed in Section 3.1, all domains we consider are closed under the meet ( $\sqcap$ ) transformer and thus it is common to implement a best transformer. In fact, any meet transformer  $T_{\sqcap}$  that obeys  $T_{\sqcap}(\mathcal{J}, \mathcal{J}') \sqsubseteq \mathcal{J}, \mathcal{J}'$  is precise. Thus, we assume a given best meet transformer, i.e.,  $\gamma(T_{\sqcap}(\mathcal{J}, \mathcal{J}')) = \gamma(\mathcal{J}) \cap \gamma(\mathcal{J}')$  for all  $\mathcal{J}, \mathcal{J}'$ . As a consequence, our decomposed construction will always yield an equivalent transformer, without any conditions.

**CONSTRUCTION FOR MEET ( $\sqcap$ )** Algorithm 3.3 shows our construction of a decomposed transformer for a given meet transformer  $T_{\sqcap}$  on input elements  $\mathcal{J}, \mathcal{J}'$  with the respective permissible partitions  $\bar{\pi}_{\mathcal{J}}, \bar{\pi}_{\mathcal{J}'}$  in domain  $\mathcal{D}$ . The algorithm computes a common permissible partition  $\bar{\pi}_{\mathcal{J}} \sqcup \bar{\pi}_{\mathcal{J}'}$  for the inputs and then applies  $T_{\sqcap}$  separately on the individual factors of  $\mathcal{J}, \mathcal{J}'$  corresponding to the blocks in  $\bar{\pi}_{\mathcal{J}} \sqcup \bar{\pi}_{\mathcal{J}'}$ .

**Theorem 3.4.5.**  $\gamma(T_{\sqcap}(\mathcal{J}, \mathcal{J}')) = \gamma(T_{\sqcap}^d(\mathcal{J}, \mathcal{J}'))$  for all inputs  $\mathcal{J}, \mathcal{J}'$  in  $\mathcal{D}$ . In particular,  $T_{\sqcap}^d$  is sound and monotone.

*Proof.*

$$\begin{aligned}
\gamma(T_{\sqcap}(\mathcal{J}, \mathcal{J}')) &= \gamma(\mathcal{J}) \cap \gamma(\mathcal{J}') \\
&= \bigcap_{\mathcal{A}} \gamma(\mathcal{J}(\mathcal{A})) \cap \gamma(\mathcal{J}'(\mathcal{A})) && (\mathcal{J} = \bigcup \mathcal{J}(\mathcal{A}), \mathcal{J}' = \bigcup \mathcal{J}'(\mathcal{A})) \\
&= \bigcap_{\mathcal{A}} \gamma(T_{\sqcap}(\mathcal{J}(\mathcal{A}), \mathcal{J}'(\mathcal{A}))) && (T_{\sqcap} \text{ is precise}) \\
&= \gamma\left(\bigcup_{\mathcal{A}} T_{\sqcap}(\mathcal{J}(\mathcal{A}), \mathcal{J}'(\mathcal{A}))\right) && (\gamma \text{ is meet-preserving}) \\
&= \gamma(T_{\sqcap}^d(\mathcal{J}, \mathcal{J}'))
\end{aligned}$$

□

The following example illustrates the decomposition of a best meet transformer  $T_{\sqcap}$  in the Octahedron domain using Algorithm 3.3.

**Example 3.4.9.** Consider

$$\begin{aligned}
\mathcal{X} &= \{x_1, x_2, x_3, x_4\}, \mathcal{L}_{\mathcal{X}, \text{Octahedron}} = (4, \mathbf{U}^4, \{\leq, =\}, \mathbf{Q}), \\
\mathcal{J} &= \{x_1 \leq 1, x_2 \leq 0, x_3 + x_4 \leq 1\} \text{ with } \bar{\pi}_{\mathcal{J}} = \pi_{\mathcal{J}} = \{\{x_1\}, \{x_2\}, \{x_3, x_4\}\} \text{ and} \\
\mathcal{J}' &= \{x_1 - x_3 - x_4 \leq 2, x_2 \leq 1\} \text{ with } \bar{\pi}_{\mathcal{J}'} = \pi_{\mathcal{J}'} = \{\{x_1, x_3, x_4\}, \{x_2\}\}.
\end{aligned}$$

$T_{\sqcap}$  computes the union  $\mathcal{J} \cup \mathcal{J}'$  and then removes redundant constraints to produce the output:

$$T_{\sqcap}(\mathcal{J}, \mathcal{J}') = \{x_1 \leq 1, x_2 \leq 0, x_3 + x_4 \leq 1, x_1 - x_3 - x_4 \leq 2\}.$$

**Algorithm 3.3** Decomposed meet transformer  $T_{\sqcap}^d$ 


---

```

1: function MEET( $(J, \bar{\pi}_J), (J', \bar{\pi}_{J'}), T_{\sqcap}$ )
2:    $\bar{\pi}_{J_O} := \bar{\pi}_J \sqcup \bar{\pi}_{J'}$ 
3:    $J_O := \bigcup_{\mathcal{A} \in \bar{\pi}_{J_O}} T_{\sqcap}(J(\mathcal{A}), J'(\mathcal{A}))$ 
4:   return  $(J_O, \bar{\pi}_{J_O})$ 
5: end function

```

---

Algorithm 3.3 computes the common partition  $\bar{\pi}_J \sqcup \bar{\pi}_{J'} = \{\{x_1, x_3, x_4\}, \{x_2\}\}$  and applies  $T_{\sqcap}$  separately on the factors of  $J, J'$  corresponding to the common partition and produces:

$$T_{\sqcap}^d(J, J') = \{x_1 \leq 1, x_2 \leq 0, x_3 + x_4 \leq 1, x_1 - x_3 - x_4 \leq 2\}$$

with  $\bar{\pi}_{J_O} = \{\{x_1, x_3, x_4\}, \{x_2\}\}$ .

The following corollary provides conditions under which the output partition is finest.

**Corollary 3.4.3.**  $\bar{\pi}_{J_O} = \pi_{J_O}$  if  $\bar{\pi}_J = \pi_J, \bar{\pi}_{J'} = \pi_{J'}$ , and  $J_O = J \cup J'$ .

3.4.4 Join ( $\sqcup$ )

As discussed in Section 3.1, none of the sub-polyhedra domains we consider are closed for the join ( $\sqcup$ ). Thus, the join transformer approximates the union of  $J$  and  $J'$  in  $\mathcal{D}$  and is usually the most expensive transformer in  $\mathcal{D}$ . As with other transformers, an arbitrary approximation can result in the  $\top$  partition. The example below shows this situation with two sound join transformers in the Zone domain.

**Example 3.4.10.** Consider

$$\begin{aligned} \mathcal{X} &= \{x_1, x_2, x_3, x_4, x_5, x_6\}, \mathcal{L}_{\mathcal{X}, \text{Zone}} : (6, \{1, 0\} \times \{0, -1\} \times \{0\}^4, \{\leq, =\}, \mathbb{R}), \\ J &= \{x_1 = 1, x_2 = 2, x_3 \leq 3, x_4 = 4, x_5 = 0, x_6 = 0\} \text{ and} \\ J' &= \{x_1 = 1, x_2 = 2, x_3 \leq 3, x_4 = 4, x_5 = 1, x_6 = 1\} \text{ with} \\ \bar{\pi}_J &= \bar{\pi}_{J'} = \pi_J = \perp. \end{aligned}$$

A sound transformer  $T_1$  may return the decomposed output:

$$\begin{aligned} J_O &= \{x_1 = 1, x_2 = 2, x_3 \leq 3, x_4 = 4, -x_5 \leq 0, x_5 \leq 1, -x_6 \leq 0, x_6 \leq 1\} \text{ with} \\ \bar{\pi}_{J_O} &= \pi_{J_O} = \perp. \end{aligned}$$

Another sound transformer  $T_2$  for the join could produce the output  $J'_O$  with the  $\top$  partition:

$$\begin{aligned} J'_O &= \{x_2 - x_1 \leq 1, x_1 - x_5 \leq 1, x_3 - x_2 \leq 1, x_3 - x_4 \leq -1, x_5 - x_6 = 0\} \text{ with} \\ \bar{\pi}_{J'_O} &= \pi_{J'_O} = \top. \end{aligned}$$

**Algorithm 3.4** Decomposed join transformer  $T_{\sqcup}^d$ 


---

```

1: function JOIN( $(\mathcal{J}, \bar{\pi}_{\mathcal{J}}), (\mathcal{J}', \bar{\pi}_{\mathcal{J}'})$ ,  $T_{\sqcup}$ )
2:    $\bar{\pi}_{\text{common}} := \bar{\pi}_{\mathcal{J}} \sqcup \bar{\pi}_{\mathcal{J}'}$ 
3:    $\mathcal{N} = \bigcup \{\mathcal{A} \in \bar{\pi}_{\text{common}} \mid \mathcal{J}(\mathcal{A}) \neq \mathcal{J}'(\mathcal{A})\}$ 
4:    $\mathcal{J}_{\sqcup} := \mathcal{J}(\mathcal{N})$ 
5:    $\mathcal{J}'_{\sqcup} := \mathcal{J}'(\mathcal{N})$ 
6:    $\mathcal{J}_{\text{rest}} := \mathcal{J}(\mathcal{X} \setminus \mathcal{N})$ 
7:    $\bar{\pi}_{\mathcal{J}_O} := \{\mathcal{A} \in \bar{\pi}_{\text{common}} \mid \mathcal{A} \cap \mathcal{N} = \emptyset\} \cup \{\mathcal{N}\}$ 
8:    $\mathcal{J}_O := T_{\sqcup}(\mathcal{J}_{\sqcup}, \mathcal{J}'_{\sqcup}) \cup \mathcal{J}_{\text{rest}}$ 
9:    $\bar{\pi}_{\mathcal{J}_O} := \text{refine}(\bar{\pi}_{\mathcal{J}_O})$ 
10:  return  $(\mathcal{J}_O, \bar{\pi}_{\mathcal{J}_O})$ 
11: end function

```

---

Let  $\bar{\pi}_{\text{common}} := \bar{\pi}_{\mathcal{J}} \sqcup \bar{\pi}_{\mathcal{J}'}$  and  $\mathcal{N} = \bigcup \{\mathcal{A} \in \bar{\pi}_{\text{common}} \mid \mathcal{J}(\mathcal{A}) \neq \mathcal{J}'(\mathcal{A})\}$  be the union of all blocks for which the corresponding factors  $\mathcal{J}(\mathcal{A})$  and  $\mathcal{J}'(\mathcal{A})$  are not semantically equal. In Example 3.4.10, we have  $\mathcal{N} = \{\mathcal{x}_5, \mathcal{x}_6\}$ .

**CONSTRUCTION FOR  $\sqcup$**  Algorithm 3.4 shows our construction of a decomposed join transformer for a given  $T_{\sqcup}$  on input elements  $\mathcal{J}, \mathcal{J}'$  with the respective permissible partitions  $\bar{\pi}_{\mathcal{J}}, \bar{\pi}_{\mathcal{J}'}$  in domain  $\mathcal{D}$ . The algorithm first computes a common permissible partition  $\bar{\pi}_{\text{common}} = \bar{\pi}_{\mathcal{J}} \sqcup \bar{\pi}_{\mathcal{J}'}$ . For each block  $\mathcal{A} \in \bar{\pi}_{\text{common}}$ , it checks if the corresponding factors  $\mathcal{J}(\mathcal{A}), \mathcal{J}'(\mathcal{A})$  are (semantically) equal. If equality holds, the algorithm adds  $\mathcal{J}(\mathcal{A})$  to the output  $\mathcal{J}_O$  and adds the corresponding block  $\mathcal{A}$  to the partition  $\bar{\pi}_{\mathcal{J}_O}$ . Those not equal are collected (using Algorithm 2.1) in the bigger factors  $\mathcal{J}_{\sqcup}, \mathcal{J}'_{\sqcup}$  on which  $T_{\sqcup}$  is applied, which reduces complexity. The associated block in  $\bar{\pi}_{\mathcal{J}_O}$  is  $\mathcal{N}$ . The partition refinement in line 9 of Algorithm 3.4 is explained at the end of this section.

In Algorithm 3.4 the output of the decomposed transformer  $T_{\sqcup}^d$  on inputs  $\mathcal{J}, \mathcal{J}'$  is computed as  $T_{\sqcup}^d(\mathcal{J}, \mathcal{J}') = T_{\sqcup}(\mathcal{J}_{\sqcup}, \mathcal{J}'_{\sqcup}) \cup \mathcal{J}_{\text{rest}}$ . Next we define a class  $\text{Join}(\mathcal{D})$  of join transformers  $T_{\sqcup}$  for which  $\gamma(T_{\sqcup}(\mathcal{J}, \mathcal{J}')) = \gamma(T_{\sqcup}^d(\mathcal{J}, \mathcal{J}'))$  for all inputs  $\mathcal{J}, \mathcal{J}'$  in  $\mathcal{D}$ . This ensures soundness and monotonicity.

**Definition 3.4.4.** A join transformer  $T_{\sqcup}$  is in  $\text{Join}(\mathcal{D})$  iff for all pairs of input elements  $\mathcal{J}, \mathcal{J}'$  and all associated common permissible partitions  $\bar{\pi}_{\text{common}}$ , the output  $T_{\sqcup}(\mathcal{J}, \mathcal{J}')$  satisfies the following conditions:

- $T_{\sqcup}(\mathcal{J}, \mathcal{J}') = \mathcal{J}_{\text{rest}} \cup \mathcal{J}' \cup \mathcal{J}''$  where  $\mathcal{J}_{\text{rest}} = \mathcal{J}(\mathcal{X} \setminus \mathcal{N})$ ,  $\mathcal{J}'$  contains non-redundant constraints between only the variables from  $\mathcal{N}$  and  $\mathcal{J}''$  contains redundant constraints between the variables in  $\mathcal{X}$ .
- $\gamma(T_{\sqcup}(\mathcal{J}_{\sqcup}, \mathcal{J}'_{\sqcup})) = \gamma(\mathcal{J}')$ .

**Theorem 3.4.6.** If  $T_{\sqcup} \in \text{Join}(\mathcal{D})$ , then  $\gamma(T_{\sqcup}(\mathcal{J}, \mathcal{J}')) = \gamma(T_{\sqcup}^d(\mathcal{J}, \mathcal{J}'))$  for all inputs  $\mathcal{J}, \mathcal{J}'$  in  $\mathcal{D}$ . In particular,  $T_{\sqcup}^d$  is sound and monotone.

*Proof.*

$$\begin{aligned}
\gamma(\mathsf{T}_{\sqcup}(\mathcal{J}, \mathcal{J}')) &= \gamma(\mathcal{J}_{\text{rest}} \cup \mathcal{J}' \cup \mathcal{J}'') && \text{(by Definition 3.4.4)} \\
&= \gamma(\mathcal{J}_{\text{rest}} \cup \mathcal{J}') && (\mathcal{J}'' \text{ is redundant}) \\
&= \gamma(\mathcal{J}_{\text{rest}}) \cap \gamma(\mathcal{J}') && (\gamma \text{ is meet-preserving}) \\
&= \gamma(\mathcal{J}_{\text{rest}}) \cap \gamma(\mathsf{T}_{\sqcup}(\mathcal{J}_{\sqcup}, \mathcal{J}'_{\sqcup})) && \text{(by Definition 3.4.4)} \\
&= \gamma(\mathcal{J}_{\text{rest}} \cup \mathsf{T}_{\sqcup}(\mathcal{J}_{\sqcup}, \mathcal{J}'_{\sqcup})) && (\gamma \text{ is meet-preserving}) \\
&= \gamma(\mathsf{T}_{\sqcup}^{\text{d}}(\mathcal{J}, \mathcal{J}'))
\end{aligned}$$

□

The join transformer  $\mathsf{T}_1$  in Example 3.4.10 is in  $\text{Join}(\mathcal{D})$  whereas  $\mathsf{T}_2$  is not in  $\text{Join}(\mathcal{D})$  as it does not keep the constraints in  $\mathcal{J}_{\text{rest}}$ . Most standard and best join transformers in practice satisfy the conditions for  $\text{Join}(\mathcal{D})$  and thus they are decomposable with our construction without any change in precision.

The following example illustrates the decomposition of a best join transformer  $\mathsf{T}_{\sqcup}$  in the Octagon domain using Algorithm 3.4.

**Example 3.4.11.** Consider

$$\begin{aligned}
\mathcal{X} &= \{x_1, x_2, x_3\}, \mathcal{L}_{\mathcal{X}, \text{Octagon}} : (3, \mathbf{U}^2 \times \{0\}, \{\leq, =\}, \mathbb{R}), \\
\mathcal{J} &= \{x_1 \leq 2, x_2 \leq 1, x_3 \leq 3\}, \mathcal{J}' = \{x_1 \leq 1, x_2 \leq 3, x_3 \leq 3\} \text{ with} \\
\bar{\pi}_{\mathcal{J}} &= \bar{\pi}_{\mathcal{J}'} = \pi_{\mathcal{J}} = \perp.
\end{aligned}$$

$\mathsf{T}_{\sqcup}$  performs strong closure on both  $\mathcal{J}, \mathcal{J}'$  to produce:

$$\begin{aligned}
\mathcal{J}^* &= \{x_1 \leq 2, x_2 \leq 1, x_3 \leq 3, x_1 + x_2 \leq 3, x_1 + x_3 \leq 5, x_2 + x_3 \leq 4\} \\
\mathcal{J}'^* &= \{x_1 \leq 1, x_2 \leq 3, x_3 \leq 3, x_1 + x_2 \leq 4, x_1 + x_3 \leq 4, x_2 + x_3 \leq 6\}.
\end{aligned}$$

It then takes the pairwise maximum of bounds for each constraint to produce the output:

$$\mathsf{T}_{\sqcup}(\mathcal{J}, \mathcal{J}') = \{x_1 \leq 2, x_2 \leq 3, x_3 \leq 3, x_1 + x_2 \leq 4, x_1 + x_3 \leq 5, x_2 + x_3 \leq 6\}.$$

which matches Definition 3.4.4 with

$$\mathcal{J}_{\text{rest}} = \{x_3 \leq 3\}, \mathcal{J}' = \{x_1 \leq 2, x_2 \leq 3, x_1 + x_2 \leq 4\}, \text{ and } \mathcal{J}'' = \{x_1 + x_3 \leq 5, x_2 + x_3 \leq 6\}.$$

Since  $\bar{\pi}_{\mathcal{J}} = \bar{\pi}_{\mathcal{J}'}$ , we have  $\bar{\pi}_{\text{common}} = \bar{\pi}_{\mathcal{J}}$ . Here  $\mathcal{J}_1 \neq \mathcal{J}'_1$ ,  $\mathcal{J}_2 \neq \mathcal{J}'_2$  and  $\mathcal{J}_3 = \mathcal{J}'_3$ . Algorithm 3.4 computes  $\mathcal{N} = \{x_1, x_2\}$  and combines  $\mathcal{J}_1, \mathcal{J}_2$  into a single factor  $\mathcal{J}_{\sqcup}$  using Algorithm 2.1. Similarly, it combines  $\mathcal{J}'_1, \mathcal{J}'_2$  into  $\mathcal{J}'_{\sqcup}$ .

$$\mathcal{J}_{\sqcup} = \{x_1 \leq 2, x_2 \leq 1\}, \mathcal{J}'_{\sqcup} = \{x_1 \leq 1, x_2 \leq 3\}.$$

The algorithm applies  $\mathsf{T}_{\sqcup}$  only on  $\mathcal{J}_{\sqcup}, \mathcal{J}'_{\sqcup}$  whereas  $\mathcal{J}_3$  is added to the output directly:

$$\mathcal{J}_{\text{O}} = \mathsf{T}_{\sqcup}^{\text{d}}(\mathcal{J}, \mathcal{J}') = \{x_1 \leq 2, x_2 \leq 3, x_1 + x_2 \leq 4, x_3 \leq 3\} \text{ with } \bar{\pi}_{\mathcal{J}_{\text{O}}} = \pi_{\mathcal{J}_{\text{O}}} = \{\{x_1, x_2\}, \{x_3\}\}.$$

In the example,  $\mathcal{J}_{\text{rest}}$  contains non-redundant constraints only between the variables from  $\mathcal{X} \setminus \mathcal{N}$ ,  $\mathcal{J}'$  contains non-redundant constraints between only the variables from  $\mathcal{N}$  and  $\gamma(\mathsf{T}_{\sqcup}(\mathcal{J}_{\sqcup}, \mathcal{J}'_{\sqcup})) = \{x_1 \leq 2, x_2 \leq 3, x_1 + x_2 \leq 4\} = \gamma(\mathcal{J}')$ , and thus  $\mathsf{T}_{\sqcup}$  satisfies the conditions for  $\text{Join}(\mathcal{D})$  in this case.

**REFINEMENT** We can sometimes refine the output partition  $\bar{\pi}_{\mathcal{J}_O}$  after computing the output  $\mathcal{J}_O$  without inspecting or modifying  $\mathcal{J}_O$ . Namely, if a variable  $x_i$  is unconstrained in either  $\mathcal{J}$  or  $\mathcal{J}'$ , then it is also unconstrained in  $\mathcal{J}_O$ .  $\bar{\pi}_{\mathcal{J}_O}$  can thus be refined by removing  $x_i$  from the block containing it and adding the singleton set  $\{x_i\}$  to  $\bar{\pi}_{\mathcal{J}_O}$ . This refinement can be performed after applying  $T_{\square}^d$ . Note that our observation model in Section 2.2.2 did not check if the variables were unconstrained for polyhedra inputs and thus it produces coarser partitions. The following theorem formalizes this refinement.

**Theorem 3.4.7.** *Let  $\mathcal{J}, \mathcal{J}'$  be abstract elements in  $\mathcal{D}$  with the associated permissible partitions  $\bar{\pi}_{\mathcal{J}}, \bar{\pi}_{\mathcal{J}'}$ , respectively. Let  $\mathcal{U} = \{x \in \mathcal{X} \mid x \text{ is unconstrained in } \mathcal{J} \text{ or } \mathcal{J}'\} = \{u_1, \dots, u_r\}$  and let  $\bar{\pi}_{\mathcal{J}_O}$  as computed in line 7 of Algorithm 3.4. Then the following partition is permissible for the output  $\mathcal{J}_O$ :*

$$\bar{\pi}_{\mathcal{J}_O} \sqcap \{\mathcal{X} \setminus \mathcal{U}, \{u_1\}, \dots, \{u_r\}\}.$$

The proof of Theorem 3.4.7 is immediate from the discussion above. Unlike other transformers, we do not know of conditions for checking whether  $\bar{\pi}_{\mathcal{J}_O} = \pi_{\mathcal{J}_O}$ .

### 3.4.5 Widening ( $\nabla$ )

The widening transformer  $T_{\nabla}$  is applied during analysis to accelerate convergence towards a fixpoint. It is a binary transformer that guarantees: (i) the output satisfies  $T_{\nabla}(\mathcal{J}, \mathcal{J}') \sqsupseteq \mathcal{J}, \mathcal{J}'$ , and (ii) the analysis terminates after a finite number of steps. In general, widening transformers are not monotonic or commutative. Further, best widening transformers do not exist for any numerical domain. In theory, it may be possible to design arbitrary widening transformers that always result in the  $\top$  partition. In practice, the standard widening transformers are of two types:

**SYNTACTIC** For syntactic widening [140, 142], the output satisfies  $\mathcal{J}_O \subseteq \mathcal{J}$ . A constraint  $\iota := \sum_{i=1}^n a_i \cdot x_i \leq c \in \mathcal{J}$  is in the output  $\mathcal{J}_O$  iff there is a constraint  $\iota' := \sum_{i=1}^n a_i \cdot x_i \leq c' \in \mathcal{J}'$  with the same linear expression and  $c' \leq c$ .

**SEMANTIC** The semantic widening [13] requires the set of constraints in the input  $\mathcal{J}$  to be non-redundant. The output satisfies  $\mathcal{J}_O \subseteq \mathcal{J} \cup \mathcal{J}'$ . Specifically,  $\mathcal{J}_O$  contains the constraints from  $\mathcal{J}$  that are satisfied by  $\mathcal{J}'$  and the constraints  $\iota'$  from  $\mathcal{J}'$  that are mutually redundant with a constraint  $\iota$  in  $\mathcal{J}$ .

Both these transformers are decomposable in practice. The following example illustrates the semantic and the syntactic widening on the Octagon domain.

**Example 3.4.12.** Consider

$$\mathcal{X} = \{x_1, x_2, x_3, x_4\}, \mathcal{L}_{\mathcal{X}, \text{Octagon}} : (4, \mathbb{U}^2 \times \{0\}^2, \{\leq, =\}, \mathbb{Q}),$$

$$\mathcal{J} = \{x_1 - x_2 = 0, x_2 = 0, x_3 \leq 0, x_4 \leq 1\} \text{ with } \bar{\pi}_{\mathcal{J}} = \pi_{\mathcal{J}} = \{\{x_1, x_2\}, \{x_3\}, \{x_4\}\} \text{ and}$$

$$\mathcal{J}' = \{x_1 = 0, x_3 + x_4 \leq 2\} \text{ with } \bar{\pi}_{\mathcal{J}'} = \pi_{\mathcal{J}'} = \{\{x_1\}, \{x_2\}, \{x_3, x_4\}\}.$$

---

**Algorithm 3.5** Decomposed widening transformer  $T_{\nabla}^d$ 


---

```

1: function WIDENING( $(J, \bar{\pi}_J), (J', \bar{\pi}_{J'}), T_{\nabla}$ )
2:    $\bar{\pi}_{J_O} := \bar{\pi}_J \sqcup \bar{\pi}_{J'}$ 
3:    $J_O := \bigcup_{\mathcal{A} \in \bar{\pi}_{J_O}} T_{\nabla}(J(\mathcal{A}), J'(\mathcal{A}))$ 
4:    $\bar{\pi}_{J_O} := \text{refine}(\bar{\pi}_{J_O})$ 
5:   return  $(J_O, \bar{\pi}_{J_O})$ 
6: end function

```

---

The semantic widening transformer  $T_1$  yields:

$$J_O = \{x_1 = 0\} \text{ with } \bar{\pi}_{J_O} = \pi_{J_O} = \perp.$$

On the other hand, the syntactic widening transformer  $T_2$  yields:

$$J'_O = \emptyset \text{ with } \bar{\pi}_{J'_O} = \pi_{J'_O} = \perp.$$

Thus, both are decomposable in this case.

**CONSTRUCTION FOR WIDENING** Algorithm 3.5 shows our construction of a decomposed widening transformer on input elements  $J, J'$  with respective permissible partitions  $\bar{\pi}_J, \bar{\pi}_{J'}$  in  $\mathcal{D}$ . The algorithm computes a common permissible partition  $\bar{\pi}_J \sqcup \bar{\pi}_{J'}$  and then applies the widening transformer  $T_{\nabla}$  separately on the individual factors of  $J, J'$  corresponding to the blocks of  $\bar{\pi}_J \sqcup \bar{\pi}_{J'}$ . The refinement of the output partition in line 4 of Algorithm 3.5 is explained at the end of this section.

Next, we define a class  $\text{Widen}(\mathcal{D})$  of widening transformers  $T_{\nabla}$  for which it holds that  $\gamma(T_{\nabla}(J, J')) = \gamma(T_{\nabla}^d(J, J'))$  for all inputs  $J, J'$  in  $\mathcal{D}$ .

**Definition 3.4.5.** A widening transformer  $T_{\nabla}$  is in  $\text{Widen}(\mathcal{D})$  iff for all pairs of input elements  $J, J'$  and all associated common permissible partitions  $\bar{\pi}_{\text{common}}$ , the output  $T_{\nabla}(J, J')$  satisfies:

$$\gamma(T_{\nabla}(J, J')) = \bigcap_{\mathcal{A}} \gamma(T_{\nabla}(J(\mathcal{A}), J'(\mathcal{A}))).$$

**Theorem 3.4.8.** If  $T_{\nabla} \in \text{Widen}(\mathcal{D})$ , then  $\gamma(T_{\nabla}(J, J')) = \gamma(T_{\nabla}^d(J, J'))$  for all inputs  $J, J'$  in  $\mathcal{D}$ . Thus,  $T_{\nabla}^d$  is sound.

*Proof.*

$$\begin{aligned}
\gamma(T_{\nabla}(J, J')) &= \bigcap_{\mathcal{A}} \gamma(T_{\nabla}(J(\mathcal{A}), J'(\mathcal{A}))) && \text{(by Definition 3.4.5)} \\
&= \gamma\left(\bigcup_{\mathcal{A}} T_{\nabla}(J(\mathcal{A}), J'(\mathcal{A}))\right) && (\gamma \text{ is meet-preserving}) \\
&= \gamma(T_{\nabla}^d(J, J'))
\end{aligned}$$

□

Both syntactic and semantic Octagon widening transformers from Example 3.4.12 are in  $\text{Widen}(\mathcal{D})$ . It can be shown that the standard transformers in existing domains are in  $\text{Widen}(\mathcal{D})$ . For syntactic widening,  $\gamma(\mathcal{J}) = \gamma(\mathcal{J}')$  does not imply  $\gamma(\text{T}_{\nabla}(\mathcal{J}'', \mathcal{J})) = \gamma(\text{T}_{\nabla}(\mathcal{J}'', \mathcal{J}'))$  in general, and thus fixpoint equivalence is not guaranteed with the corresponding decomposed transformer. The following example illustrates the decomposition of the standard semantic TVPI widening transformer  $\text{T}_{\nabla}$  using Algorithm 3.5.

**Example 3.4.13.** Consider

$$\mathcal{X} = \{x_1, x_2, x_3, x_4\}, \mathcal{L}_{\mathcal{X}, \text{TVPI}} = (\mathbb{Z}^2 \times \{0\}^2, \{\leq, =\}, \mathbb{Q}),$$

$$\mathcal{J} = \{x_1 = 1, x_2 = 0, x_3 + x_4 \leq 1\}, \text{ with } \bar{\pi}_{\mathcal{J}} = \{\{x_1\}, \{x_2\}, \{x_3, x_4\}\} \text{ and}$$

$$\mathcal{J}' = \{2 \cdot x_1 - 3 \cdot x_2 \leq 2, x_1 + x_2 = 1, x_3 \leq 0, x_4 \leq 0\} \text{ with } \bar{\pi}_{\mathcal{J}'} = \{\{x_1, x_2\}, \{x_3\}, \{x_4\}\}.$$

$\text{T}_{\nabla}$  keeps the constraint  $x_3 + x_4 \leq 1$  from  $\mathcal{J}$  as it is satisfied by  $\mathcal{J}'$  (using  $x_3 \leq 0, x_4 \leq 0$ ). It also adds the constraint  $x_1 + x_2 = 1$  from  $\mathcal{J}'$  to the output as it is mutually redundant with the constraint  $x_1 = 1$  in  $\mathcal{J}$ . The output of  $\text{T}_{\nabla}$  is:

$$\text{T}_{\nabla}(\mathcal{J}, \mathcal{J}') = \{x_1 + x_2 = 1, x_3 + x_4 \leq 1\}.$$

Algorithm 3.5 computes the common permissible partition  $\mathcal{J}_0 = \bar{\pi}_{\mathcal{J}} \sqcup \bar{\pi}_{\mathcal{J}'} = \{\{x_1, x_2\}, \{x_3, x_4\}\}$  and then computes the output  $\mathcal{J}_0$  by applying  $\text{T}_{\nabla}$  separately on the individual factors of  $\mathcal{J}, \mathcal{J}'$  corresponding to the blocks of  $\bar{\pi}_{\text{common}}$ :

$$\mathcal{J}_0 = \text{T}_{\nabla}^d(\mathcal{J}, \mathcal{J}') = \{x_1 + x_2 = 1, x_3 + x_4 \leq 1\} \text{ with } \bar{\pi}_{\mathcal{J}_0} = \{\{x_1, x_2\}, \{x_3, x_4\}\}.$$

Here  $\gamma(\text{T}_{\nabla}(\mathcal{J}, \mathcal{J}')) = \bigcap_{\mathcal{A}} \gamma(\text{T}_{\nabla}(\mathcal{J}(\mathcal{A}), \mathcal{J}'(\mathcal{A})))$  and thus  $\text{T}_{\nabla}$  is in  $\text{Widen}(\mathcal{D})$ .

**REFINEMENT**  $\text{T}_{\nabla}$  in Algorithm 3.5 does not create constraints between variables in different blocks of the common partition in the output  $\mathcal{J}_0$ . By construction  $\bar{\pi}_{\mathcal{J}_0} = \bar{\pi}_{\text{common}} = \bar{\pi}_{\mathcal{J}} \sqcup \bar{\pi}_{\mathcal{J}'}$ . For syntactic widening,  $\mathcal{J}_0 \subseteq \mathcal{J}$  and thus the output partition  $\bar{\pi}_{\mathcal{J}_0}$  can be refined to  $\bar{\pi}_{\mathcal{J}}$  after computing the output  $\mathcal{J}_0$ .

The following corollaries provide conditions when  $\bar{\pi}_{\mathcal{J}_0} = \pi_{\mathcal{J}_0}$  for the semantic and the syntactic widening respectively.

**Corollary 3.4.4.** For semantic widening,  $\bar{\pi}_{\mathcal{J}_0} = \pi_{\mathcal{J}_0}$  if  $\bar{\pi}_{\mathcal{J}} = \pi_{\mathcal{J}}, \bar{\pi}_{\mathcal{J}'} = \pi_{\mathcal{J}'}$  and  $\mathcal{J}_0 = \mathcal{J} \cup \mathcal{J}'$ .

**Corollary 3.4.5.** For syntactic widening,  $\bar{\pi}_{\mathcal{J}_0} = \pi_{\mathcal{J}_0}$  if  $\bar{\pi}_{\mathcal{J}} = \pi_{\mathcal{J}}$  and  $\mathcal{J}_0 = \mathcal{J}$ .

### 3.5 EXPERIMENTAL EVALUATION

In this section we evaluate the performance of our generic decomposition approach on three popular domains: Polyhedra, Octagon, and Zone. Using standard implementations of these domains, we show that our decomposition of their transformers leads to substantial performance improvements, often surpassing existing transformers designed for specific domains.



Our decomposed implementation for these domains is available as part (i.e., an update) of the ELINA library [1]. Below, we compare to the prior ELINA as described in [189, 190].

**EXPERIMENTAL SETUP** We used the same setup as in Section 2.5.

**BENCHMARKS** The benchmarks were taken from the popular software verification competition [24]. The benchmark suite is divided into categories suited for different kinds of analyses, e.g., pointer, array, numerical, and others. We chose two categories suited for numerical analysis: (i) Linux Device Drivers (LD), and (ii) Control Flow (CF). Each of these categories contains hundreds of benchmarks and we evaluated the performance of our analysis on each of these. We use the `crab-llvm` analyzer, part of the SeaHorn verification framework [91], for performing the analysis as in Section 2.5 but a different version. Therefore our reported numbers for the baselines are different than in Table 2.3.

### 3.5.1 *Polyhedra*

The standard implementation of the Polyhedra domain contains the best conditional, assignment, meet, and join transformers together with a semantic widening transformer (as described in Chapter 2). All these transformers are in the classes of decomposable transformers defined in Section 3.4.

We refer the reader to Table 2.2 and Table 2.1 for the asymptotic complexity of the Polyhedra transformers in the standard implementation with and without decomposition [190] respectively. We compare the runtime and memory consumption for end-to-end Polyhedra analysis with our generic decomposed transformers versus the original non-decomposed transformers from the Parma Polyhedra Library (PPL) [12] and the decomposed transformers from ELINA [190] presented in Chapter 2. Note that we do not compare against NewPolka [104] as it performed worse than PPL in our previous evaluation in Section 2.5. PPL, ELINA, and our implementation store the constraints and the generators using matrices with 64-bit integers. PPL stores a single matrix for either representation whereas both ELINA and our implementation use a set of matrices corresponding to the factors, which requires exponential space in the worst case.

Table 3.2 shows the results on 13 large, representative benchmarks. These benchmarks were chosen based on the following criteria which is similar to the one in Section 2.5:

- The most time consuming function in the benchmark did not produce any integer overflow with ELINA or our approach.
- The benchmark ran for at least 2 minutes with PPL.

Our decomposition maintains semantic equivalence with both ELINA and PPL as long as there is no integer overflow and thus gets the same semantic invariants. All three implementations set the abstract element to  $\top$  when an integer overflow occurs. The total number of integer overflows on the chosen benchmarks were 58, 23 and 21 for PPL, ELINA, and our decomposition, respectively. We also had fewer integer overflows than both ELINA and PPL on the remaining benchmarks. Thus, our decomposition improves in some cases also the precision of the analysis with respect to both ELINA and PPL.

Table 3.2: Speedup for the Polyhedra analysis with our decomposition vs. PPL and ELINA.

Benchmark	PPL		ELINA		Our Decomposition		Speedup vs.	
	time(s)	memory(GB)	time(s)	memory(GB)	time(s)	memory(GB)	PPL	ELINA
firewire_firedtv	331	0.9	0.4	0.2	0.2	0.2	1527	2
net_fddi_skfp	6142	7.2	9.2	0.9	4.4	0.3	1386	2
mtd_ubi	M0	M0	4	0.9	1.9	0.3	$\infty$	2.1
usb_core_main0	4003	1.4	65	2	29	0.7	136	2.2
tty_synclinkmp	M0	M0	3.4	0.1	2.5	0.1	$\infty$	1.4
scsi_advansys	T0	T0	4	0.4	3.4	0.2	>4183	1.2
staging_vt6656	T0	T0	2	0.4	0.5	0.1	>28800	4
net_ppp	10530	0.1	924	0.3	891	0.1	11.8	1
p10_l00	121	0.9	11	0.8	5.4	0.2	22.4	2
p16_l40	M0	M0	11	3	2.9	0.4	$\infty$	3.8
p12_l57	M0	M0	14	0.8	6.5	0.3	$\infty$	2.1
p13_l53	M0	M0	54	2.7	25	0.9	$\infty$	2.2
p19_l59	M0	M0	70	1.7	12	0.6	$\infty$	5.9

Table 3.2 shows our experimental findings. The entry M0 and T0 in the table have the same meaning as in Table 2.3. We follow the same convention of reporting speedups in the case of a memory overflow or a time out as in Table 2.3.

In the table, PPL either ran out of memory or did not finish within four hours on 8 out of the 13 benchmarks. Both ELINA and our decomposition are able to analyze all benchmarks. We are faster than ELINA on all benchmarks with a maximum speedup of 5.9x on the P19\_l59 benchmark. We also save significant memory over ELINA. The speedups on the remaining (not shown) benchmarks over the decomposed version of ELINA varies from 1.1x to 4x with an average of  $\approx 1.4x$ .

**BETTER PARTITIONING LEADS TO PERFORMANCE IMPROVEMENTS** Table 3.3 shows further statistics about the category (LD or CF) and the number of lines of code in each benchmark. As can be seen, the benchmarks are quite large and contain up to 50K lines of code. Further, after each join, we measured the total number of variables  $n$  and report the maximum and the average. For the decomposed analyses (ELINA and ours) we measured the size of the largest block and report again maximum and average under  $n_{\max}^{\text{elina}}$ ,  $n_{\max}^{\text{our}}$ . To assess the quality of the

Table 3.3: Partition statistics for the Polyhedra domain analysis.

Benchmark	Category	LOC	n		$n_{\max}^{\text{elina}}$		$n_{\max}^{\text{our}}$		$n_{\max}^{\text{finest}}$	
			max	avg	max	avg	max	avg	max	avg
firewire_firedtv	LD	14506	159	25	81	7	40	4	39	3
net_fddi_skfp	LD	30186	589	88	111	25	45	9	13	4
mtd_ubi	LD	39334	528	59	111	14	28	5	23	4
usb_core_main0	LD	52152	365	72	267	30	60	11	40	7
tty_synclinkmp	LD	19288	332	49	48	10	40	6	26	4
scsi_advansys	LD	21538	282	63	117	18	49	12	41	9
staging_vt6656	LD	25340	675	53	204	17	25	4	12	3
net_ppp	LD	15744	218	58	112	40	51	28	43	20
p10_l00	CF	592	303	174	234	54	79	16	14	6
p16_l40	CF	1783	874	266	86	31	39	14	5	3
p12_l57	CF	4828	921	261	461	78	21	7	4	3
p13_l53	CF	5816	1631	342	617	111	26	10	9	3
p19_l59	CF	9794	1272	358	867	187	31	8	12	3

partitions, we also computed (with the needed overhead) the finest partition after each join and show the largest blocks under  $n_{\max}^{\text{finest}}$  (maximum and average). As can be observed, our partitions are strictly finer than the ones produced by our polyhedra decomposition in Chapter 2 on all benchmarks due to the refinements for the assignment and join transformers. Moreover, it can be seen that the average size of our partitions is sometimes close to that of the finest partition but in many cases there is room for further improvement. We consider this as an interesting item for future work.

### 3.5.2 Octagon

The standard implementation of the Octagon domain works only with the constraint representation and approximates the best conditional and best assignment transformers but implements best join and meet transformers. The widening is defined syntactically. All of these transformers are in the classes of (decomposable) transformers from Section 3.4. Since the syntactic widening does not produce semantically equivalent outputs for semantically equivalent but syntactically different inputs, our fixpoint can be different than the one computed by non-decomposed analysis. However, we still get the same semantic invariants at fixpoint on most of our benchmarks. The standard implementation requires a strong closure operation for the efficiency and precision of transformers such as join, conditional, assignment, and others.

Table 3.4 shows the asymptotic complexity of standard Octagon transformers as well as the strong closure operation with and without decomposition [189]. In the table,  $n$ ,  $n_i$ ,  $n_{\max}$  have the same meaning as in Table 2.2. In can be seen that strong

Table 3.4: Asymptotic time complexity of the Octagon transformers.

Transformer	Non-Decomposed	Decomposed
Conditional	$O(n^2)$	$O(n_{\max}^2)$
Assignment	$O(n^2)$	$O(n_{\max}^2)$
Meet ( $\sqcap$ )	$O(n^2)$	$O(\sum_{i=1}^r n_i^2)$
Join ( $\sqcup$ )	$O(n^2)$	$O(\sum_{i=1}^r n_i^2)$
Widening ( $\nabla$ )	$O(n^2)$	$O(\sum_{i=1}^r n_i^2)$
Strong Closure	$O(n^3)$	$O(\sum_{i=1}^r n_i^3)$

Table 3.5: Speedup for the Octagon domain analysis with our decomposition over the non-decomposed and the decomposed versions of ELINA.

Benchmark	ELINA-ND	ELINA-D	Our Decomposition	Speedup vs.	
	time(s)	time(s)		ELINA-ND	ELINA-D
firewire_firedtv	0.4	0.07	0.07	5.7	1
net_fddi_skfp	28	2.6	1.9	15	1.4
mtd_ubi	3411	979	532	6.4	1.8
usb_core_main0	107	6.1	4.9	22	1.2
tty_synclinkmp	8.2	1	0.8	10	1.2
scsi_advansys	9.3	1.5	0.8	12	1.9
staging_vt6656	4.8	0.3	0.2	24	1.5
net_ppp	11	1.1	1.2	9.2	0.9
p10_l00	20	0.5	0.5	40	1
p16_l40	8.8	0.6	0.5	18	1.2
p12_l57	19	1.2	0.7	27	1.7
p13_l53	43	1.7	1.3	33	1.3
p19_l59	41	2.8	1.2	31	2.2

closure is the most expensive operation with cubic complexity. It is applied incrementally with quadratic cost for the conditional and the assignment transformers.

We compare the performance of our approach for the standard Octagon analysis, using the non-decomposed ELINA (ELINA-ND) and the decomposed (ELINA-D) transformers from ELINA. All of these implementations store the constraint representation using a single matrix with 64-bit doubles. The matrix requires quadratic space in  $n$ . Thus, overall memory consumption is the same for all implementations.

We compare the runtime and report speedups for the end-to-end Octagon analysis in Table 3.5. We achieve up to 40x speedup for the end-to-end analysis over the non-decomposed implementation. More importantly, we are either faster or have the same runtime as the decomposed version of ELINA on all benchmarks but one. The maximum speedup over the decomposed version of ELINA is 2.2x. The speedups on the remaining (not shown) benchmarks vary between 1x and 1.6x

Table 3.6: Partition statistics for the Octagon domain analysis.

Benchmark	Category	LOC	n		$n_{\max}^{\text{elina}}$		$n_{\max}^{\text{our}}$		$n_{\max}^{\text{finest}}$	
			max	avg	max	avg	max	avg	max	avg
firewire_firedtv	LD	14506	159	25	31	6	40	4	27	3
net_fddi_skfp	LD	30186	573	86	49	18	30	10	14	7
mtd_ubi	LD	39334	553	46	111	65	22	9	16	9
usb_core_main0	LD	52152	364	72	59	22	39	9	35	7
tty_synclinkmp	LD	19288	324	49	84	15	26	6	25	4
scsi_advansys	LD	21538	293	64	94	19	41	6	20	5
staging_vt6656	LD	25340	651	52	63	7	25	4	14	3
net_ppp	LD	15744	218	54	40	23	55	29	39	19
p10_l00	CF	592	305	173	19	10	77	16	17	9
p16_l40	CF	1783	874	266	32	12	13	7	10	5
p12_l57	CF	4828	954	265	55	15	13	4	11	4
p13_l53	CF	5816	1635	337	41	12	22	7	10	5
p19_l59	CF	9794	1291	363	79	14	22	4	18	3

with an average of about 1.2x. Notice that on the `mtd_ubi` benchmark, the Octagon analysis takes longer than the Polyhedra analysis. This is because the Octagon widening takes longer to converge compared to the Polyhedra widening.

Table 3.6 shows the partition statistics for the Octagon analysis (as we did for the Polyhedra analysis). It can be seen that while our refinements often produce finer partitions than the decomposed version of ELINA, they are coarser on 3 of the 13 benchmarks. This is because the decomposed transformers in ELINA are specialized for the standard approximations of the conditional and assignment transformers. We still achieve comparable performance on these benchmarks. Note that the average size of our partitions is close to that of the finest in most cases.

### 3.5.3 Zone

The standard Zone domain uses only the constraint representation. The conditional and assignment transformers are approximate whereas the meet and join are best transformers [140]. The widening is defined syntactically. All of these transformers are in the class of (decomposable) transformers from Section 3.4. As for Octagon, fixpoint equivalence is not guaranteed due to syntactic widening. However, we still get the same semantic invariants at fixpoint on most of our benchmarks. As for the Octagon domain, a cubic closure operation is required. The domain transformers have the same asymptotic complexity as in the Octagon domain.

We implemented both, a non-decomposed version as well as a version with our decomposition method of the standard transformers. Both implementations store the constraints using a single matrix with 64-bit doubles that requires quadratic space in  $n$ . We compare the runtime and report speedups for the Zone analy-

Table 3.7: Speedup for the Zone domain analysis with our decomposition over the non-decomposed implementation.

Benchmark	Non-Decomposed	Our Decomposition	Speedup vs. Non-Decomposed
	time(s)	time(s)	
firewire_firedtv	0.05	0.05	1
net_fddi_skfp	3	1.5	2
mtd_ubi	1.4	0.7	2
usb_core_main0	10.3	4.6	2.2
tty_synclinkmp	1.1	0.7	1.6
scsi_advansys	0.9	0.7	1.3
staging_vt6656	0.5	0.2	2.5
net_ppp	1.1	0.7	1.5
p10_l00	1.9	0.4	4.6
p16_l40	1.7	0.7	2.5
p12_l57	3.5	0.9	3.9
p13_l53	8.7	2.1	4.2
p19_l59	9.8	1.6	6.1

Table 3.8: Partition statistics for the Zone domain analysis.

Benchmark	Category	LOC	n		n <sub>max</sub> <sup>our</sup>		n <sub>max</sub> <sup>finest</sup>	
			max	avg	max	avg	max	avg
			firewire_firedtv	LD	14506	159	25	40
net_fddi_skfp	LD	30186	578	88	30	9	13	5
mtd_ubi	LD	39334	553	59	23	5	14	3
usb_core_main0	LD	52152	362	71	37	8	33	7
tty_synclinkmp	LD	19288	328	49	26	6	25	5
scsi_advansys	LD	21538	293	65	41	8	21	7
staging_vt6656	LD	25340	675	53	25	3	13	2
net_ppp	LD	15744	219	58	54	29	47	24
p10_l00	CF	592	303	174	77	16	17	8
p16_l40	CF	1783	856	261	13	7	10	6
p12_l57	CF	4828	882	249	12	4	10	3
p13_l53	CF	5816	1557	317	22	7	20	5
p19_l59	CF	9794	1243	331	14	4	13	3

sis in Table 3.7. Our decomposition achieves speedups of up to 6x over the non-decomposed implementation. The speedups over the remaining benchmarks not shown in the table vary between 1.1x and 5x with an average of  $\approx 1.6x$ .

Table 3.8 shows the partition statistics for the Zone analysis. It can be seen that partitioning is the core reason for the speed-ups obtained and that the average size of our partitions is close to that of the finest in most cases.

### 3.5.4 Summary

Overall, our results show that the generic decomposition method proposed in this chapter works well. It speeds up analysis compared to non-decomposed domains significantly, and, importantly, the more expensive the domain, the higher the speed-ups. Our generic method also compares favorably with the prior manually decomposed domains provided by ELINA due to refined partitioning for the outputs of the assignment and join transformers presented in Section 3.4. The refinement is possible because we refine our model for observing the abstract elements. We also show that the partitions computed during analysis are close to optimal for Octagon and Zone but have further room for improvement for Polyhedra. The challenge is how to obtain those with reasonable cost. Further speed-ups can also be obtained by different implementations of the transformers that are, for example, selectively approximate to achieve finer partitions.

## 3.6 RELATED WORK

We now discuss the work most closely related to Chapters 2 and 3 for improving the performance of numerical program analysis.

**POLYHEDRA** The concept of polyhedra partitioning has been explored before in [93, 94]. Here, the partitions are based upon the decomposition of the matrix encoding the constraint representation of polyhedra. Their output partitions for the join are coarser than ours. This is because the authors rely on a syntactic criteria for computing the output partition which does not detect equal factors. This degrades performance; for example, using the output join partitions computed by this approach in ELINA, it takes  $> 1$  hour to analyze the `usb_core_main0` benchmark in Table 2.3. Further, their partition for the join requires the constraint representation to be available which is not compatible with the eager approach for conversion (Section 2.1.4).

The authors of [182] observe that the polyhedra arising during analysis of their benchmarks show sparsity, i.e., a given variable occurs in only a few constraints. The authors work only with the constraint representation and exploit sparsity in the constraints to speed up the expensive join transformer. In case the output becomes too large, the join is approximated. We implemented this approach in ELINA but without the approximation step so that we do not lose precision. For our benchmarks, we found that the performance of this approach degrades quickly due to frequent calls to the linear solver for redundancy removal.

Another work [143] decomposes polyhedra  $P$  and  $Q$  before applying the join into two factors  $P = \{P_1, P_2\}$  and  $Q = \{Q_1, Q_2\}$ , such that  $P_1 = Q_1$  and  $P_2 \neq Q_2$ . Thus, the conversion is only required for  $\mathcal{G}_{P_2 \sqcup Q_2}$ . This is similar to Theorem 2.2.5. However, the authors rely on syntactic equality between the constraints for iden-

tifying the factors; in contrast, Theorem 2.2.5 relies on semantic equality. Further, their partition is coarser as it has only two blocks which increases the number of generators.

The work of [137, 138, 218] focuses on improving the performance of standard Polyhedra transformers based on constraint representation using parametric linear programming. We believe that our approach is complementary and their transformers could benefit from our decomposition.

[10, 75] provide conversion algorithms that can be more efficient than the Chernikova algorithm used in ELINA currently for certain polyhedra. In the future, a straightforward way to speedup ELINA would be to run the Chernikova algorithm and the ones from [10, 75] in parallel and pick the fastest one.

[184] proposed an incremental conversion algorithm when the constraints are removed. This is useful for speeding up conversions for the assignment and widening transformers. Integrating their algorithms in ELINA would yield further speedups.

[20] introduces a new double representation and efficient conversion algorithm for the NNC (not necessarily closed) Polyhedra domain which is more expressive than the closed Polyhedra domain considered in our work. The follow-up work [21] provides a domain implementation based on the new representation and conversion algorithm. [219] identifies a number of optimization opportunities to make NNC Polyhedra domain even faster. We believe that online decomposition can further speedup the NNC Polyhedra domain without precision loss.

[33] performs bottom-up interprocedural analysis and computes procedure summaries as a disjunction of convex polyhedra. We note that our framework of online decomposition presented here can also be extended to disjunctions of polyhedra.

[29] targets the analysis of particular kinds of programs which produce polyhedra that are not decomposable using online decomposition. The authors develop new data structures for the efficient analysis of the intended programs with the Polyhedra domain. Similarly, [22] identifies a list of optimization opportunities when analyzing hybrid systems. We believe that while our approach is agnostic to the program being analyzed, tailoring the Polyhedra domain implementation for the class of programs being analyzed can further improve our performance and it is an interesting direction of future work.

**OCTAGON** Variable packing [27, 99] has been used for decomposing the Octagon transformers. It partitions  $\mathcal{X}$  statically before running the analysis based on certain criteria. For example, two variables are in the same block of the partition if they occur together in the same program statement. Although variable packing could also be generalized to decompose transformers of other domains, it is fundamentally different from our dynamic decomposition; it is not guaranteed to preserve precision. This is because the permissible partition depends on the Octagon produced during the analysis. Therefore the enforced static partition would not be permissible throughout the analysis and thus the analysis loses precision. Further, the dynamic decomposition often yields even finer partitions than can be detected



statically. So dynamic decomposition (of transformers within the classes defined) provides both higher precision and faster execution.

Our prior work [189] maintains partitions dynamically for the Octagon domain and does not lose precision. The partitions are hand-crafted for the Octagon domain. The partitions computed by our generic approach in Section 3.4 are finer than those produced by [189]. As a result, our approach yields better results than [189] in Section 3.5.

The work of [45, 46] presents a new algorithm for reducing the operation count of incremental closure for the Octagon domain. In cases where the structure of the program allows the analyzer to use incremental closure more frequently than the full closure, incremental closure becomes a bottleneck for the overall analysis. The work of [18] parallelizes the standard octagon operators on GPUs. [44] introduces a new data structure for representing octagons by noticing that the entries in the corresponding matrices are usually identical for their analyzed programs. This reduces the memory required for storing octagons. This approach is similar in spirit to the works mentioned above that adapt the Polyhedra domain to the particular programs being analyzed. Follow-up work [44] proposes optimizations based on the proposed data structure in [44] that reduce the performance gap between octagons implemented with arbitrary-precision arithmetic and those implemented with machine doubles. We believe that our framework is complementary to the above approaches and a combination can yield more performance gains.

The work of [109] designs sparse algorithms for the Octagon domain. While the proposed algorithms cannot be extended to more expressive domains, they could be combined with our decomposition to potentially achieve better performance.

**ZONE** The work of [204] dynamically maintains partitions based on a syntactic criteria for the Zone domain. They also explicitly relate all variables that are modified within a loop. This can lead to a coarser partition for the corresponding join. Further, the resulting analysis is not guaranteed to be as precise as the original non-decomposed analysis.

The authors of [76] observe that the analysis with the Zone domain on their benchmarks is usually sparse. They exploit sparsity by using graph-based algorithms for the Zone domain transformers. Due to the Zone widening being defined syntactically, their sparse analysis is also (like our approach) not guaranteed to obtain the same fixpoint as the original analysis. While these algorithms cannot be extended to more expressive domains such as TVPI or Polyhedra, they could be combined with our decomposition to potentially achieve better performance.

**BEYOND NUMERICAL DOMAINS** In a recent work [56], the authors generalize our online decomposition framework beyond numerical domains and show that it can be applied for decomposing any abstract domain. However, the authors do not provide conditions on the structure of domains and the corresponding transform-

ers when the resulting decomposed analysis does not lose precision. We believe that providing such conditions is interesting future work.

### 3.7 DISCUSSION

Online decomposition is a promising avenue to make numerical domain analysis faster, possibly by orders of magnitude, and thus practical for analyzing many real-world programs. This is made possible thanks to the inherent “locality” in the way program statements, and sequences of such, access variables. In this chapter, we advanced partitioning presented in Chapter 2 by showing that it is generally applicable to all sub-polyhedra domains and constructed decomposed transformers from existing non-decomposed transformers. This way, existing implementations can be re-factored to incorporate decomposition. We also showed that our decomposition does not lose precision on most practical transformers already in use. Recent research has shown that online decomposition can be extended to any abstract domain, not only numerical.

We introduced techniques for refining the output partitions of domain transformers at small extra cost, an improvement over the computed output partitions in Chapter 2. We evaluated our generic approach on three expensive abstract domains: Zone, Octagon, and Polyhedra. We obtain significant speedups over prior work, including on the domain implementations that were previously manually decomposed. Most importantly, based on our results, we observe that the more expensive a domain is, the higher the speedups from online decomposition. Our highest speedups are orders of magnitude on the exponentially expensive Polyhedra. We believe that decomposition can level the playing field among domains, requiring a rethinking of the fundamental question when designing program analysis for a particular application: selecting the domain with the best tradeoff between analysis precision and performance.

In both Chapters 2 and 3, our focus has been on ensuring that our faster, decomposed analysis produces the same result as the original one at every step. It is, however, also possible to selectively lose precision for abstract transformers such that even though the intermediate results are imprecise, the final computed invariants are still the same as the ones from the original analysis. This concept is based on our observation that many of the intermediate analysis results are not needed for computing the final invariants at the fixpoint. Devising such a strategy requires deciding where and how much precision to lose. We take a data-driven approach to learning such strategies in Chapter 4. Our results show that our learned strategies make context-sensitive decisions based on the abstract state and outperform fixed manually designed heuristics.

## REINFORCEMENT LEARNING FOR NUMERICAL DOMAINS

In Chapters 2 and 3, we presented a theoretical framework for dynamically decomposing the abstract elements and transformers of numerical domains without losing precision. The framework is effective in reducing redundancy at each step of the analysis. However, redundancy remains over sequences of abstract transformers. A precise but expensive transformer applied at each step may compute intermediate results discarded downstream in the analysis. This means that it may be possible to achieve the same fixpoint *faster* by selectively losing precision. A key challenge then is coming up with effective, general dynamic approaches that can decide where and how to lose precision for the best tradeoff between performance and precision. Statically fixed policies [42, 99, 131, 156, 157] for losing precision often produce imprecise results.

**Our Work.** We address the above challenge by offering a new approach to *dynamically* losing precision based on reinforcement learning (RL) [195]. The key idea is to use RL and learn a policy that determines when and how the analyzer should lose the least precision at an abstract state, to achieve the best performance gains. The key insight of the work is to establish a correspondence between classic concepts in static analysis with those in reinforcement learning, demonstrating that RL is a viable approach for handling choices in the inner workings of an analyzer.

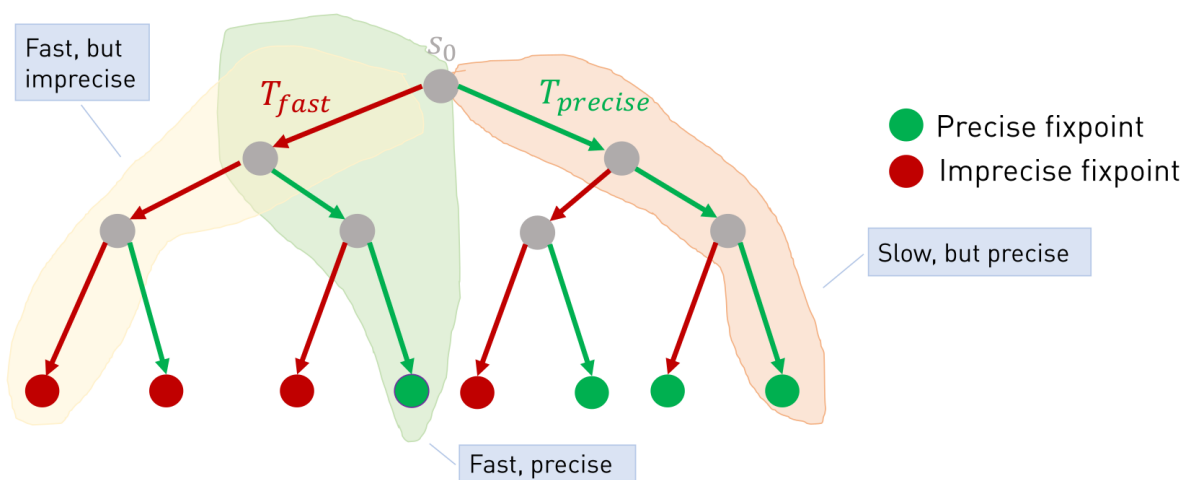


Figure 4.1: Policies for balancing precision and speed in static analysis.

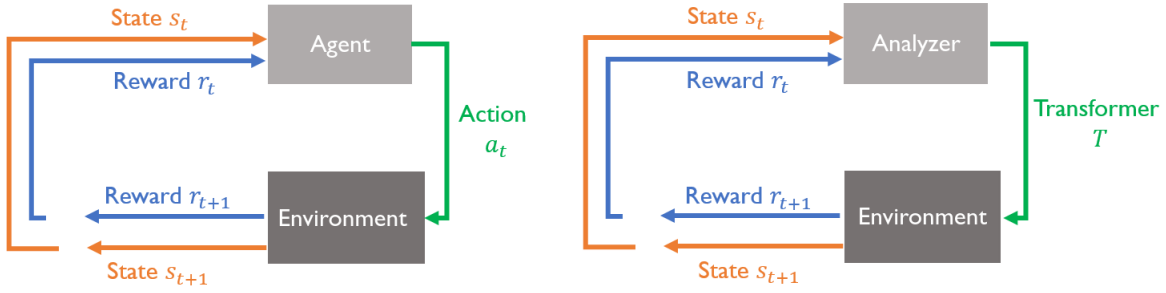


Figure 4.2: Reinforcement learning for static analysis.

We illustrate this connection on the example shown in Fig. 4.1. Here, a static analyzer starts from an initial abstract state  $s_0$  depicted by the root node of the tree. It transitions to a new abstract state, i.e., one of its children, by applying a transformer. At each step, the analyzer can select either a precise but expensive transformer  $T_{\text{precise}}$  or a fast but imprecise one  $T_{\text{fast}}$ . If the analyzer follows a fixed policy that guarantees maximum precision (orange path in Fig. 4.1), it will always apply  $T_{\text{precise}}$  and obtain a precise fixpoint at the rightmost leaf. However, the computation is slow. Analogously, by following a fixed policy maximizing performance (yellow path in Fig. 4.1), the analyzer always chooses  $T_{\text{fast}}$  and obtains an imprecise fixpoint at the leftmost leaf. A policy maximizing both speed and precision (green path in Fig. 4.1) yields a precise fixpoint but is computed faster as the policy applies both  $T_{\text{fast}}$  and  $T_{\text{precise}}$  selectively. A globally optimal sequence of transformer choices that optimizes both objectives is generally very difficult to achieve. However, as we show in this chapter, effective policies that work well in practice can be obtained using principled RL based methods.

Fig. 4.2 explains the connection with RL intuitively. The left-hand side of the figure shows an RL agent in a state  $s_t$  at timestep  $t$ . The state  $s_t$  represents the agents' knowledge about its environment. It takes an action  $a_t$  and moves to a new state  $s_{t+1}$ . The agent obtains a numerical reward  $r_{t+1}$  for the action  $a_t$  in the state  $s_t$ . The agents' goal is maximizing long-term rewards. Notice that the obtained reward depends on both the action and the state. A policy maximizing short-term rewards at each step does not necessarily yield better long term gains as the agent may reach an intermediate state from which all further rewards are negative. RL algorithms typically learn the so-called Q-function, which quantifies the expected long term gains by taking action  $a_t$  in the state  $s_t$ . This setting also mimics the situation that arises in an iterative static analysis shown on the right-hand side of Fig. 4.2. Here the analyzer obtains a representation of the abstract state (its environment) via a set of features  $\phi$ . The analyzer selects among a set of transformers  $T$  with different precision and speed. The transformer choice represents the action. The analyzer obtains a reward in terms of speed and precision. In Fig. 4.1, a learned policy would determine at each step whether to choose  $T_{\text{precise}}$  or  $T_{\text{fast}}$ . To do that, for a given state and action, the analyzer would compute the value of the Q-function

using the features  $\phi$ . Querying the Q-function would then return the suggested action from that state.

While the overall connection between the two areas is conceptually clean, the details of making it work in practice pose significant challenges. The first is the design of suitable approximations to be able to gain performance when precision is lost. The second is the design of the features  $\phi$ , which should be cheap to compute yet be expressive enough to capture key properties of abstract states so that the learned policy generalizes to unseen states. And finally, a suitable reward function is needed that combines both, precision and performance.

The work in this chapter was published in [192].

**MAIN CONTRIBUTIONS** Our main contributions are:

- A new, general approach for speeding up static analysis with reinforcement learning based on establishing a correspondence between concepts in both areas (Section 4.1).
- An instantiation of the approach for speeding up Polyhedra domain analysis, which is known to be expensive with worst-case exponential complexity. The instantiation consists of several contributions:
  - We build on recent work on online decomposition to systematically create a space of approximate Polyhedra transformers (i.e., actions) with different trade-offs between precision and cost (Section 4.2).
  - We design a set of feature functions which capture key properties of abstract states and transformers, yet are efficient to extract during analysis (Section 4.3).
  - We develop a complete instantiation of reinforcement learning for Polyhedra analysis based on Q-learning with linear function approximation (i.e., actions, Q-function, reward function, and policy).
- We provide an end-to-end implementation and evaluation of our approach. Given a training dataset of programs, we first learn a policy (based on the Q-function) over analysis runs of these programs. We then use the resulting policy during the analysis of new, unseen programs. The experimental results on a set of realistic programs (e.g., Linux device drivers) show that our reinforcement learning based Polyhedra analysis achieves substantial speed-ups, in many cases, two to four orders of magnitude over a heavily optimized state-of-the-art implementation of Polyhedra.

Overall, we believe the recipe outlined in this chapter opens up the possibility for speeding-up other analyzers with reinforcement learning based concepts.

#### 4.1 REINFORCEMENT LEARNING FOR STATIC ANALYSIS

In this section we first introduce the general framework of reinforcement learning and then discuss its instantiation for static analysis.

##### 4.1.1 Reinforcement Learning

Reinforcement learning (RL) [195] involves an *agent* learning to achieve a long-term goal by interacting with its *environment*. The agent starts from an initial representation of its environment in the form of an initial state  $s_0 \in \mathcal{S}$  where  $\mathcal{S}$  is the set of possible states. Then, at each time step  $t = 0, 1, 2, \dots$ , the agent performs an action  $a_t \in \mathcal{A}$  in state  $s_t$  ( $\mathcal{A}$  is the set of possible actions) and moves to the next state  $s_{t+1}$ . The agent receives a numerical reward  $r(s_t, a_t, s_{t+1}) \in \mathbb{R}$  for moving from the state  $s_t$  to  $s_{t+1}$  by taking the action  $a_t$ . The agent repeats this process until it reaches a final state. Each sequence of states and actions from an initial state to the final state is called an *episode*.

In RL, state transitions typically satisfy the Markov property: the next state  $s_{t+1}$  depends only on the current state  $s_t$  and the action  $a_t$  taken from  $s_t$ . A *policy*  $p: \mathcal{S} \rightarrow \mathcal{A}$  is a mapping from states to actions: it specifies the action  $a_t = p(s_t)$  that the agent will take when in state  $s_t$ . The agent's goal is to learn a policy that maximizes not an immediate but a cumulative reward for its actions in the long term. The agent does this by selecting the action with the highest expected long-term reward in a given state. The quality function (Q-function)  $Q: \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$  specifies the long term cumulative reward associated with choosing an action  $a_t$  in state  $s_t$ . Learning this function, which is not available a priori, is essential for determining the best policy and is explained next.

**Q-LEARNING AND APPROXIMATING THE Q-FUNCTION.** Q-learning [208] can be used to learn the Q-function over state-action pairs. Typically the size of the state space is so large that it is not feasible to explicitly compute the Q-function for each state-action pair and thus the function is approximated. In this chapter, we consider a *linear* function approximation of the Q-function for three reasons: (i) *effectiveness*: the approach is efficient, can handle large state spaces, and works well in practice [79]; (ii) *it leverages our application domain*: in our setting, it is possible to choose meaningful features (e.g., approximation of polyhedra volume and cost of transformer) that relate to precision and performance of the static analysis and thus it is not necessary to uncover them automatically (as done, e.g., by training a neural net); and (iii) *interpretability of policy*: once the Q-function and associated policy are learned they can be inspected and interpreted.

The Q-function is described as a linear combination of  $\ell$  basis functions  $\phi_i: \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ ,  $i = 1, \dots, \ell$ . Each  $\phi_i$  is a feature that assigns a value to a (state, action) pair and  $\ell$  is the total number of chosen features. The choice of features is important

**Algorithm 4.1** Q-learning algorithm

---

```

1: function Q-learn( $\mathcal{S}, \mathcal{A}, r, \gamma, \alpha, \phi$ )
2:   Input:
3:      $\mathcal{S} \leftarrow$  set of states,  $\mathcal{A} \leftarrow$  set of actions,  $r \leftarrow$  reward function
4:      $\gamma \leftarrow$  discount factor,  $\alpha \leftarrow$  learning rate
5:      $\phi \leftarrow$  set of feature functions over  $\mathcal{S}$  and  $\mathcal{A}$ 
6:   Output: parameters  $\theta$ 
7:    $\theta =$  Initialize arbitrarily (which also initializes Q)
8:   for each episode do
9:     Start with an initial state  $s_0 \in \mathcal{S}$ 
10:    for  $t = 0, 1, 2, \dots, \text{length}(\text{episode})$  do
11:      Take action  $a_t$ , observe next state  $s_{t+1}$  and  $r(s_t, a_t, s_{t+1})$ 
12:       $\theta := \theta + \alpha \cdot (r(s_t, a_t, s_{t+1}) + \gamma \cdot \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)) \cdot \phi(s_t, a_t)$ 
13:    end for
14:  end for
15:  return  $\theta$ 
16: end function

```

---

and depends on the application domain. We collect the feature functions into a vector  $\phi(s, a) = (\phi_1(s, a), \phi_2(s, a), \dots, \phi_\ell(s, a))$ ; doing so, the Q-function has the form:

$$Q(s, a) = \sum_{j=1}^{\ell} \theta_j \cdot \phi_j(s, a) = \phi(s, a) \cdot \theta^T, \quad (4.1)$$

where  $\theta = (\theta_1, \theta_2, \dots, \theta_\ell)$  is the parameter vector. The goal of Q-learning with linear function approximation is thus to estimate (learn)  $\theta$ .

Algorithm 4.1 shows the Q-learning procedure with linear function approximation. In the algorithm,  $0 \leq \gamma < 1$  is the *discount factor* which represents the difference in importance between immediate and future rewards.  $\gamma = 0$  makes the agent only consider immediate rewards while  $\gamma \approx 1$  gives more importance to future rewards. The parameter  $0 < \alpha \leq 1$  is the *learning rate* that determines the extent to which the newly acquired information overrides the old information. The algorithm first initializes  $\theta$  randomly. Then at each step  $t$  in an episode, the agent takes an action  $a_t$ , moves to the next state  $s_{t+1}$  and receives a reward  $r(s_t, a_t, s_{t+1})$ . Line 12 in the algorithm shows the equation for updating the parameters  $\theta$ . Notice that Q-learning is an off-policy learning algorithm as the update in the equation assumes that the agent follows a greedy policy (from state  $s_{t+1}$ ) while the action ( $a_t$ ) taken by the agent (in  $s_t$ ) need not be greedy.

Once the Q-function is learned, a policy  $p^*$  for maximizing the agent's cumulative reward is obtained as:

$$p^*(s) = \operatorname{argmax}_{a \in \mathcal{A}} Q(s, a). \quad (4.2)$$

In the application,  $p^*$  is computed on the fly at each stage  $s$  by computing Q for each action  $a$  and choosing the one with maximal  $Q(s, a)$ . Since the number of actions is typically small, this incurs little overhead.

Table 4.1: Mapping of RL concepts to Static analysis concepts.

RL concept	Static analysis concept
Agent	Static analyzer
State $s \in \mathcal{S}$	Features of abstract state
Action $a \in \mathcal{A}$	Abstract transformer
Reward function $r$	Transformer precision and runtime
Feature	Value associated with abstract state features and transformer

#### 4.1.2 Instantiation of RL to Static Analysis

We now discuss a general recipe for instantiating the RL framework described above to the domain of static analysis. The precise formal instantiation to the specific numerical (Polyhedra) analysis is provided later.

In Table 4.1, we show a mapping between RL and program analysis concepts. Here, the analyzer is the agent that observes its environment, which is the abstract program state (e.g., polyhedron) arising at every iteration of the analysis. In general, the number of possible abstract states can be very large (or infinite) and thus, to enable RL in this setting, we abstract the state through a set of features (Table 4.2). An example of a feature could be the number of bounded program variables in a polyhedron or its volume. The challenge is to define the features to be fast to evaluate, yet sufficiently representative so the policy derived through learning generalizes well to unseen abstract program states.

Further, at every abstract state, the analyzer should have the choice between different actions corresponding to different abstract transformers. The transformers should range from expensive and precise to cheap and approximate. The reward function  $r$  is composed of a measure of precision and speed and should encourage approximations that are both precise and fast.

The goal of our agent then is to learn an approximation policy that at each analysis step selects an action that tries to minimize the loss of analysis precision at fixpoint while improving overall performance. Learning this policy is typically done offline using a given dataset  $\mathcal{D}$  of programs (discussed in Section 4.4). We note that this is computationally challenging because the dataset  $\mathcal{D}$  can contain many programs and each will need to be analyzed many times during training: even a single run of the analysis can contain many (e.g., thousands) calls to transformers. Generating all combinations of different approximate transformers applied at each program point is infeasible. Hence, to improve the efficiency of learning in practice, one would typically exercise the choice for multiple transformers/actions only at selected program points. In our work, we approximate at the join points where the most expensive transformer in the numerical domains is usually applied.



Another key challenge lies in defining a suitable space of transformers. As we will see later, we accomplish this by leveraging recent advances in online decomposition for numerical domains [189, 190, 191] presented in Chapters 2 and 3. We show how to do that for the notoriously expensive Polyhedra analysis; however, the approach is easily extendable to other popular numerical domains, which all benefit from decomposition.

## 4.2 POLYHEDRA ANALYSIS AND APPROXIMATE TRANSFORMERS

In this section we leverage online decomposition to define a flexible approximation framework for the Polyhedra domain analysis (see Chapters 2 and 3) that *loses precision* in a way that directly translates into performance gains. The basic idea is simple: we approximate by dropping constraints to reduce connectivity among constraints and thus to yield finer decompositions of abstract elements. These directly translate into speedup. We consider various options for these approximations; reinforcement learning (in Section 3.3) will then learn a proper, context-sensitive strategy that stipulates when and which approximation option to apply.

Next we describe our strategies for approximating a given transformer to yield finer online decompositions. The strategies are generic and can be instantiated to approximate any transformer in a given domain.

### 4.2.1 Block Splitting

The cost of a decomposed abstract transformer applied on the abstract element(s)  $P$  depends on the sizes of the blocks in the permissible partition  $\bar{\pi}_P$  relevant to the transformer and, more specifically, on the size of the largest such block. Thus, it is desirable to bound this size by a *threshold*  $\in \mathbb{N}$ . The common goal of all our splitting strategies is to satisfy this bound (after splitting). This is done by first identifying all blocks  $\mathcal{X}_t \in \bar{\pi}_P$  with  $|\mathcal{X}_t| > \textit{threshold}$  that the transformer requires and then removing constraints from  $P(\mathcal{X}_t)$  until it decomposes into blocks of sizes  $< \textit{threshold}$ . Since we only remove constraints from the abstract element, the resulting transformer remains sound. Obviously, there are many choices for removing constraints as shown in the next example.

**Example 4.2.1.** Consider the following polyhedron and *threshold* = 4

$$\begin{aligned} \mathcal{X}_t &= \{x_1, x_2, x_3, x_4, x_5, x_6\}, \\ P(\mathcal{X}_t) &= \{x_1 - x_2 + x_3 \leq 0, x_2 + x_3 + x_4 \leq 0, x_2 + x_3 \leq 0, \\ &\quad x_3 + x_4 \leq 0, x_4 - x_5 \leq 0, x_4 - x_6 \leq 0\}. \end{aligned}$$

We can remove  $\mathcal{M} = \{x_4 - x_5 \leq 0, x_4 - x_6 \leq 0\}$  from  $P(\mathcal{X}_t)$  to obtain the constraint set  $\{x_1 - x_2 + x_3 \leq 0, x_2 + x_3 + x_4 \leq 0, x_2 + x_3 \leq 0, x_3 + x_4 \leq 0\}$  with partition  $\{\{x_1, x_2, x_3, x_4\}, \{x_5\}, \{x_6\}\}$ , which obeys the threshold.

**Algorithm 4.2** Block Splitting algorithm

---

```

1: function block_split( $\mathcal{X}_t, P(\mathcal{X}_t), threshold$ )
2:   Input:
3:      $\mathcal{X}_t \leftarrow$  input block,  $P(\mathcal{X}_t) \leftarrow$  input factor,  $threshold \leftarrow$  threshold for the size of  $\mathcal{X}_t$ 
4:   if  $|\mathcal{X}_t| < threshold$  then ▷ Nothing to decompose
5:      $\bar{\pi}_O := \bar{\pi}_O \cup \mathcal{X}_t$ 
6:      $O := O \cup P(\mathcal{X}_t)$ 
7:     return ( $O, \bar{\pi}_O$ )
8:   end if
9:    $r\_algo :=$  choose_removal_algorithm( $\mathcal{X}_t, P(\mathcal{X}_t)$ ) ▷ Choose constraint removal algorithm
10:   $\mathcal{M} :=$  remove_cons( $\mathcal{X}_t, P(\mathcal{X}_t), r\_algo$ ) ▷ Compute constraints  $\mathcal{M} \subseteq P(\mathcal{X}_t)$  to be removed
11:   $O := P(\mathcal{X}_t) \setminus \mathcal{M}$ 
12:   $\bar{\pi}_O :=$  finest_partition( $O, \mathcal{X}_t$ ) ▷ Partition  $\mathcal{X}_t$  w.r.t.  $O$ 
13:  return ( $O, \bar{\pi}_O$ )
14: end function

```

---

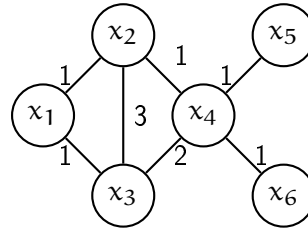
We could also remove  $\mathcal{M}' = \{x_2 + x_3 + x_4 \leq 0, x_3 + x_4 \leq 0\}$  from  $P(\mathcal{X}_t)$  to get the constraint set  $\{x_1 - x_2 + x_3 \leq 0, x_2 + x_3 \leq 0, x_4 - x_5 \leq 0, x_4 - x_6 \leq 0\}$  with partition  $\{\{x_1, x_2, x_3\}, \{x_4, x_5, x_6\}\}$ , which also obeys the threshold.

Algorithm 4.2 shows our generic function `block_split` for splitting a given block  $\mathcal{X}_t$  and the associated factor  $P(\mathcal{X}_t)$ . If the size of the block  $\mathcal{X}_t$  is below the threshold, no decomposition is performed. Otherwise, one out of several possible constraint removal algorithms is chosen (these are explained below) using the function `choose_removal_algorithm` learned by RL. Using the removal algorithm, a set of constraints is removed obtaining  $O$  such that  $\mathcal{X}_t$  decomposes into blocks of size  $\leq threshold$ . We note that  $O$  approximates  $P(\mathcal{X}_t)$  by construction. The associated partition of  $\mathcal{X}_t$  is computed from scratch by connecting the variables that occur in the same constraint using the function `finest_partition`.

We discuss next choices for the removal algorithm that we consider. Depending on the inputs, each may yield different decompositions and different precisions.

**STOER-WAGNER MIN-CUT.** The first basic idea is to remove a minimal number of constraints in  $P(\mathcal{X}_t)$  that decomposes the block  $\mathcal{X}_t$  into two blocks. To do so, we associate with  $P(\mathcal{X}_t)$  a weighted undirected graph  $G = (\mathcal{V}, \mathcal{E})$ , where  $\mathcal{V} = \mathcal{X}_t$ . Further, there is an edge between  $x_i$  and  $x_j$ , if there is a constraint containing both; its weight  $m_{ij}$  is the number of such constraints. We then apply the standard Stoer-Wagner min-cut algorithm [194] to obtain a partition of  $\mathcal{X}_t$  into  $\mathcal{X}'_t$  and  $\mathcal{X}''_t$ .  $\mathcal{M}$  collects all constraints that need to be removed, i.e., those that contain at least one variable from both  $\mathcal{X}'_t$  and  $\mathcal{X}''_t$ . If  $\mathcal{X}'_t, \mathcal{X}''_t$  have sizes  $> threshold$ , then the above procedure is repeated until we get a partition of  $\mathcal{X}_t$  where each block has size  $\leq threshold$ .

**Example 4.2.2.** Fig. 4.3 shows the graph  $G$  for  $P(\mathcal{X}_t)$  in Example 4.2.1. Applying the Stoer-Wagner min-cut on  $G$  once will cut off  $x_5$  or  $x_6$  by removing the constraint

Figure 4.3: Graph  $G$  for  $P(\mathcal{X}_t)$  in Example 4.2.1

$x_4 - x_5$  or  $x_4 - x_6$ , respectively. In either case a block of size 5 remains, exceeding the threshold of 4. After two applications, both constraints have been removed and the resulting block structure is given by  $\{\{x_1, x_2, x_3, x_4\}, \{x_5\}, \{x_6\}\}$ . The associated factors are  $\{x_1 - x_2 + x_3 \leq 0, x_2 + x_3 + x_4 \leq 0, x_2 + x_3 \leq 0, x_3 + x_4 \leq 0\}$  and  $x_5, x_6$  become unconstrained.

**WEIGHTED CONSTRAINT REMOVAL.** Our second approach for constraints removal does not associate weights with edges but with constraints. It then removes greedily edges with high weights. Specifically, we consider the following two choices of constraint weights, yielding two different constraint removal policies:

- For each variable  $x_i \in \mathcal{X}_t$ , we first compute the number  $n_i$  of constraints containing  $x_i$ . The weight of a constraint is then the sum of the  $n_i$  over all variables occurring in the constraint.
- For each pair of variables  $x_i, x_j \in \mathcal{X}_t$ , we first compute the number  $n_{ij}$  of constraints containing both  $x_i$  and  $x_j$ . The weight of a constraint is then the sum of the  $n_{ij}$  over all pairs  $x_i, x_j$  occurring in the constraint.

Once the weights are computed, we remove the constraint with the maximum weight. The intuition is that variables in this constraint most likely occur in other constraints in  $P(\mathcal{X}_t)$  and thus they do not become unconstrained upon constraint removal. This reduces the loss of information. The procedure is repeated until we get the desired partition of  $\mathcal{X}_t$ .

**Example 4.2.3.** Applying the first definition of weights in Example 4.2.1, we get  $n_1 = 1, n_2 = 3, n_3 = 4, n_4 = 4, n_5 = 1, n_6 = 1$ . The constraint  $x_2 + x_3 + x_4 \leq 0$  has the maximum weight of  $n_2 + n_3 + n_4 = 11$  and thus is chosen for removal. Removing this constraint from  $P(\mathcal{X}_t)$  does not yet yield a decomposition; thus we have to repeat. Doing so  $\{x_3 + x_4 \leq 0\}$  is chosen. Now,  $P(\mathcal{X}_t) \setminus \mathcal{M} = \{x_1 - x_2 + x_3 \leq 0, x_2 + x_3 \leq 0, x_4 - x_5 \leq 0, x_4 - x_6 \leq 0\}$  which can be decomposed into two factors  $\{x_1 - x_2 + x_3 \leq 0, x_2 + x_3 \leq 0\}$  and  $\{x_4 - x_5 \leq 0, x_4 - x_6 \leq 0\}$  corresponding to blocks  $\{x_1, x_2, x_3\}$  and  $\{x_4, x_5, x_6\}$ , respectively, each of size  $\leq \text{threshold}$ .

### 4.2.2 Merging of Blocks

Our basic objective when approximating is to ensure that the maximal block size remains below a chosen threshold. Besides splitting to ensure this, there can also be a benefit of merging small blocks, again provided the resulting block size remains below the threshold. The merging itself does not change precision, but the resulting transformer may be more precise when working on larger blocks. In particular this can happen with the inputs of the join transformer as we will explain later.

We consider the following three merging strategies. To simplify the explanation, we assume that the blocks in  $\bar{\pi}_p$  are ordered by ascending size:

1. *No merge*: None of the blocks are merged.
2. *Merge smallest first*: We start merging the smallest blocks as long as the size stays below the threshold. These blocks are then removed and the procedure is repeated on the remaining set.
3. *Merge large with small*: We start to merge the largest block with the smallest blocks as long as the size stays below the threshold. These blocks are then removed and the procedure is repeated on the remaining set.

**Example 4.2.4.** Consider  $threshold = 5$  and  $\bar{\pi}_p$  with block sizes  $\{1, 1, 2, 2, 2, 2, 3, 5, 7, 10\}$ . Merging smallest first yields blocks  $1 + 1 + 2, 2 + 2, 2 + 3$  leaving the rest unchanged. The resulting sizes are  $\{4, 4, 5, 5, 7, 10\}$ . Merging large with small leaves  $10, 7, 5$  unchanged and merges  $3 + 1 + 1, 2 + 2,$  and  $2 + 2$ . The resulting sizes are also  $\{4, 4, 5, 5, 7, 10\}$  but the associated factors are different (since different blocks are merged), which will yield different results in following transformations.

### 4.2.3 Approximation for Polyhedra Analysis

Using the notation and terminology from Chapter 2, we now show how our approximation methods discussed above can be instantiated for the Polyhedra analysis. So far the discussion has been rather generic, i.e., approximation could be done at any time during analysis. We choose to perform approximation only with the join transformer. This is because as explained in Section 2.4 and Section 2.5, the join usually coarsens the partitions substantially and is the most expensive transformer of the Polyhedra analysis.

Let  $\bar{\pi}_{common} = \bar{\pi}_{p_1} \sqcup \bar{\pi}_{p_2}$  be a common permissible partition for the inputs  $P_1, P_2$  of the join transformer. Then, from Chapter 2, a permissible partition for the (not approximated) output is obtained by keeping all blocks  $\mathcal{X}_t \in \bar{\pi}_{common}$  for which  $P_1(\mathcal{X}_t) = P_2(\mathcal{X}_t)$  in the output partition  $\bar{\pi}_O$ , and fusing all remaining blocks into one. Formally,  $\bar{\pi}_O = \{\mathcal{N}\} \cup \mathcal{U}$ , where

$$\mathcal{N} = \bigcup \{\mathcal{X}_k \in \bar{\pi}_{common} : P_1(\mathcal{X}_k) \neq P_2(\mathcal{X}_k)\}, \quad \mathcal{U} = \{\mathcal{X}_k \in \bar{\pi}_{common} : P_1(\mathcal{X}_k) = P_2(\mathcal{X}_k)\}.$$

**Algorithm 4.3** Approximation algorithm for Polyhedra join

---

```

1: function approximate_join( $(P_1, \bar{\pi}_{P_1}), (P_2, \bar{\pi}_{P_2}), threshold$ )
2:   Input:
3:      $(P_1, \bar{\pi}_{P_1}), (P_2, \bar{\pi}_{P_2}) \leftarrow$  decomposed inputs to the join
4:      $threshold \leftarrow$  Upper bound on size of  $\mathcal{N}$ 
5:      $O := \bigcup \{P_1(\mathcal{X}_k) : \mathcal{X}_k \in \mathcal{U}\}$ 
6:      $\bar{\pi}_O := \mathcal{U}$   $\triangleright$  initialize output partition
7:      $\mathcal{B} := \{\mathcal{X}_k \in \bar{\pi}_{P_1} \sqcup \bar{\pi}_{P_2} : \mathcal{X}_k \subseteq \mathcal{N}\}$ 
8:      $\mathcal{B}_t := \{\mathcal{X}_t \in \mathcal{B} : |\mathcal{X}_t| > threshold\}$ 
9:
10:     $\triangleright$  join factors for blocks in  $\mathcal{B}_t$  and split the outputs
11:    for  $\mathcal{X}_t \in \mathcal{B}_t$  do
12:       $P' := P_1(\mathcal{X}_t) \sqcup P_2(\mathcal{X}_t)$ 
13:       $(\mathcal{C}, \bar{\pi}) := \text{block\_split}(\mathcal{X}_t, P', threshold)$ 
14:      for  $\mathcal{X}_{t'} \in \bar{\pi}$  do
15:         $\mathcal{G}(\mathcal{X}_{t'}) := \text{conversion}(\mathcal{C}(\mathcal{X}_{t'}))$ 
16:         $O := O \cup (\mathcal{C}(\mathcal{X}_{t'}), \mathcal{G}(\mathcal{X}_{t'}))$ 
17:      end for
18:       $\bar{\pi}_O := \bar{\pi}_O \cup \bar{\pi}$ 
19:    end for
20:
21:     $\triangleright$  merge blocks  $\in \mathcal{B} \setminus \mathcal{B}_t$  via a merge algorithm and apply join
22:     $m\_algo := \text{choose\_merge\_algorithm}(\mathcal{B} \setminus \mathcal{B}_t)$ 
23:     $\mathcal{B}_m := \text{merge}(\mathcal{B} \setminus \mathcal{B}_t, m\_algo)$ 
24:    for  $\mathcal{X}_m \in \mathcal{B}_m$  do
25:       $O := O \cup (P_1(\mathcal{X}_m) \sqcup P_2(\mathcal{X}_m))$ 
26:       $\bar{\pi}_O := \bar{\pi}_O \cup \{\mathcal{X}_m\}$ 
27:    end for
28:    return  $(O, \bar{\pi}_O)$ 
29: end function

```

---

The join transformer computes the generators  $\mathcal{G}_O$  for the output  $O$  as  $\mathcal{G}_O = \mathcal{G}_{P_1(\mathcal{X} \setminus \mathcal{N})} \times (\mathcal{G}_{P_1(\mathcal{N})} \cup \mathcal{G}_{P_2(\mathcal{N})})$  where  $\times$  is the Cartesian product. The constraint representation  $\mathcal{C}_O$  is computed as  $\mathcal{C}_O = \mathcal{C}_{P_1(\mathcal{X} \setminus \mathcal{N})} \cup \text{conversion}(\mathcal{G}_{P_1(\mathcal{N})} \cup \mathcal{G}_{P_2(\mathcal{N})})$ . The conversion algorithm has worst-case exponential complexity and is the most expensive step of the join. Note that the decomposed join applies it only on the generators  $\mathcal{G}_{P_1(\mathcal{N})} \cup \mathcal{G}_{P_2(\mathcal{N})}$  corresponding to the block  $\mathcal{N}$ .

Let  $\mathcal{B} \in \bar{\pi}_{\text{common}}$  be the set of blocks that merge into  $\mathcal{N}$  in the output  $O = P_1 \sqcup P_2$ :

$$\mathcal{B} = \{\mathcal{X}_k \in \bar{\pi}_{\text{common}} : \mathcal{X}_k \cap \mathcal{N} \neq \emptyset\}.$$

A straightforward way of approximating the join is to compute it separately on each block  $\mathcal{X}_k$  in  $\mathcal{B}$  as  $P_1(\mathcal{X}_k) \sqcup P_2(\mathcal{X}_k)$ , which yields the output partition  $\bar{\pi}_{\text{common}}$ . However, it may contain block sizes above *threshold*. Further, precision can be gained for subsequent transformers by merging  $\mathcal{X}_k$  with sizes below *threshold*. Thus we merge small input blocks using a merge *before* the join, and split large output blocks in  $\mathcal{B}_t = \{\mathcal{X}_k \in \mathcal{B} : |\mathcal{X}_k| > threshold\}$  *after* the join.

Algorithm 4.3 shows the overall algorithm for approximating the Polyhedra join transformer and is explained in greater detail next.

**SPLITTING OF LARGE BLOCKS** For each block in  $\mathcal{X}_t$  in  $\mathcal{B}_t$  we apply the join on the associated factors:  $O(\mathcal{X}_t) = P_1(\mathcal{X}_t) \sqcup P_2(\mathcal{X}_t)$ . Then we call the function `block_split` (Algorithm 4.2) to split  $\mathcal{X}_t$  into smaller blocks, each of size  $\leq \textit{threshold}$ . The conversion is now applied on the set of constraint sets returned by `block_split`.

We perform a number of optimizations in our implementation that are not shown in Algorithm 4.3 for simplicity. For example,  $\{\mathcal{X}_t\}$  is permissible for  $\mathcal{C}_{P'}$ , but may not be finest. Thus we first compute the finest partition from scratch and then call `block_split` only if it contains a block of size  $> \textit{threshold}$ . The cost of this preprocessing is cheap compared to the cost of the overall join.

**MERGING BLOCKS** All blocks in  $\mathcal{B} \setminus \mathcal{B}_t$  obey the threshold size and we can apply merging to obtain larger blocks  $\mathcal{X}_m$  of size  $\leq \textit{threshold}$  to increase precision of the subsequent transformers. The merging function `choose_merge_algorithm` in Algorithm 4.3 is learned by RL. The join is then applied on the factors  $P_1(\mathcal{X}_m), P_2(\mathcal{X}_m)$  and the result is added to the output  $O$ .

**NEED FOR RL.** Different choices of the threshold, splitting, and merge strategies in Algorithm 4.3 yield a range of transformers with different performance and precision depending on the inputs. Determining the suitability of a given choice on inputs is highly non-trivial and thus we use RL to learn a policy that makes decisions adapted to the join inputs. We note that all of our approximate transformers are non-monotonic; however, the analysis always converges to a fixpoint when combined with widening [13].

#### 4.3 REINFORCEMENT LEARNING FOR POLYHEDRA ANALYSIS

We now describe how to instantiate reinforcement learning for approximating Polyhedra domain analysis. The instantiation consists of the following steps:

- Extracting the RL state  $s$  from the abstract program state numerically using a set of features.
- Defining actions  $a$  as the choices among the threshold, merge and split methods defined in the previous section.
- Defining a reward function  $r$  favoring both high precision and fast execution.
- Defining the feature functions  $\phi(s, a)$  to enable Q-learning.

**STATES.** We consider nine features for defining a state  $s$  for RL. The features  $\psi_i$ , their extraction complexity, and their typical range on our benchmarks are shown

Table 4.2: Features for describing RL state  $s$  ( $m \in \{1, 2\}, 0 \leq j \leq 8, 0 \leq h \leq 3$ ).

Feature $\psi_i$	Extraction complexity	Typical range	$n_i$	Buckets for feature $\psi_i$
$ \mathcal{B} $	$O(1)$	1–10	10	$\{[j+1, j+1]\} \cup \{[10, \infty)\}$
$\min( \mathcal{X}_k  : \mathcal{X}_k \in \mathcal{B})$	$O( \mathcal{B} )$	1–100	10	$\{[10 \cdot j + 1, 10 \cdot (j+1)]\} \cup \{[91, \infty)\}$
$\max( \mathcal{X}_k  : \mathcal{X}_k \in \mathcal{B})$	$O( \mathcal{B} )$	1–100	10	$\{[10 \cdot j + 1, 10 \cdot (j+1)]\} \cup \{[91, \infty)\}$
$\text{avg}( \mathcal{X}_k  : \mathcal{X}_k \in \mathcal{B})$	$O( \mathcal{B} )$	1–100	10	$\{[10 \cdot j + 1, 10 \cdot (j+1)]\} \cup \{[91, \infty)\}$
$\min( \bigcup \mathcal{G}_{P_m(x_k)}  : \mathcal{X}_k \in \mathcal{B})$	$O( \mathcal{B} )$	1–1000	10	$\{[100 \cdot j + 1, 100 \cdot (j+1)]\} \cup \{[901, \infty)\}$
$\max( \bigcup \mathcal{G}_{P_m(x_k)}  : \mathcal{X}_k \in \mathcal{B})$	$O( \mathcal{B} )$	1–1000	10	$\{[100 \cdot j + 1, 100 \cdot (j+1)]\} \cup \{[901, \infty)\}$
$\text{avg}( \bigcup \mathcal{G}_{P_m(x_k)}  : \mathcal{X}_k \in \mathcal{B})$	$O( \mathcal{B} )$	1–1000	10	$\{[100 \cdot j + 1, 100 \cdot (j+1)]\} \cup \{[901, \infty)\}$
$ \{x_i \in \mathcal{X} : x_i \in [l_m, u_m] \text{ in } P_m\} $	$O(\text{ng})$	1–25	5	$\{[5 \cdot h + 1, 5 \cdot (h+1)]\} \cup \{[21, \infty)\}$
$ \{x_i \in \mathcal{X} : x_i \in [l_m, \infty) \text{ in } P_m\} +$	$O(\text{ng})$	1–25	5	$\{[5 \cdot h + 1, 5 \cdot (h+1)]\} \cup \{[21, \infty)\}$
$ \{x_i \in \mathcal{X} : x_i \in (-\infty, u_m] \text{ in } P_m\} $				

in Table 4.2. The first seven features capture the asymptotic complexity of the join, as in Table 2.2, on the input polyhedra  $P_1, P_2$ . These are the number of blocks, the distribution (using maximum, minimum and average) of their sizes, and the distribution (using maximum, minimum and average) of the number of generators in the different factors. The precision of the inputs is captured by considering the number of variables  $x_i \in \mathcal{X}$  with finite upper and lower bound, and the number of those with only a finite upper or lower bound in both  $P_1$  and  $P_2$ .

As shown in Table 4.2, each state feature  $\psi_i$  returns a natural number; however, its range can be rather large, resulting in a massive state space. To ensure scalability and generalization of learning, we use bucketing to reduce the state space size by clustering states with similar precision and expected join cost. The number  $n_i$  of buckets for each  $\psi_i$  and their definition are shown in the last two columns of Table 4.2. Using bucketing, the RL state  $s$  is then a 9-tuple consisting of the indices of buckets where each index indicates the bucket that  $\psi_i$ 's return value falls into.

**ACTIONS.** An action  $a$  is a 3-tuple  $(th, r\_algo, m\_algo)$  consisting of:

- $th \in \{1, 2, 3, 4\}$  depending on  $threshold \in [5, 9], [10, 14], [15, 19],$  or  $[20, \infty)$ .
- $r\_algo \in \{1, 2, 3\}$ : the choice of a constraint removal, i.e., splitting method.
- $m\_algo \in \{1, 2, 3\}$ : the choice of merge algorithm.

All three of these have been discussed in detail in Section 4.2. The *threshold* values were chosen based on performance characterization on our benchmarks. With the above, we have 36 possible actions per state.

**REWARD.** After applying the (approximated) join transformer according to action  $a_t$  in state  $s_t$ , we compute the precision of the output polyhedron  $P_1 \sqcup P_2$  by

first computing the smallest (often unbounded) box<sup>1</sup> covering  $P_1 \sqcup P_2$  which has complexity  $O(ng)$ . We then compute the following quantities from this box:

- $n_s$ : number of variables  $x_i$  with finite singleton interval, i.e.,  $x_i \in [l, u], l = u$ .
- $n_b$ : number of variables  $x_i$  with finite upper and lower bounds, i.e.,  $x_i \in [l, u], l \neq u$ .
- $n_{hb}$ : number of variables  $x_i$  with either finite upper or finite lower bounds, i.e.,  $x_i \in (-\infty, u]$  or  $x_i \in [l, \infty)$ .

Further, we measure the runtime in CPU cycles  $cyc$  for the approximate join transformer. The reward is then defined by

$$r(s_t, a_t, s_{t+1}) = 3 \cdot n_s + 2n_b + n_{hb} - \log_{10}(cyc). \quad (4.3)$$

As the order of precision for different types of intervals is: singleton  $>$  bounded  $>$  half bounded interval, the reward function in (4.3) weighs their numbers by 3, 2, 1. The reward function in (4.3) favors both high performance and precision. It also ensures that the precision part ( $3 \cdot n_s + 2n_b + n_{hb}$ ) has a similar magnitude range as the performance part  $(\log_{10}(cyc))^2$ .

**Q-FUNCTION.** As mentioned before, we approximate the Q-function by a linear function (4.1). We define binary feature functions  $\phi_{ijk}$  for each (state, action) pair.  $\phi_{ijk}(s, a) = 1$  if the tuple  $s(i)$  lies in  $j$ -th bucket and action  $a = a_k$

$$\phi_{ijk}(s, a) = 1 \iff s(i) = j \text{ and } a = a_k \quad (4.4)$$

The Q-function is a linear combination of state action features  $\phi_{ijk}$

$$Q(s, a) = \sum_{i=1}^9 \sum_{j=1}^{n_i} \sum_{k=1}^{36} \theta_{ijk} \cdot \phi_{ijk}(s, a). \quad (4.5)$$

**Q-LEARNING.** During the training phase, we are given a dataset of programs  $\mathcal{D}$  and we use Q-LEARN from Algorithm 4.1 on each program in  $\mathcal{D}$  to perform Q-learning. Q-learning is performed with input parameters instantiated as explained above and summarized in Table 4.3. Each episode consists of a run of Polyhedra analysis on a benchmark in  $\mathcal{D}$ . We run the analysis multiple times on each program in  $\mathcal{D}$  and update the Q-function after each join by calling Q-LEARN.

The Q-function is typically learned using an  $\epsilon$ -greedy policy [195] where the agent takes greedy actions by exploiting the current Q-estimates with  $1 - \epsilon$  probability while also exploring randomly with  $\epsilon$  probability. The policy requires initial

<sup>1</sup> A natural measure of precision is the volume of  $P_1 \sqcup P_2$ . However, calculating it is very expensive and  $P_1 \sqcup P_2$  is often unbounded.

<sup>2</sup> The log is used since the runtime of join in cycles is exponential.



Table 4.3: Instantiation of Q-learning to Polyhedra domain analysis.

RL concept	Polyhedra Analysis Instantiation
Agent	Polyhedra analysis
State $s \in \mathcal{S}$	As described in Table 4.2
Action $a \in \mathcal{A}$	Tuple $(th, r\_algo, m\_algo)$
Reward function $r$	Shown in (4.3)
Feature $\phi$	Defined in (4.4)
Q-function	Q-function from (4.5)

random exploration to learn good Q-estimates that can be exploited later. The number of episodes required for obtaining such estimates is infeasible for the Polyhedra analysis as an episode typically contains thousands of join calls. Therefore, we generate actions for Q-learning by exploiting the optimal policy for precision (which always selects the precise join) and explore for performance by choosing a random approximate join: both with a probability of 0.5. We note that we also tried the exploitation probabilities of 0.7 and 0.9. However, the resulting policies had a suboptimal performance during testing due to limited exploration.

Formally, the action  $a_t := p(s_t)$  selected in state  $s_t$  during learning is given by  $a_t = (th, r\_algo, m\_algo)$  where

$$th = \begin{cases} \text{rand}() \% 4 + 1 & \text{with probability } 0.5 \\ \min(4, (\sum_{i=1}^{|\mathcal{B}|} |\mathcal{X}_k|) / 5 + 1) & \text{with probability } 0.5 \end{cases} \quad (4.6)$$

$$r\_algo = \text{rand}() \% 3 + 1, m\_algo = \text{rand}() \% 3 + 1.$$

**OBTAINING THE LEARNED POLICY.** After learning over the dataset  $\mathcal{D}$ , the learned approximating join transformer in state  $s_t$  chooses an action according to (4.2) by selecting the maximal value over all actions. The value of  $th = 1, 2, 3, 4$  is decoded as *threshold* = 5, 10, 15, 20 respectively.

#### 4.4 EXPERIMENTAL EVALUATION

We implemented our approach in the form of a C-library for Polyhedra analysis, called Poly-RL. We compare the performance and precision of Poly-RL against the state-of-the-art ELINA [1] (Chapters 2 and 3), which uses online decomposition for Polyhedra analysis without losing precision. In addition, we implemented two Polyhedra analysis approximations (baselines) based on the following heuristics:

- **Poly-Fixed:** uses a *fixed* strategy based on the results of Q-learning. Namely, we selected the threshold, split and merge algorithm most frequently chosen by our (adaptive) learned policy during testing.

- Poly-Init: uses a random approximate join with probability 0.5 based on (4.6).

All Polyhedra implementations use 64-bit integers to encode rational numbers. In the case of overflow, the corresponding polyhedron is set to top as in Section 2.5 and Section 3.5.

**EXPERIMENTAL SETUP** All our experiments including learning the parameters  $\theta$  for the Q-function and the evaluation of the learned policy on unseen benchmarks were carried out on a 2.13 GHz Intel Xeon E7- 4830 Haswell CPU with 24 MB L3 cache and 256 GB memory. All Polyhedra implementations were compiled with gcc 5.4.0 using the flags `-O3 -m64 -march=native`.

**ANALYZER** For both learning and testing, we used a newer version of the *crab-llvm* analyzer that is different from the versions used in Section 2.5 and Section 3.5.

**BENCHMARKS** We chose benchmarks from the Linux Device Drivers (LD) category of the popular software verification competition [24]. Some of the benchmarks we used for both learning and testing were also used in Section 2.5 and Section 3.5 while others are different.

**TRAINING DATASET** We chose 70 large benchmarks for Q-learning. We ran each benchmark a thousand times over a period of three days to generate sample traces of Polyhedra analysis containing thousands of calls to the join transformer. Since the *crab-llvm* analyzer is intra-procedural, we get an analysis trace for each function leading to several traces per analyzed benchmark. We set a timeout of 5 minutes per run and discarded incomplete traces in case of a timeout. In total, we performed Q-learning over 110811 traces.

**EVALUATION METHOD** For evaluating the effectiveness of our learned policy, we then chose benchmarks based on the following criteria:

- No overfitting: the benchmark was not used for learning the policy.
- Challenging: ELINA takes  $\geq 5$  seconds on the benchmark.
- Fair: there is no integer overflow in the expensive functions in the benchmark. Because in the case of an overflow, the polyhedron is set to top resulting in a trivial fixpoint at no cost and thus in a speedup that is due to overflow. This fairness criterion is the same as the one used in Section 2.5 and Section 3.5.

Based on these criteria, we found 11 benchmarks on which we present our results. We used a timeout of 1 hour and a memory limit of 100 GB for our experiments.

Table 4.4: Timings (seconds) and precision of approximations (%) w.r.t. ELINA.

Benchmark	#Program Points	ELINA	Poly-RL		Poly-Fixed		Poly-Init	
		time	time	precision	time	precision	time	precision
wireless_airo	2372	877	6.6	100	6.7	100	5.2	74
net_ppp	680	2220	9.1	87	T0	34	7.7	55
mfd_sm501	369	1596	3.1	97	1421	97	2	64
ideapad_laptop	461	172	2.9	100	157	100	M0	41
pata_legacy	262	41	2.8	41	2.5	41	M0	27
usb_ohci	1520	22	2.9	100	34	100	M0	50
usb_gadget	1843	66	37	60	35	60	T0	40
wireless_b43	3226	19	13	66	T0	28	83	34
lustre_llite	211	5.7	4.9	98	5.4	98	6.1	54
usb_cx231xx	4752	7.3	3.9	≈100	3.7	≈100	3.9	94
netfilter_ipvs	5238	20	17	≈100	9.8	≈100	11	94

**INSPECTING THE LEARNED POLICY** Our learned policy chooses in the majority of cases  $threshold=20$ , the binary weighted constraint removal algorithm for splitting, and the merge smallest first algorithm for merging. Poly-Fixed always uses these values for defining an approximate transformer, i.e., it follows a fixed strategy. Our experimental results show that following this fixed strategy results in suboptimal performance compared to our learned policy that makes adaptive, context-sensitive decisions to improve performance.

**RESULTS** We measure the precision as a fraction of program points at which the Polyhedra invariants generated by approximate analysis are semantically the same or stronger than the ones generated by ELINA. This is a less biased and more challenging measure than the number of discharged assertions [42, 156, 157] where one can write weak assertions that even a weaker domain can prove.

Table 4.4 shows the number of program points,<sup>3</sup> timings (in seconds), and the precision (in %) of Poly-RL, Poly-Fixed, and Poly-Init w.r.t. ELINA on all 11 benchmarks. In the table, the entry T0 (M0) means that the analysis did not finish within 1 hour (exceeded the memory limit). For an analysis that did not finish, we compute the precision by comparing program points for which the incomplete analysis can produce invariants.

**POLY-RL VS ELINA** In Table 4.4, Poly-RL obtains  $> 7x$  speed-up over ELINA on 6 of the 11 benchmarks with a maximum of 515x speedup for the `mfd_sm501` benchmark. It also obtains the same or stronger invariants on  $\geq 87\%$  of program points on 8 benchmarks. Note that Poly-RL obtains both large speedups and the same invariants at all program points on 3 benchmarks.

<sup>3</sup> The benchmarks contain up to 50K LOC but SeaHorn encodes each basic block as one program point; thus, the number of points in Table 4.4 is significantly reduced.

Many of the constraints produced by the precise join transformer from ELINA are removed by the subsequent transformers in the analysis which allows Poly-RL to obtain the same invariants as ELINA despite the loss of precision during join in most cases. Due to non-monotonic join transformers, Poly-RL can produce fixpoints non-comparable to those produced by ELINA. Because of the non-comparability, the quality of the obtained invariants cannot be established using our precision metric. We take a conservative approach and mark all non-comparable invariants as being imprecise. We note that this is the case for the 3 benchmarks in Table 4.4 where Poly-RL obtains low precision.

We also tested Poly-RL on 17 benchmarks from the product lines category. ELINA did not finish within an hour on any of these benchmarks whereas Poly-RL finished within 1 second. Poly-RL had 100% precision on the subset of program points at which ELINA produces invariants. With Poly-RL, SeaHorn successfully discharged the assertions. We did not include these results in Table 4.4 as the precision w.r.t. ELINA cannot be completely compared.

**POLY-RL VS POLY-FIXED** Poly-Fixed is never significantly more precise than Poly-RL in Table 4.4. Poly-Fixed is faster than Poly-RL on 4 benchmarks; however, the speedups are small. These results validate our hypothesis that a fixed policy yields suboptimal performance and precision. We note that Poly-Fixed is slower than ELINA on 3 benchmarks and times out on 2 of these. This is due to the overhead of the binary weight constraints removal algorithm and the approximate analysis converging slower than ELINA.

**POLY-RL VS POLY-INIT** From (4.6), Poly-Init takes random actions and thus the quality of its result varies depending on the run. Table 4.4 shows the results on a sample run. Poly-RL is more precise than Poly-Init on all benchmarks in Table 4.4. Poly-Init also does not finish on 4 benchmarks.

## 4.5 RELATED WORK

As our work touches on several topics, we next survey some of the work that is most closely related to ours.

**LEARNING IN PROGRAM ANALYSIS** Our work in this chapter can be seen as part of the general research direction of parametric program analysis [7, 42, 96, 99, 100, 105, 131, 156, 157], a high level approach for building program analyzers that tune the precision and cost of the analysis by adapting to the analyzed program. In this setting, the analysis has parameters that control its precision and cost.

The authors in [157] propose parameter learning as a blackbox optimization problem, and use Bayesian optimization for finding the tuned parameters. The work of [42, 99, 156] computes a fixed static partition of the set of variables  $\mathcal{X}$  for

the Octagon domain analysis on a given program. The abstract transformers are then applied individually on the relevant factors. The work of [131] provides algorithms to learn *minimal* values of the tuning parameters for points-to analysis. The work of [105] proposes a data-driven approach that automatically learns a good heuristic rule for choosing important context elements for points-to analysis. The difference between all of these works and ours is that the above approaches fix the learning parameters for a given program. We believe that better tuning of cost and precision can be achieved by changing the learning parameters dynamically based on the abstract states encountered by the analyzer during the analysis. Furthermore, these approaches measure precision by the number of queries proved whereas we target the stronger notion of fixed point equivalence.

The work of [100] uses reinforcement learning to select a subset of variables to be tracked in a flow sensitive manner with the weaker Interval domain so that the memory footprint of the resulting analysis fits within a pre-determined memory budget and the loss of precision is minimized. The work of [47] uses reinforcement learning to guide relational proof search for verifying relational properties defined over multiple programs. The authors in [7] use bayesian optimization to learn a verification policy that guides numerical domain analysis during proof search.

In our recent work [96], we presented a new structured learning method based on graph neural networks for speeding up numerical domains. The method is more generic than the one presented in this chapter as the approximate transformers there are not derived via custom splitting and merging algorithms but simply obtained via constraint removal. Further, the features are more precise than ours and capture structural dependencies between constraints. The results show that the new method outperforms Poly-RL and also outperforms [191] when instantiated for the Octagon domain.

In a recent work by [119], the authors observe that many programs share common pieces of code. Thus analysis results can be reused across programs. To achieve this, the authors use cross program training. This is a complementary approach to the one taken in this work and we believe that in the future the two methods can be combined to further improve the performance of static analysis.

The work of [25] automatically learns abstract transformers from examples. This is a rather different approach, instead, we build approximations on top of standard transformers based on online decomposition. We believe that the approach of [25] can be combined with ours in the future to automate the process of generating approximate abstract transformers.

**ONLINE DECOMPOSITION** The work of [189, 190, 191] improve the performance of the numerical domain analysis based on online decomposition without losing precision. We compare against [191] in this chapter. As our experimental results suggest, the performance of Polyhedra analysis can be significantly improved with our approach. Further, some benchmarks are inherently dense, i.e., fully precise

online decomposition cannot decompose the set of variables  $|\mathcal{X}|$  efficiently, and in such cases our approach can be used to generate precise invariants efficiently.

**ABSTRACTION REFINEMENT** In general and as demonstrated by our experiments, our method can sometimes be imprecise when our approximate join transformers lose precision due to a low threshold value. We believe that in the future, techniques similar to classic counter example guided abstraction refinement (CEGAR) [52] and lazy abstraction [98] can be combined with our approach, i.e., concretely, to increase the precision of our approximate abstract transformers by finding the ones that lead to the precision loss and (optionally) trying out larger values of the threshold (i.e, a more precise transformer).

**NUMERICAL SOLVERS** Machine learning methods have been extensively applied for optimizing different solvers. Reinforcement learning based on linear function approximation of the Q-function has been applied to learn branching rules for SAT solvers in the work of [121]. The learned policies achieve performance levels equal to those of the best branching rules. The work of [14, 115] learns branching rules via empirical risk minimization for solving mixed integer linear programming problems. The work of [116] learns to solve combinatorial optimization problems over graphs via reinforcement learning. FastSMT [16] learns a policy to apply appropriate tactics to speed up numerical SMT solving. The work of [126] uses reinforcement learning combined with graph neural networks for improving the efficiency of solvers for quantified boolean formulas. The work of [135] learns branching rules for mixed integer linear programming using graph neural networks to improve the scalability of complete verifiers of neural networks.

**INFERRING INVARIANTS** Recent research has investigated the problem of inferring numerical program invariants with machine learning. The works of [77, 181, 222] use machine learning for inferring inductive loop invariants for program verification. The learning algorithms in these works require specifications in the form of pre/post conditions. Our work speeds up numerical abstract interpretation which can be used to infer program invariants without pre/post conditions.

**TESTING** In recent years, there has been emerging interest in learning to produce test inputs for finding program bugs or vulnerabilities. AFLFast [28] models program branching behavior with Markov Chain, which guides the input generation. Several works train neural networks to generate new test inputs, where the training set can be obtained from an existing test corpus [61, 84], inputs generated earlier in the testing process [178], or inputs generated by symbolic execution [95].

## 4.6 DISCUSSION

In this chapter, we further improved the performance of numerical program analysis based on online decomposition presented in Chapters 2 and 3. Our main contribution here is to offer and demonstrate a new approach based on reinforcement learning (RL) towards achieving scalable and precise analysis. The basic idea is simple: using RL to compute an adaptive, context-sensitive policy on how to lose as little precision as possible during analysis and to speed up the analyzer as much as possible. To do so, we first showed that program analysis naturally maps to the basic scenario and concepts needed to perform RL with Q-learning. To make the approach work, three key ingredients are needed: a suitably designed set of features that capture state/agent (abstract element/transformer) pairs and that are efficient to compute, a set of transformers with different precision/runtime trade-offs, and a reward function that assesses the quality of choosing a transformer at some state during analysis.

To evaluate these general ideas, we instantiated the approach for the notoriously expensive Polyhedra abstract domain, which has worst-case exponential complexity. To define transformers with different precision/performance trade-offs, we built on recent successes with online decomposition. Namely, we defined a novel set of flexible approximate transformers that enforce a finer decomposition, which directly translates to reduced asymptotic complexity and thus faster execution. As a feature set, we considered various readily available statistics on the block sizes in the decomposition and the bounds within the polyhedra. Our reward function composed both the precision of a polyhedron and the runtime of the transformer.

Based on this concrete instantiation, we then learned the appropriate parameters of the Q-function from a training dataset of programs and obtained a learned policy function, which, during analysis, makes context-sensitive choices on which approximate transformer to employ.

We provided a complete implementation of our approach and evaluated it on a range of realistic programs, including Linux device drivers, that are expensive to analyze. The results demonstrate that RL-based analysis can provide massive speed-ups of sometimes orders of magnitude over a highly optimized Polyhedra analysis library, while often maintaining precision at most program points.

Overall, we believe the correspondence between reinforcement learning and static analysis (as well as its concrete instantiation) established in this work, is instructive and can serve as a starting point for exploring and exploiting this connection for other types of analyzers.





---

## PART II

---

# FAST AND PRECISE NEURAL NETWORK CERTIFICATION



---

## DEEPPOLY DOMAIN FOR CERTIFYING NEURAL NETWORKS

---

In Chapters 2-4, we focussed on designing precise and scalable reasoning methods for programs. Next we shift our focus to the problem domain of deep learning models mentioned in Chapter 1. In the next two chapters, we describe our contributions for designing state-of-the-art neural network certification methods.

In this chapter, we target incomplete certification of neural networks using abstract interpretation. Incompleteness implies that there can be cases where we incorrectly fail to certify the network because the approximation is too coarse (as in Fig. 1.1 (a)). We describe our new custom polyhedral domain DeepPoly for neural network certification containing specialized parallelizable transformers for handling non-linearities in neural networks. DeepPoly currently produces the most scalable and precise neural network certification results. We also show how to combine DeepPoly with a form of abstraction refinement based on trace partitioning. This enables us to prove, for the first time, the robustness of the network when the input image is subjected to complex perturbations such as rotations that employ linear interpolation. In Chapter 6, we will focus on combining our domain with precise solvers. We will show that this increases the precision of incomplete certification while also improving the scalability of complete certification. Before proceeding, we note that parts of this and the next chapter include contributions from Timon Gehr and Rupanshu Ganvir.

Over the last few years, deep neural networks have become increasingly popular and have now started penetrating safety-critical domains such as autonomous driving [30] and medical diagnosis [6] where they are often relied upon for making important decisions. As a result of this widespread adoption, it has become even more important to ensure that neural networks behave reliably and as expected. Unfortunately, reasoning about these systems is challenging due to their “black box” nature: it is difficult to understand what the network does since it is typically parameterized with thousands or millions of real-valued weights that are hard to interpret. Further, it has been discovered that neural nets can sometimes be surprisingly brittle and exhibit non-robust behaviors, for instance, by classifying two very similar inputs (e.g., images that differ only in brightness or in one pixel) to different labels [86].

To address the challenge of reasoning about neural networks, recent research has started exploring new methods and systems which can *automatically prove* that

a given network satisfies a specific property of interest (e.g., robustness to certain perturbations, pre/post conditions). State-of-the-art works include methods based on SMT solving [37, 69, 113, 114], mixed integer linear programming [8, 32, 36, 49, 66, 135, 197], Lipschitz optimization [170], duality [67, 212], convex relaxations [68, 78, 163, 175, 186, 199, 211], and combination of relaxations with solvers [206].

Despite the progress made by these works, more research is needed to reach the point where we can solve the overall neural network reasoning challenge successfully. In particular, we still lack an analyzer that can scale to large networks, can handle popular neural architectures (e.g., fully-connected, convolutional), and yet is sufficiently precise to prove relevant properties required by applications. For example, the works based on SMT solving and mixed-integer linear programming are precise yet can only handle very small networks. To mitigate the scalability issues, the work of [78] introduced the first neural network certifier based on abstract interpretation enabling the analysis of larger networks than solver-based methods. However, it relies on existing generic abstract domains that either do not scale to larger neural networks (such as Convex Polyhedra [57]) or are too imprecise (e.g., Zonotope [81]). We note that online decomposition described in Chapters 2 and 3 for improving the scalability of the Polyhedra domain does not work in the neural network setting as the transformations in neural networks create constraints between all neurons. Recent work by [211] scales better than [78] but only handles fully-connected networks and cannot handle the widely used convolutional networks. Both [113] and [211] are, in fact, unsound for floating-point arithmetic, which is heavily used in neural nets, and thus they can suffer from false negatives. Recent work by [186] handles fully-connected and convolutional networks and is sound for floating-point arithmetic; however, as we demonstrate experimentally, it can lose significant precision when dealing with larger perturbations.

**THIS WORK** In this work, we propose a new polyhedral domain, called DeepPoly, that makes a step forward in addressing the challenge of certifying neural networks with respect to both scalability and precision. The key technical idea behind our work is a novel abstract interpreter specifically tailored to the setting of neural networks. Concretely, our abstract domain is a combination of floating-point polyhedra with intervals, coupled with custom abstract transformers for common neural network functions such as affine transforms, the rectified linear unit (ReLU), sigmoid and tanh activations, and the maxpool operator. These abstract transformers are carefully designed to exploit key properties of these functions and balance analysis scalability and precision. As a result, DeepPoly is more precise than [211], [78] and [186], yet can handle large convolutional networks and is also sound for floating-point arithmetic.

**PROVING ROBUSTNESS: ILLUSTRATIVE EXAMPLES** To provide an intuition for the kinds of problems that DeepPoly can solve, consider the images shown in







Attack	Original	Lower	Upper
$L_\infty$			
Rotation			

Figure 5.1: Two different attacks applied to MNIST images.

Fig. 5.1. Here, we will illustrate two kinds of robustness properties:  $L_\infty$ -norm based perturbations (first row) and image rotations (second row).

In the first row, we are given an image of the digit 7 (under “Original”). Then, we consider an attack where we allow a small perturbation to *every pixel* in the original image (visually this may correspond to darkening or lightening the image). That is, instead of a number, each pixel now contains an interval. If each of these intervals has the same size, we say that we have formed an  $L_\infty$  ball around the image (typically with a given epsilon  $\epsilon \in \mathbb{R}$ ). This ball is captured visually by the Lower image (in which each pixel contains the smallest value allowed by its interval) and the Upper image (in which each pixel contains the largest value allowed by its interval). We call the modification of the original image to a perturbed version inside this ball an attack, reflecting an adversary who aims to trick the network. There have been various works which aim to find such an attack, otherwise called an adversarial example (e.g., [40]), typically using gradient-based methods. For our setting however, the question is: are all possible images “sitting between” the Lower and the Upper image classified to the same label as the original? Or, in other words, is the neural net robust to this kind of attack?

The set of possible images induced by the attack is also called an *adversarial region*. Note that enumerating all possible images in this region and simply running the network on each to check if it is classified correctly, is practically infeasible. For example, an image from the standard MNIST [124] dataset contains 784 pixels and a perturbation that allows for even two values for every pixel will lead to  $2^{784}$  images that one would need to consider. In contrast, our domain DeepPoly can *automatically prove* that all images in the adversarial region classify correctly (that is, no attack is possible) by soundly propagating the entire input adversarial region through the abstract transformers of the network.

We also consider a more complex type of perturbation in the second row. Here, we rotate the image by an angle and our goal is to show that any rotation up to this angle classifies to the same label. In fact, we consider an even more challenging problem where we not only rotate an image but first form an adversarial region around the image and then reason about all possible rotations of any image in that region. This is challenging, as again, the enumeration of images is infeasible when using geometric transformations that perform linear interpolation (which is needed to improve output image quality). Further, unlike the  $L_\infty$  ball above, the entire set of possible images represented by a rotation up to a given angle does not have a

closed-form and needs to somehow be captured. Directly approximating this set is too imprecise and the analysis fails to prove the wanted property. Thus, we introduce a method where we *refine* the initial approximation into smaller regions that correspond to smaller angles (a form of trace partitioning [169]), use DeepPoly to prove the property on each smaller region, and then deduce the property holds for the initial, larger approximation. To our best knowledge, this is the first work that shows how to prove the robustness of a neural network under complex input perturbations such as rotations.

The work in this chapter was published in [188].

**MAIN CONTRIBUTIONS** Our main contributions are:

- A new abstract domain for the certification of neural nets. The domain combines floating-point polyhedra and intervals with custom abstract transformers for affine transforms, ReLU, sigmoid, tanh, and maxpool functions. These abstract transformers carefully balance the scalability and precision of the analysis (Section 5.3).
- An approach for proving more complex perturbation specifications than considered so far, including rotations using linear interpolation, based on the refinement of the abstract input. To our best knowledge, this is the first time such perturbations have been certified (Section 5.4).
- A complete, parallelized implementation of our approach in a system called ERAN fully available at <https://github.com/eth-sri/eran>. ERAN can handle both fully-connected and convolutional neural networks (Section 5.5).
- An extensive evaluation on a range of datasets and networks including defended ones, showing DeepPoly is more precise than prior work yet scales to large networks (Section 5.5).

We note that in [152], we introduced new abstract transformers for handling residual layers with DeepPoly integrated into ERAN. Overall, we believe that DeepPoly is a promising step towards addressing the challenge of reasoning about neural networks and a useful building block for proving complex specifications (e.g., rotations) and other applications of analysis. As an example, because our abstract transformers for the output of a neuron are “point-wise” (i.e., can be computed in parallel), they can be implemented on GPUs. In [152], we designed efficient GPU algorithms for DeepPoly, and the resulting implementation which we call GPUPoly enables the analysis of a 34-layer deep neural network containing up to 1M neurons within a minute. In the future, we believe our transformers can also be directly plugged into the latest systems training robust neural networks using abstract interpretation on GPUs [144, 148]. As our transformers are substantially more precise in practice than those from [144, 148], we expect they can help improve the overall robustness of the trained network.

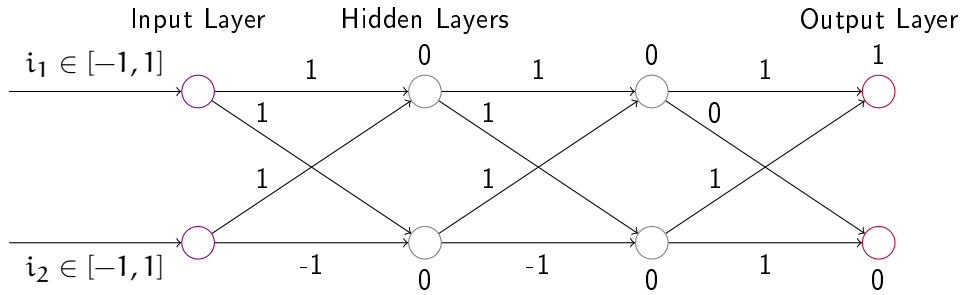


Figure 5.2: Example fully-connected neural network with ReLU activations.

## 5.1 OVERVIEW

In this section, we provide an overview of our abstract domain on a small illustrative example. Full formal details are provided in later sections.

### 5.1.1 Running example on a fully-connected network with ReLU activation

We consider the simple fully-connected neural network with ReLU activations shown in Fig. 5.2. This network has already been trained and we have the learned weights shown in the figure. The network consists of four layers: an input layer, two hidden layers, and an output layer with two neurons each. The weights on the edges represent the learned coefficients of the weight matrix used by the affine transformations done at each layer. Note that these values are usually detailed floating point numbers (e.g., 0.03), however, here we use whole numbers to simplify the presentation. The learned bias for each neuron is shown above or below it. All of the biases in one layer constitute the translation vector of the affine transformation.

To compute its output, each neuron in the hidden layer applies an affine transformation based on the weight matrix and bias to its inputs (these inputs are the outputs of the neurons in the previous layer), producing a value  $v$ . Then, the neuron applies an activation function to  $v$ , in our example ReLU, which outputs  $v$ , if  $v > 0$ , and 0 otherwise. Thus, the input to every neuron goes through two stages: first, an affine transformation, followed by an activation function application. In the last layer, a final affine transform is applied to yield the output of the entire network, typically a class label that describes how the input is classified.

**SPECIFICATION** Suppose we work with a hypothetical image that contains only two pixels and the perturbation is such that it places both pixels in the range  $[-1, 1]$  (pixels are usually in the range  $[0, 1]$ , however, we use  $[-1, 1]$  to better illustrate our analysis). Our goal will be to prove that the output of the network at one of the output neurons is always greater than the output at the other one, for any possible input of two pixels in the range  $[-1, 1]$ . If the proof is successful, it implies that the network produces the same classification label for all of these images.

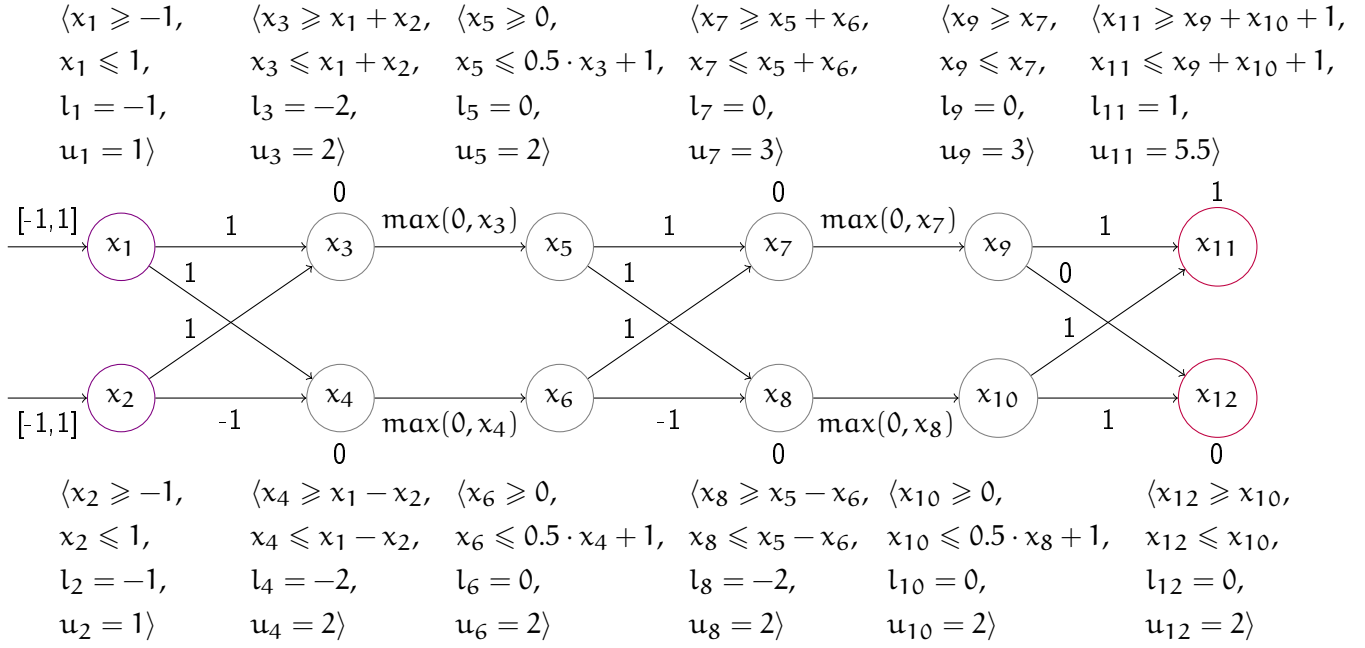


Figure 5.3: The neural network from Fig. 5.2 transformed for analysis with the DeepPoly abstract domain.

**ABSTRACT DOMAIN** To perform the analysis, we introduce an abstract domain with the appropriate abstract transformers that propagate the (abstract) input of the network through the layers, computing an over-approximation of the possible values at each neuron. Concretely, for our example, we need to propagate both intervals  $[-1, 1]$  (one for each pixel) simultaneously. We now briefly discuss our abstract domain, which aims to balance analysis scalability and precision. Then we illustrate its effect on our example network and discuss why we made certain choices in the approximation over others.

To perform the analysis, we first rewrite the network by expanding each neuron into two nodes: one for the associated affine transform and one for the ReLU activation. Transforming the network of Fig. 5.2 in this manner produces the network shown in Fig. 5.3. Because we assign a variable to each node, the network of Fig. 5.3 consists of  $n = 12$  variables. Our abstract domain, formally described in Section 5.3, associates two constraints with each variable  $x_i$ : an upper polyhedral constraint and a lower polyhedral constraint. Additionally, the domain tracks auxiliary (concrete) bounds, one upper bound and one lower bound for each variable, describing a bounding box of the concretization of the abstract element. Our domain is less expressive than the Polyhedra domain [57] because it bounds the number of conjuncts that can appear in the overall formula to  $2 \cdot n$  where  $n$  is the number of variables of the network. Such careful restrictions are necessary because supporting the full expressive power of convex polyhedra leads to an exponential number of constraints that make the analysis for thousands of neurons practically infeasible. We now discuss the two types of constraints and the two types of bounds, and show how they are computed on our example.



First, the lower ( $\alpha_i^{\leq}$ ) and upper ( $\alpha_i^{\geq}$ ) relational polyhedral constraints associated with  $x_i$  have the form  $v + \sum_j w_j \cdot x_j$  where  $v \in \mathbb{R} \cup \{-\infty, +\infty\}$ ,  $w \in \mathbb{R}^n$ ,  $\forall j \geq i$ .  $w_j = 0$ . That is, a polyhedral constraint for  $x_i$  can consider and refer to variables “before”  $x_i$  in the network, but cannot refer to variables “after”  $x_i$  (because their coefficient is set to 0). Second, for the concrete lower and upper bounds of  $x_i$ , we use  $l_i, u_i \in \mathbb{R} \cup \{-\infty, +\infty\}$ , respectively. All abstract elements  $a$  in our domain satisfy the additional invariant that the interval  $[l_i, u_i]$  overapproximates the set of values that the variable  $x_i$  can take (we formalize this requirement in Section 5.3).

**ABSTRACT INTERPRETATION OF THE NETWORK** We now illustrate the operation of our abstract interpreter (using the DeepPoly domain above) on our example network, abstract input ( $[-1, 1]$  for both pixels), and specification (which is to prove that any image in the concretization of  $[-1, 1] \times [-1, 1]$  classifies to the same label).

The analysis starts at the input layer, i.e., in our example from  $x_1$  and  $x_2$ , and simply propagates the inputs, resulting in  $\alpha_1^{\leq} = \alpha_2^{\leq} = -1$ ,  $\alpha_1^{\geq} = \alpha_2^{\geq} = 1$ ,  $l_1 = l_2 = -1$ , and  $u_1 = u_2 = 1$ . Next, the affine transform at the first layer updates the constraints for  $x_3$  and  $x_4$ . The abstract transformer first adds the constraints:

$$\begin{aligned} x_1 + x_2 &\leq x_3 \leq x_1 + x_2 \\ x_1 - x_2 &\leq x_4 \leq x_1 - x_2 \end{aligned} \tag{5.1}$$

The transformer uses these constraints and the constraints for  $x_1, x_2$  to compute  $l_3 = l_4 = -2$  and  $u_3 = u_4 = 2$ .

Next, the transformer for the ReLU activation is applied. In general, the output  $x_j$  of the ReLU activation on variable  $x_i$  is equivalent to the assignment  $x_j := \max(0, x_i)$ . If  $u_i \leq 0$ , then our abstract transformer sets the state of the variable  $x_j$  to  $0 \leq x_j \leq 0$ ,  $l_j = u_j = 0$ . In this case, our abstract transformer is exact. If  $l_i \geq 0$ , then our abstract transformer adds  $x_i \leq x_j \leq x_i$ ,  $l_j = l_i$ ,  $u_j = u_i$ . Again, our abstract transformer is exact in this case.

However, when  $l_i < 0$  and  $u_i > 0$ , the result cannot be captured exactly by our abstraction and we need to decide how to lose information. Fig. 5.4 shows several candidate convex approximations of the ReLU assignment in this case. The approximation [69] of Fig. 5.4 (a) minimizes the area in the  $x_i, x_j$  plane, and would add the following relational constraints and concrete bounds for  $x_j$ :

$$\begin{aligned} x_i &\leq x_j, 0 \leq x_j, \\ x_j &\leq u_i \cdot (x_i - l_i) / (u_i - l_i). \\ l_j &= 0, u_j = u_i. \end{aligned} \tag{5.2}$$

However, the approximation in (5.2) contains two lower polyhedra constraints for  $x_j$ , which we disallow in our abstract domain. The reason for this is the potential blowup of the analysis cost as it proceeds. We will explain this effect in more detail later in this section.

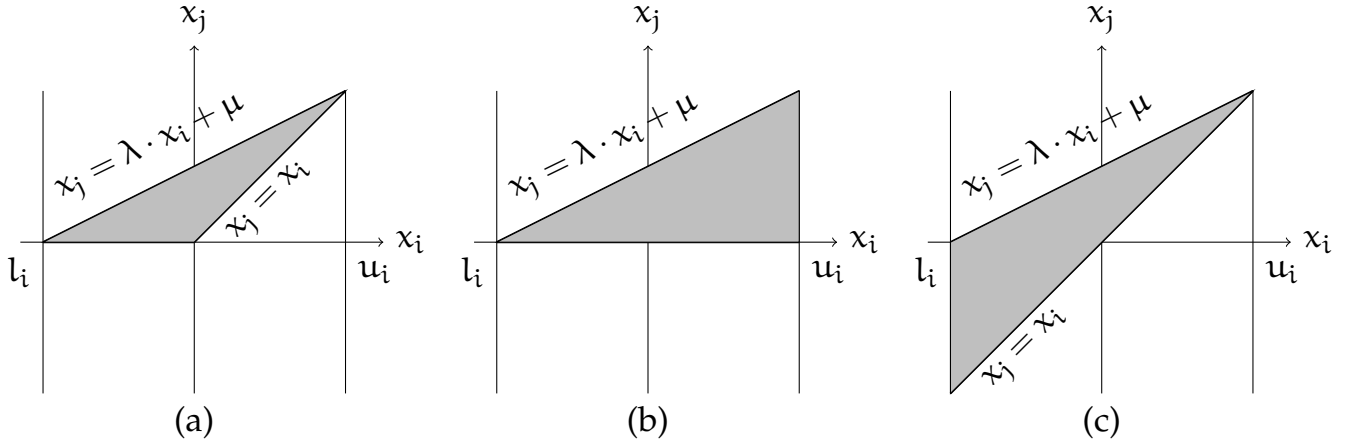


Figure 5.4: Convex approximations for the ReLU function: (a) shows the convex approximation [69] with the minimum area in the input-output plane, (b) and (c) show the two convex approximations used in DeepPoly. In the figure,  $\lambda = u_i / (u_i - l_i)$  and  $\mu = -l_i \cdot u_i / (u_i - l_i)$ .

To avoid this explosion we further approximate (5.2) by allowing only one lower bound. There are two ways of accomplish this, shown in Fig. 5.4 (b) and (c), both of which can be expressed in our domain. During analysis we always consider both and choose the one with the least area dynamically.

The approximation from Fig. 5.4 (b) adds the following constraints and bounds for  $x_j$ :

$$\begin{aligned} 0 &\leq x_j \leq u_i \cdot (x_i - l_i) / (u_i - l_i), \\ l_j &= 0, u_j = u_i. \end{aligned} \quad (5.3)$$

The approximation from Fig. 5.4 (c) adds the following constraints and bounds:

$$\begin{aligned} x_i &\leq x_j \leq u_i \cdot (x_i - l_i) / (u_i - l_i), \\ l_j &= l_i, u_j = u_i. \end{aligned} \quad (5.4)$$

Note that it would be incorrect to set  $l_j = 0$  in (5.4) above (instead of  $l_j = l_i$ ). The reason is that this would break a key domain invariant which we aim to maintain, namely that the concretization of the two symbolic bounds for  $x_j$  is contained inside the concretization of the concrete bounds  $l_j$  and  $u_j$  (we discuss this domain invariant later in Section 5.3). In particular, if we only consider the two symbolic bounds for  $x_j$ , then  $x_j$  would be allowed to take on negative values and these negative values would not be included in the region  $[0, u_i]$ . This domain invariant is important to ensure efficiency of our transformers and as we prove later, all of our abstract transformers maintain it.

Returning to our example, the area of the approximation in Fig. 5.4 (b) is  $0.5 \cdot u_i \cdot (u_i - l_i)$  whereas the area in Fig. 5.4 (c) is  $0.5 \cdot -l_i \cdot (u_i - l_i)$ . We choose the tighter approximation; i.e., when  $u_i \leq -l_i$ , we add the constraints and the bounds from (5.3); otherwise, we add the constraints and the bounds from (5.4). We note

that the approximations in Fig. 5.4 (b) and (c) cannot be captured by the Zonotope abstraction as used in [78, 191].

In our example, for both  $x_3$  and  $x_4$ , we have  $l_3 = l_4 = -2$  and  $u_3 = u_4 = 2$ . The areas are equal in this case; thus we choose (5.3) and get the following constraints and bounds for  $x_5$  and  $x_6$ :

$$\begin{aligned} 0 \leq x_5 \leq 0.5 \cdot x_3 + 1, \quad l_5 = 0, \quad u_5 = 2, \\ 0 \leq x_6 \leq 0.5 \cdot x_4 + 1, \quad l_6 = 0, \quad u_6 = 2. \end{aligned} \quad (5.5)$$

Next, we apply the abstract affine transformer, which first adds the following constraints for  $x_7$  and  $x_8$ :

$$\begin{aligned} x_5 + x_6 \leq x_7 \leq x_5 + x_6, \\ x_5 - x_6 \leq x_8 \leq x_5 - x_6. \end{aligned} \quad (5.6)$$

It is possible to compute bounds for  $x_7$  and  $x_8$  from the above equations by substituting the concrete bounds for  $x_5$  and  $x_6$ . However, the resulting bounds are in general too imprecise. Instead, we can obtain better bounds by recursively substituting the polyhedral constraints until the bounds only depend on the input variables for which we then use their concrete bounds. In our example we substitute the relational constraints for  $x_5, x_6$  from equation (5.5) to obtain:

$$\begin{aligned} 0 \leq x_7 \leq 0.5 \cdot x_3 + 0.5 \cdot x_4 + 2, \\ -0.5 \cdot x_4 - 1 \leq x_8 \leq 0.5 \cdot x_3 + 1. \end{aligned} \quad (5.7)$$

Replacing  $x_3$  and  $x_4$  with the constraints in (5.1), we get:

$$\begin{aligned} 0 \leq x_7 \leq x_1 + 2, \\ -0.5 \cdot x_1 + 0.5 \cdot x_2 - 1 \leq x_8 \leq 0.5 \cdot x_1 + 0.5 \cdot x_2 + 1. \end{aligned} \quad (5.8)$$

Now we use the concrete bounds of  $\pm 1$  for  $x_1, x_2$  to obtain  $l_7 = 0, u_7 = 3$  and  $l_8 = -2, u_8 = 2$ . Indeed, this is more precise than if we had directly substituted the concrete bounds for  $x_5$  and  $x_6$  in (5.6) because that would have produced concrete bounds  $l_7 = 0, u_7 = 4$  (which are not as tight as the ones above).

**AVOIDING EXPONENTIAL BLOWUP OF THE ANALYSIS COST** As seen above, to avoid exponential cost, our analysis introduces exactly one polyhedral constraint for the lower bound of a variable. It is instructive to understand the effect of introducing more than one constraint via the ReLU approximation of Fig. 5.4 (a). This ReLU approximation introduces two lower relational constraints for both  $x_5$  and  $x_6$ . Substituting them in (5.6) would have created four lower relational constraints for  $x_7$ . More generally, if the affine expression for a variable  $x_i$  contains  $p$  variables with positive coefficients and  $n$  variables with negative coefficients, then the number of possible lower and upper relational constraints after substitution is  $2^p$  and  $2^n$ , respectively, leading to an exponential blowup. This is the reason why we keep only one lower relational constraint for each variable in the network, which creates exactly one lower and upper relational constraint after substitution, and use either the ReLU transformer illustrated in Fig. 5.4 (b) or the one in Fig. 5.4 (c).

**ASYMPTOTIC RUNTIME** The computation of concrete bounds by the abstract affine transformer in the hidden layers is the most expensive step of our analysis. If there are  $L$  network layers and the maximum number of variables in a layer is  $n_{\max}$ , then this step for one variable is in  $O(n_{\max}^2 \cdot L)$ . Storing the concrete bounds ensures that the subsequent ReLU transformer has constant cost.

All our transformers work point-wise for the variables in a layer; i.e., they are independent for different variables since they only read constraints and bounds from the previous layers. This makes it possible to parallelize our analysis on both CPUs and GPUs. The work of [144] defines pointwise Zonotope transformers for training neural networks on GPUs to be more robust against adversarial attacks. The more precise Zonotope transformers of [186] were used for training more robust networks than [144] in [148]. Our pointwise transformers are more precise in practice than those from [144, 186]. We believe that our transformers can be used to train even more robust neural networks in the future.

**PRECISION VS. PERFORMANCE TRADE-OFF** We also note that our approach allows one to easily vary the precision-performance knob of the affine transformer in the hidden layers: (i) we can select a subset of variables for which to perform complete substitution all the way back to the first layer (the example above showed this for all variables), and (ii) we can decide at which layer we would like to stop the substitution and select the concrete bounds at that layer.

Returning to our example, next, the ReLU transformers are applied again. Since  $l_7 = 0$ , the ReLU transformer is exact for the assignment to  $x_9$  and adds the relational constraints  $x_7 \leq x_9 \leq x_7$  and the bounds  $l_9 = 0, u_9 = 3$  for  $x_9$ . However, the transformer is not exact for the assignment to  $x_{10}$  and the following constraints and bounds for  $x_{10}$  are added:

$$\begin{aligned} 0 &\leq x_{10} \leq 0.5 \cdot x_8 + 1, \\ l_{10} &= 0, u_{10} = 2. \end{aligned} \tag{5.9}$$

Finally, the analysis reaches the output layer and the abstract affine transformer adds the following constraints for  $x_{11}$  and  $x_{12}$ :

$$\begin{aligned} x_9 + x_{10} + 1 &\leq x_{11} \leq x_9 + x_{10} + 1 \\ x_{10} &\leq x_{12} \leq x_{10} \end{aligned} \tag{5.10}$$

Again, backsubstitution up to the input layer yields  $l_{11} = 1, u_{11} = 5.5$  and  $l_{12} = 0, u_{12} = 2$ . This completes our analysis of the neural network.

**CHECKING THE SPECIFICATION** Next, we check our specification, namely whether all concrete output values of one neuron are always greater than all concrete output values of the other neuron, i.e., if

$$\begin{aligned} \forall i_1, i_2 \in [-1, 1] \times [-1, 1], x_{11} &> x_{12} \text{ or} \\ \forall i_1, i_2 \in [-1, 1] \times [-1, 1], x_{12} &> x_{11}, \end{aligned}$$

where  $x_{11}, x_{12} = N_{fc}(i_1, i_2)$  are the concrete values for variables  $x_{11}$  and  $x_{12}$  produced by our small fully-connected (fc) neural network  $N_{fc}$  for inputs  $i_1, i_2$ .

In our simple example, this amounts to proving whether  $x_{11} - x_{12} > 0$  or  $x_{12} - x_{11} > 0$  holds given the abstract results computed by our analysis. Note that using the concrete bounds for  $x_{11}$  and  $x_{12}$ , that is,  $l_{11}, l_{12}, u_{11}$ , and  $u_{12}$  leads to the bound  $[-1, 5.5]$  for  $x_{11} - x_{12}$  and  $[-5.5, 1]$  for  $x_{12} - x_{11}$  and hence we cannot prove that either constraint holds. To address this imprecision, we first create a new temporary variable  $x_{13}$  and apply our abstract transformer for the assignment  $x_{13} := x_{11} - x_{12}$ . Our transformer adds the following constraint:

$$x_{11} - x_{12} \leq x_{13} \leq x_{11} - x_{12} \quad (5.11)$$

The transformer then computes bounds for  $x_{13}$  by backsubstitution (to the first layer), as described so far, which produces  $l_{13} = 1$  and  $u_{13} = 4$ . As the (concrete) lower bound of  $x_{13}$  is greater than 0, our analysis concludes that  $x_{11} - x_{12} > 0$  holds. Hence, we have proved our (robustness) specification. Of course, if we had failed to prove the property, we would have tried the same analysis using the second constraint (i.e.,  $x_{12} > x_{11}$ ). And if that would fail, then we would declare that we are unable to prove the property. For our example, this was not needed since we were able to prove the first constraint.

## 5.2 BACKGROUND: NEURAL NETWORKS AND ADVERSARIAL REGIONS

In this section, we provide the minimal necessary background on neural networks and adversarial regions. Further, we show how we represent neural networks for our analysis.

**NEURAL NETWORKS** Neural networks are functions  $N: \mathbb{R}^m \rightarrow \mathbb{R}^n$  that can be implemented using straight-line programs (i.e., without loops) of a certain form. In this work, we focus on neural networks that follow a *layered architecture*, but all our methods can be used unchanged for more general neural network shapes. A layered neural network is given by a composition of  $l$  layers  $f_1: \mathbb{R}^m \rightarrow \mathbb{R}^{n_1}, \dots, f_l: \mathbb{R}^{n_{l-1}} \rightarrow \mathbb{R}^n$ . Each layer  $f_i$  is one of the following: (i) an affine transformation  $f_i(x) = Ax + b$  for some  $A \in \mathbb{R}^{n_i \times n_{i-1}}$  and  $b \in \mathbb{R}^{n_i}$  (in particular, convolution with one or more filters is an affine transformation), (ii) the ReLU activation function  $f(x) = \max(0, x)$ , where the maximum is applied componentwise, (iii) the sigmoid ( $\sigma(x) = \frac{e^x}{e^x + 1}$ ) or the tanh ( $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ ) activation function (again applied componentwise), or (iv) a maxpool operator, which subdivides the input  $x$  into multiple parts, and returns the maximal value in each part.

**NEURONS AND ACTIVATIONS** Each component of one of the vectors passed along through the layers is called a *neuron*, and its value is called an *activation*.

There are three types of neurons:  $m$  input neurons whose activations form the input to the network,  $n$  output neurons whose activations form the output of the network, and all other neurons, called *hidden*, as they are not directly observed.

**CLASSIFICATION** For a neural network that classifies its inputs to multiple possible labels,  $n$  is the number of distinct classes, and the neural network classifies a given input  $x$  to a given class  $k$  if  $N(x)_k > N(x)_j$  for all  $j$  with  $1 \leq j \leq n$  and  $j \neq k$ .

**ADVERSARIAL REGION** In our evaluation, we consider the following (standard, e.g., see [40]) threat model: an input is drawn from the input distribution, perturbed by an adversary and then classified by the neural network. The perturbations that the adversary can perform are restricted and the set  $X \subseteq \mathbb{R}^n$  of possible perturbations for a given input is called an *adversarial region*. The maximal possible error (i.e., the fraction of misclassified inputs) that the adversary can obtain by picking a worst-case input from each adversarial region is called the *adversarial error*. A neural network is *robust* for a given adversarial region if it classifies all inputs in that region the same way. This means that it is impossible for an adversary to influence the classification by picking an input from the adversarial region.

In our evaluation, we focus on certifying robustness for adversarial regions that can be represented using a set of interval constraints, i.e.,  $X = \times_{i=1}^m [l_i, u_i]$  for  $l_i, u_i \in \mathbb{R} \cup \{-\infty, +\infty\}$ . We also show how to use our analyzer to certify robustness against rotations which employ linear interpolation.

**NETWORK REPRESENTATION** For our analysis, we represent neural networks as a sequence of assignments, one per hidden and per output neuron. We need four kinds of assignments: ReLU assignments  $x_i \leftarrow \max(0, x_j)$ , sigmoid/tanh assignments  $x_i \leftarrow g(x_j)$  for  $g = \sigma$  or  $g = \tanh$ , maxpool assignments  $x_i \leftarrow \max_{j \in J} x_j$  and affine assignments  $x_i \leftarrow v + \sum_j w_j \cdot x_j$ . Convolutional layers can be described with affine assignments [78].

For example, we represent the neural network from Fig. 5.2 as the following program:

$$\begin{aligned} x_3 &\leftarrow x_1 + x_2, x_4 \leftarrow x_1 - x_2, x_5 \leftarrow \max(0, x_3), x_6 \leftarrow \max(0, x_4), \\ x_7 &\leftarrow x_5 + x_6, x_8 \leftarrow x_5 - x_6, x_9 \leftarrow \max(0, x_7), x_{10} \leftarrow \max(0, x_8), \\ x_{11} &\leftarrow x_9 + x_{10} + 1, x_{12} \leftarrow x_{10}. \end{aligned}$$

In Fig. 5.2, the adversarial region is given by  $X = [-1, 1] \times [-1, 1]$ . The variables  $x_1$  and  $x_2$  are the input to the neural network, and the variables  $x_{11}$  and  $x_{12}$  are the outputs of the network. Therefore, the final class of an input  $(x_1, x_2)$  is 1 if  $x_{11} > x_{12}$ , and 2 if  $x_{11} < x_{12}$ . To prove the considered specification, we either need to prove that  $\forall (x_1, x_2) \in X. x_{11} > x_{12}$  or that  $\forall (x_1, x_2) \in X. x_{12} > x_{11}$ .

We note that even though our experimental evaluation focuses on different kinds of robustness, our method and abstract domain are general and can also be used to prove other properties, such as those in Fig. 1.7.

### 5.3 ABSTRACT DOMAIN AND TRANSFORMERS

In this section, we introduce our abstract domain as well as the abstract transformers needed to analyze the four kinds of assignment statements mentioned in Section 5.2.

Elements in our abstract domain  $\mathcal{A}_n$  consist of a set of polyhedral constraints of a specific form, over  $n$  variables. Each constraint relates one variable to a linear combination of the variables of a smaller index. Each variable has two associated polyhedral constraints: one lower bound and one upper bound. In addition, the abstract element records derived interval bounds for each variable. Formally, an abstract element  $\mathbf{a} \in \mathcal{A}_n$  over  $n$  variables can be written as a tuple  $\mathbf{a} = \langle \mathbf{a}^{\leq}, \mathbf{a}^{\geq}, \mathbf{l}, \mathbf{u} \rangle$  where

$$\mathbf{a}_i^{\leq}, \mathbf{a}_i^{\geq} \in \{x \mapsto v + \sum_{j \in [i-1]} w_j \cdot x_j \mid v \in \mathbb{R} \cup \{-\infty, +\infty\}, w \in \mathbb{R}^{i-1}\} \text{ for } i \in [n]$$

and  $\mathbf{l}, \mathbf{u} \in (\mathbb{R} \cup \{-\infty, +\infty\})^n$ . Here, we use the notation  $[n] := \{1, 2, \dots, n\}$ . The concretization function  $\gamma_n: \mathcal{A}_n \rightarrow \mathcal{P}(\mathbb{R}^n)$  is then given by

$$\gamma_n(\mathbf{a}) = \{x \in \mathbb{R}^n \mid \forall i \in [n]. \mathbf{a}_i^{\leq}(x) \leq x_i \wedge \mathbf{a}_i^{\geq}(x) \geq x_i\}.$$

**DOMAIN INVARIANT** All abstract elements in our domain additionally satisfy the following invariant:  $\gamma_n(\mathbf{a}) \subseteq \times_{i \in [n]} [l_i, u_i]$ . In other words, every abstract element in our domain maintains concrete lower and upper bounds which over-approximate the two symbolic bounds. This property is essential for creating efficient abstract transformers.

To simplify our exposition of abstract transformers, we will only consider the case where all variables are bounded, which is always the case when our analysis is applied to neural networks. Further, we require that variables are assigned exactly once, in increasing order of their indices. Our abstract transformers  $T_f^\#$  for a deterministic function  $f: \mathcal{A}^m \rightarrow \mathcal{A}^n$  satisfy the following soundness property based on Definition 3.1.1:  $T_f(\gamma_m(\mathbf{a})) \subseteq \gamma_n(T_f^\#(\mathbf{a}))$  for all  $\mathbf{a} \in \mathcal{A}^m$ , where  $T_f$  is the corresponding concrete transformer of  $f$ , given by  $T_f(X) = \{f(x) \mid x \in X\}$ .

#### 5.3.1 ReLU Abstract Transformer

Let  $f: \mathbb{R}^{i-1} \rightarrow \mathbb{R}^i$  be a function that executes the assignment  $x_i \leftarrow \max(0, x_j)$  for some  $j < i$ . The corresponding abstract ReLU transformer is  $T_f^\#(\langle \mathbf{a}^{\leq}, \mathbf{a}^{\geq}, \mathbf{l}, \mathbf{u} \rangle) = \langle \mathbf{a}'^{\leq}, \mathbf{a}'^{\geq}, \mathbf{l}', \mathbf{u}' \rangle$  where  $\mathbf{a}'_k^{\leq} = \mathbf{a}_k^{\leq}$ ,  $\mathbf{a}'_k^{\geq} = \mathbf{a}_k^{\geq}$ ,  $l'_k = l_k$  and  $u'_k = u_k$  for  $k < i$ . For the new component  $i$ , there are three cases. If  $u_j \leq 0$ , then  $\mathbf{a}'_i^{\leq}(x) = \mathbf{a}'_i^{\geq}(x) = 0$  and  $l'_i = u'_i = 0$ . If  $0 \leq l_j$ , then  $\mathbf{a}'_i^{\leq}(x) = \mathbf{a}'_i^{\geq}(x) = x_j$ ,  $l'_i = l_j$  and  $u'_i = u_j$ .

Otherwise, the abstract ReLU transformer approximates the assignment by a set of linear constraints forming the convex hull of the polyhedra obtained by intersecting the interval constraints on the input  $l_j \leq x_j \leq u_j$  with the constraints

from the two ReLU branches, i.e., the convex hull of  $\{l_j \leq x_j \leq u_j, x_j \leq 0, x_i = 0\}$  and  $\{l_j \leq x_j \leq u_j, x_j \geq 0, x_i = x_j\}$ :

$$\begin{aligned} 0 &\leq x_i, \quad x_j \leq x_i, \\ x_i &\leq u_j \cdot (x_j - l_j) / (u_j - l_j). \end{aligned}$$

As there is only one upper bound for  $x_i$ , we obtain the following rule:

$$a_i^{\geq}(\mathbf{x}) = u_j \cdot (x_j - l_j) / (u_j - l_j).$$

On the other hand, we have two lower bounds for  $x_i$ :  $x_j$  and 0. Any convex combination of those two constraints is still a valid lower bound. Therefore, we can set

$$a_i^{\leq}(\mathbf{x}) = \lambda \cdot x_j,$$

for any  $\lambda \in [0, 1]$ . We select the  $\lambda \in [0, 1]$  that minimizes the area of the resulting shape in the  $(x_i, x_j)$ -plane. Finally, we set  $l_i' = \lambda \cdot l_j$  and  $u_i' = u_j$ .

### 5.3.2 Sigmoid and Tanh Abstract Transformers

Let  $g: \mathbb{R} \rightarrow \mathbb{R}$  be a continuous, twice-differentiable function with  $g'(x) > 0$  and  $0 \leq g''(x) \Leftrightarrow x \leq 0$  for all  $x \in \mathbb{R}$  where  $g'$  and  $g''$  are the first and second derivatives of  $g$ . The sigmoid function  $\sigma(x) = \frac{e^x}{e^x + 1}$  and the tanh function  $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$  both satisfy these conditions. For such a function  $g$ , let  $f: \mathbb{R}^{i-1} \rightarrow \mathbb{R}^i$  be the function that executes the assignment  $x_i \leftarrow g(x_j)$  for  $j < i$ .

The corresponding abstract transformer is  $T_f^\#(\langle a^{\leq}, a^{\geq}, l, u \rangle) = \langle a_i^{\leq}, a_i^{\geq}, l', u' \rangle$  where  $a_k^{\leq} = a_k^{\leq}$ ,  $a_k^{\geq} = a_k^{\geq}$ ,  $l'_k = l_k$  and  $u'_k = u_k$  for  $k < i$ . For the new component  $i$ , we set  $l'_i = g(l_j)$  and  $u'_i = g(u_j)$ . If  $l_j = u_j$ , then  $a_i^{\leq}(\mathbf{x}) = a_i^{\geq}(\mathbf{x}) = g(l_j)$ . Otherwise, we consider  $a_i^{\leq}(\mathbf{x})$  and  $a_i^{\geq}(\mathbf{x})$  separately. Let  $\lambda = (g(u_j) - g(l_j)) / (u_j - l_j)$  and  $\lambda' = \min(g'(l_j), g'(u_j))$ . If  $0 < l_j$ , then  $a_i^{\leq}(\mathbf{x}) = g(l_j) + \lambda \cdot (x_j - l_j)$ , otherwise  $a_i^{\leq}(\mathbf{x}) = g(l_j) + \lambda' \cdot (x_j - l_j)$ . Similarly, if  $u_j \leq 0$ , then  $a_i^{\geq}(\mathbf{x}) = g(u_j) + \lambda \cdot (x_j - u_j)$  and  $a_i^{\geq}(\mathbf{x}) = g(u_j) + \lambda' \cdot (x_j - u_j)$  otherwise.

### 5.3.3 Maxpool Abstract Transformer

Let  $f: \mathbb{R}^{i-1} \rightarrow \mathbb{R}^i$  be a function that executes  $x_i \leftarrow \max_{j \in J} x_j$  for some  $J \subseteq [i-1]$ . The corresponding abstract maxpool transformer is  $T_f^\#(\langle a^{\leq}, a^{\geq}, l, u \rangle) = \langle a_i^{\leq}, a_i^{\geq}, l', u' \rangle$  where  $a_k^{\leq} = a_k^{\leq}$ ,  $a_k^{\geq} = a_k^{\geq}$ ,  $l'_k = l_k$  and  $u'_k = u_k$  for  $k < i$ . For the new component  $i$ , there are two cases. If there is some  $k \in J$  with  $u_j < l_k$  for all  $j \in J \setminus \{k\}$ , then  $a_i^{\leq}(\mathbf{x}) = a_i^{\geq}(\mathbf{x}) = x_k$ ,  $l'_i = l_k$  and  $u'_i = u_k$ . Otherwise, we choose  $k \in J$  such that  $l_k$  is maximized and set  $a_i^{\leq}(\mathbf{x}) = x_k$ ,  $l'_i = l_k$  and  $a_i^{\geq}(\mathbf{x}) = u'_i = \max_{j \in J} u_j$ .



### 5.3.4 Affine Abstract Transformer

Let  $f: \mathbb{R}^{i-1} \rightarrow \mathbb{R}^i$  be a function that executes  $x_i \leftarrow v + \sum_{j \in [i-1]} w_j \cdot x_j$  for some  $w \in \mathbb{R}^{i-1}$ . The corresponding abstract affine transformer is  $T_f^\#(\langle a^\leq, a^\geq, l, u \rangle) = \langle a'^\leq, a'^\geq, l', u' \rangle$  where  $a'_k{}^\leq = a_k^\leq$ ,  $a'_k{}^\geq = a_k^\geq$ ,  $l'_k = l_k$  and  $u'_k = u_k$  for  $k < i$ . Further,  $a'_i{}^\leq(x) = a'_i{}^\geq(x) = v + \sum_{j \in [i-1]} w_j \cdot x_j$ .

To compute  $l_i$  and  $u_i$ , we repeatedly substitute bounds for  $x_j$  into the constraint, until no further substitution is possible. Formally, if we want to obtain  $l'_i$ , we start with  $b_1(x) = a'_i{}^\leq(x)$ . If we have  $b_s(x) = v' + \sum_{j \in [k]} w'_j \cdot x_j$  for some  $k \in [i-1]$ ,  $v' \in \mathbb{R}$ ,  $w' \in \mathbb{R}^k$ , then

$$b_{s+1}(x) = v' + \sum_{j \in [k]} \left( \max(0, w'_j) \cdot a'_j{}^\leq(x) + \min(w'_j, 0) \cdot a'_j{}^\geq(x) \right).$$

We iterate until we reach  $b_{s'}$  with  $b_{s'}(x) = v''$  (i.e.,  $s'$  is the smallest number with this property). We then set  $l'_i = v''$ .

We compute  $u'_i$  in an analogous fashion: to obtain  $u'_i$ , we start with  $c_1(x) = a'_i{}^\geq(x)$ . If we have  $c_t(x) = v' + \sum_{j \in [k]} w'_j \cdot x_j$  for some  $k \in [i-1]$ ,  $v' \in \mathbb{R}$ ,  $w' \in \mathbb{R}^k$ , then

$$c_{t+1}(x) = v' + \sum_{j \in [k]} \left( \max(0, w'_j) \cdot a'_j{}^\geq(x) + \min(w'_j, 0) \cdot a'_j{}^\leq(x) \right).$$

We iterate until we reach  $c_{t'}$  with  $c_{t'}(x) = v''$ . We then set  $u'_i = v''$ .

### 5.3.5 Neural Network Robustness Analysis

We now show how to use our analysis to prove robustness of a neural network with  $p$  inputs,  $q$  hidden activations and  $r$  output classes, resulting in a total of  $p + q + r$  activations. More explicitly, our goal is to prove that the neural network classifies all inputs satisfying the given interval constraints (the adversarial region) to a particular class  $k$ .

We first create an abstract element  $a = \langle a^\leq, a^\geq, l, r \rangle$  over  $p$  variables, where  $a_i^\leq(x) = l_i$  and  $a_i^\geq(x) = u_i$  for all  $i$ . The bounds  $l_i$  and  $u_i$  are initialized such that they describe the adversarial region. For example, for the adversarial region in Fig. 5.2, we get

$$a = \langle (x \mapsto l_1, x \mapsto l_2), (x \mapsto u_1, x \mapsto u_2), (-1, -1), (1, 1) \rangle.$$

Then, the analysis proceeds by processing assignments for all  $q$  hidden activations and the  $r$  output activations of the neural network, layer by layer, processing nodes in ascending order of variable indices, using their respective abstract transformers. Finally, the analysis executes the following  $r-1$  (affine) assignments in the abstract:

$$\begin{aligned} x_{p+q+r+1} &\leftarrow x_{p+q+k} - x_{p+q+1}, \dots, x_{p+q+r+(k-1)} \leftarrow x_{p+q+k} - x_{p+q+(k-1)}, \\ x_{p+q+r+k} &\leftarrow x_{p+q+k} - x_{p+q+(k+1)}, \dots, x_{p+q+r+(r-1)} \leftarrow x_{p+q+k} - x_{p+q+r}. \end{aligned}$$

As output class  $k$  has the highest activation if and only if those differences are all positive, the neural network is proved robust if for all  $i \in \{p + q + r + 1, \dots, p + q + r + (r - 1)\}$  we have  $0 < l_i$ . Otherwise, our robustness analysis fails to certify.

For the neural network in Fig. 5.2, if we want to prove that class 1 is most likely, this means we execute one additional assignment  $x_{13} \leftarrow x_{11} - x_{12}$ . Abstract interpretation derives the bounds  $l_{13} = 1$ ,  $u_{13} = 4$ . The neural network is proved robust, because  $l_{13}$  is positive.

The above discussion showed how to use our abstract transformers to prove robustness. However, a similar procedure could be used to prove standard pre/post conditions (by performing the analysis starting with the pre-condition).

### 5.3.6 Correctness of Abstract Transformers

In this section, we prove that our abstract transformers are sound, and that they preserve the invariant. Formally, for  $T_f^\#(a) = a'$  we have  $T_f(\gamma_{i-1}(a)) \subseteq \gamma_i(a')$  and  $\gamma_i(a') \subseteq \times_{k \in [i]} [l'_k, u'_k]$ .

**SOUNDNESS** We first prove a lemma that is needed to prove soundness of our ReLU transformer.

**Lemma 5.3.1.** For  $l < 0$ ,  $0 < u$ ,  $l \leq x \leq u$ , and  $\lambda \in [0, 1]$  we have  $\lambda \cdot x \leq \max(0, x) \leq u \cdot \frac{x-l}{u-l}$ .

*Proof.* If  $x < 0$ , then  $\lambda \cdot x \leq 0 = \max(0, x)$ . If  $x \geq 0$ , then  $\lambda \cdot x \leq x = \max(0, x)$ . If  $x < 0$ , then  $\max(0, x) = 0 \leq u \cdot \frac{x-l}{u-l}$ . If  $x \geq 0$  then  $\max(0, x) = x \leq u \cdot \frac{x-l}{u-l}$  because  $x \cdot (-l) \leq u \cdot (-l) \Leftrightarrow x \cdot u - x \cdot l \leq x \cdot u - u \cdot l \Leftrightarrow x \cdot (u - l) \leq u \cdot (x - l)$ .  $\square$

**Theorem 5.3.2.** *The ReLU abstract transformer is sound.*

*Proof.* Let  $f: \mathbb{R}^{i-1} \rightarrow \mathbb{R}^i$  execute the assignment  $x_i \leftarrow \max(0, x_j)$  for some  $j < i$ , and let  $a \in \mathcal{A}_{i-1}$  be arbitrary. We have  $\gamma_{i-1}(a) \subseteq \times_{k \in [i-1]} [l_k, u_k]$  and

$$\begin{aligned} T_f(\gamma_{i-1}(a)) &= \{f(x) \mid x \in \gamma_{i-1}(a)\} \\ &= \{(x_1, \dots, x_{i-1}, \max(0, x_j)) \mid (x_1, \dots, x_{i-1}) \in \gamma_{i-1}(a)\} \\ &= \{x \in \mathbb{R}^i \mid (x_1, \dots, x_{i-1}) \in \gamma_{i-1}(a) \wedge x_i = \max(0, x_j)\} \\ &= \{x \in \mathbb{R}^i \mid (\forall k \in [i-1]. a_k^{\leq}(x) \leq x_k \wedge a_k^{\geq}(x) \geq x_k) \wedge x_i = \max(0, x_j)\}. \end{aligned}$$

If  $u_j \leq 0$ , we have that  $(\forall k \in [i-1]. a_k^{\leq}(x) \leq x_k \wedge a_k^{\geq}(x) \geq x_k)$  implies  $x_j \leq 0$ , and

$$\begin{aligned} T_f(\gamma_{i-1}(a)) &= \{x \in \mathbb{R}^i \mid (\forall k \in [i-1]. a_k^{\leq}(x) \leq x_k \wedge a_k^{\geq}(x) \geq x_k) \wedge x_i = \max(0, x_j) \\ &\quad \wedge x_j \leq 0\} \\ &= \{x \in \mathbb{R}^i \mid (\forall k \in [i-1]. a_k^{\leq}(x) \leq x_k \wedge a_k^{\geq}(x) \geq x_k) \wedge x_i = 0\} \\ &= \{x \in \mathbb{R}^i \mid \forall k \in [i]. a_k^{\leq}(x) \leq x_k \wedge a_k^{\geq}(x) \geq x_k\} \\ &= \gamma_i(T_f^\#(a)). \end{aligned}$$

Otherwise, if  $0 \leq l_j$ , we have that  $(\forall k \in [i-1]. a_k^{\leq}(x) \leq x_k \wedge a_k^{\geq}(x) \geq x_k)$  implies  $0 \leq x_j$ , and

$$\begin{aligned} T_f(\gamma_{i-1}(\mathbf{a})) &= \{x \in \mathbb{R}^i \mid (\forall k \in [i-1]. a_k^{\leq}(x) \leq x_k \wedge a_k^{\geq}(x) \geq x_k) \wedge x_i = \max(0, x_j) \\ &\quad \wedge 0 \leq x_j\} \\ &= \{x \in \mathbb{R}^i \mid (\forall k \in [i-1]. a_k^{\leq}(x) \leq x_k \wedge a_k^{\geq}(x) \geq x_k) \wedge x_i = x_j\} \\ &= \{x \in \mathbb{R}^i \mid \forall k \in [i]. a_k^{\prime \leq}(x) \leq x_k \wedge a_k^{\prime \geq}(x) \geq x_k\} \\ &= \gamma_i(T_f^\#(\mathbf{a})). \end{aligned}$$

Otherwise, we have  $l_j < 0$  and  $0 < u_j$  and that  $(\forall k \in [i-1]. a_k^{\leq}(x) \leq x_k \wedge a_k^{\geq}(x) \geq x_k)$  implies  $l_j \leq x_j \leq u_j$  and therefore

$$\begin{aligned} T_f(\gamma_{i-1}(\mathbf{a})) &= \{x \in \mathbb{R}^i \mid (\forall k \in [i-1]. a_k^{\leq}(x) \leq x_k \wedge a_k^{\geq}(x) \geq x_k) \wedge x_i = \max(0, x_j)\} \\ &\subseteq \{x \in \mathbb{R}^i \mid (\forall k \in [i-1]. a_k^{\leq}(x) \leq x_k \wedge a_k^{\geq}(x) \geq x_k) \wedge x_i \leq u_j \cdot \frac{x_j - l_j}{u_j - l_j} \\ &\quad \wedge x_i \geq \lambda \cdot x_j\} \\ &= \{x \in \mathbb{R}^i \mid \forall k \in [i]. a_k^{\prime \leq}(x) \leq x_k \wedge a_k^{\prime \geq}(x) \geq x_k\} \\ &= \gamma_i(T_f^\#(\mathbf{a})). \end{aligned}$$

Therefore, in all cases,  $T_f(\gamma_{i-1}(\mathbf{a})) \subseteq \gamma_i(T_f^\#(\mathbf{a}))$ . Note that we lose precision only in the last case.  $\square$

**Theorem 5.3.3.** *The sigmoid and tanh abstract transformers are sound.*

*Proof.* A function  $g: \mathbb{R} \rightarrow \mathbb{R}$  with  $g'(x) > 0$  and  $0 \leq g''(x) \Leftrightarrow 0 \leq x$  is monotonically increasing, and furthermore,  $g|_{(-\infty, 0]}$  (the restriction to  $(-\infty, 0]$ ) is convex and  $g|_{(0, \infty)}$  is concave.

Let  $f: \mathbb{R}^{i-1} \rightarrow \mathbb{R}^i$  execute the assignment  $x_i \leftarrow g(x_j)$  for some  $j < i$ , and let  $\mathbf{a} \in \mathcal{A}_{i-1}$  be arbitrary. We have

$$\begin{aligned} T_f(\gamma_{i-1}(\mathbf{a})) &= \{f(x) \mid x \in \gamma_{i-1}(\mathbf{a})\} \\ &= \{(x_1, \dots, x_{i-1}, g(x_j)) \mid (x_1, \dots, x_{i-1}) \in \gamma_{i-1}(\mathbf{a})\} \\ &= \{x \in \mathbb{R}^i \mid (\forall k \in [i-1]. a_k^{\leq}(x) \leq x_k \wedge a_k^{\geq}(x) \geq x_k) \wedge x_i = g(x_j)\}. \end{aligned}$$

If  $l_j = u_j$ , then  $(\forall k \in [i-1]. a_k^{\leq}(x) \leq x_k \wedge a_k^{\geq}(x) \geq x_k)$  implies  $x_j = l_j$  and therefore

$$\begin{aligned} T_f(\gamma_{i-1}(\mathbf{a})) &= \{x \in \mathbb{R}^i \mid (\forall k \in [i-1]. a_k^{\leq}(x) \leq x_k \wedge a_k^{\geq}(x) \geq x_k) \wedge x_i = g(x_j)\} \\ &= \{x \in \mathbb{R}^i \mid (\forall k \in [i-1]. a_k^{\leq}(x) \leq x_k \wedge a_k^{\geq}(x) \geq x_k) \wedge x_i = g(l_j)\} \\ &= \{x \in \mathbb{R}^i \mid (\forall k \in [i-1]. a_k^{\leq}(x) \leq x_k \wedge a_k^{\geq}(x) \geq x_k) \wedge g(l_j) \leq x_i \\ &\quad \wedge x_j \leq g(l_j)\} \\ &= \{x \in \mathbb{R}^i \mid \forall k \in [i]. a_k^{\prime \leq}(x) \leq x_k \wedge a_k^{\prime \geq}(x) \geq x_k\} \\ &= \gamma_i(T_f^\#(\mathbf{a})). \end{aligned}$$

Therefore, the transformer is exact in this case.

Otherwise, we need to show that  $(\forall k \in [i-1]. a_k^{\leq}(x) \leq x_k \wedge a_k^{\geq}(x) \geq x_k) \wedge x_i = g(x_j)$  implies  $a_i^{\leq}(x) \leq x_i$  and  $a_i^{\geq}(x) \geq x_i$ . We let  $x \in \mathbb{R}^i$  be arbitrary with  $(\forall k \in [i-1]. a_k^{\leq}(x) \leq x_k \wedge a_k^{\geq}(x) \geq x_k) \wedge x_i = g(x_j)$  and consider  $a_i^{\leq}(x)$  and  $a_i^{\geq}(x)$  separately. Recall that  $\lambda = (g(u_j) - g(l_j))/(u_j - l_j)$  and  $\lambda' = \min(g'(l_j), g'(u_j))$ . If  $0 \leq l_j$ , then, because  $g$  is concave on positive inputs,

$$\begin{aligned} a_i^{\leq}(x) &= g(l_j) + \lambda \cdot (x_j - l_j) = \left(1 - \frac{x_j - l_j}{u_j - l_j}\right) \cdot g(l_j) + \frac{x_j - l_j}{u_j - l_j} \cdot g(u_j) \\ &\leq g\left(\left(1 - \frac{x_j - l_j}{u_j - l_j}\right) \cdot l_j + \frac{x_j - l_j}{u_j - l_j} \cdot u_j\right) = g(x_j) = x_i. \end{aligned}$$

Otherwise, because  $g'$  is non-decreasing on  $(-\infty, 0]$  and decreasing on  $(0, \infty)$ , we have that  $\lambda' = \min(g'(l_j), g'(u_j)) \leq g'(\xi)$  for all  $\xi \in [l_j, u_j]$ . Therefore,

$$a_i^{\leq}(x) = g(l_j) + \lambda' \cdot (x_j - l_j) = g(l_j) + \int_{l_j}^{x_j} \lambda' d\xi \leq g(l_j) + \int_{l_j}^{x_j} g'(\xi) d\xi = g(x_j).$$

The proof of  $a_i^{\geq}(x) \geq x_i$  is analogous.

We conclude

$$\begin{aligned} T_f(\gamma_{i-1}(a)) &= \{x \in \mathbb{R}^i \mid (\forall k \in [i-1]. a_k^{\leq}(x) \leq x_k \wedge a_k^{\geq}(x) \geq x_k) \wedge x_i = g(x_j)\} \\ &\subseteq \{x \in \mathbb{R}^i \mid (\forall k \in [i-1]. a_k^{\leq}(x) \leq x_k \wedge a_k^{\geq}(x) \geq x_k) \wedge a_i^{\leq}(x) \leq x_i \\ &\quad \wedge a_i^{\geq}(x) \geq x_i\} \\ &= \{x \in \mathbb{R}^i \mid (\forall k \in [i]. a_k^{\leq}(x) \leq x_k \wedge a_k^{\geq}(x) \geq x_k)\} \\ &= \gamma_i(T_f^\#(a)). \end{aligned}$$

where the inclusion is strict because we have dropped the constraint  $x_i = g(x_j)$ . Therefore, the abstract transformer is sound.  $\square$

**Theorem 5.3.4.** *The maxpool abstract transformer is sound.*

*Proof.* Let  $f: \mathbb{R}^{i-1} \rightarrow \mathbb{R}^i$  execute the assignment  $x_i \leftarrow \max_{j \in J}(0, x_j)$  for some  $J \subseteq [i-1]$ , and let  $a \in \mathcal{A}_{i-1}$  be arbitrary. We have

$$\begin{aligned} T_f(\gamma_{i-1}(a)) &= \{f(x) \mid x \in \gamma_{i-1}(a)\} \\ &= \{(x_1, \dots, x_{i-1}, \max_{j \in J} x_j) \mid (x_1, \dots, x_{i-1}) \in \gamma_{i-1}(a)\} \\ &= \{x \in \mathbb{R}^i \mid (\forall k \in [i-1]. a_k^{\leq}(x) \leq x_k \wedge a_k^{\geq}(x) \geq x_k) \wedge x_i = \max_{j \in J} x_j\}. \end{aligned}$$

There are two cases. If there is some  $k \in J$  with  $u_j < l_k$  for all  $j \in J \setminus \{k\}$ , then  $(\forall k \in [i-1]. a_k^{\leq}(x) \leq x_k \wedge a_k^{\geq}(x) \geq x_k)$  implies that  $\max_{j \in J} x_j = x_k$  and therefore

$$\begin{aligned} T_f(\gamma_{i-1}(a)) &= \{x \in \mathbb{R}^i \mid (\forall k \in [i-1]. a_k^{\leq}(x) \leq x_k \wedge a_k^{\geq}(x) \geq x_k) \wedge x_i = \max_{j \in J} x_j\} \\ &= \{x \in \mathbb{R}^i \mid (\forall k \in [i-1]. a_k^{\leq}(x) \leq x_k \wedge a_k^{\geq}(x) \geq x_k) \wedge x_i = x_k\} \\ &= \{x \in \mathbb{R}^i \mid \forall k \in [i]. a_k^{\prime \leq}(x) \leq x_k \wedge a_k^{\prime \geq}(x) \geq x_k\} \\ &= \gamma_i(T_f^\#(a)). \end{aligned}$$

Otherwise, the transformer chooses a  $k$  with maximal  $l_k$ . We also know that  $(\forall k \in [i-1]. a_k^{\leq}(x) \leq x_k \wedge a_k^{\geq}(x) \geq x_k)$  implies  $x_j \leq u_j$  for all  $j \in J$ , and therefore

$$\begin{aligned} T_f(\gamma_{i-1}(a)) &= \{x \in \mathbb{R}^i \mid (\forall k \in [i-1]. a_k^{\leq}(x) \leq x_k \wedge a_k^{\geq}(x) \geq x_k) \wedge x_i = \max_{j \in J} x_j\} \\ &\subseteq \{x \in \mathbb{R}^i \mid (\forall k \in [i-1]. a_k^{\leq}(x) \leq x_k \wedge a_k^{\geq}(x) \geq x_k) \wedge x_k \leq x_i \\ &\quad \wedge \max_{j \in J} u_j \geq x_i\} \\ &= \{x \in \mathbb{R}^i \mid \forall k \in [i]. a_k^{\prime \leq}(x) \leq x_k \wedge a_k^{\prime \geq}(x) \geq x_k\} \\ &= \gamma_i(T_f^\#(a)). \end{aligned}$$

In summary, in both cases,  $T_f(\gamma_{i-1}(a)) \subseteq \gamma_i(T_f^\#(a))$ . □

**Theorem 5.3.5.** *The affine abstract transformer is sound and exact.*

*Proof.* Let  $f: \mathbb{R}^{i-1} \rightarrow \mathbb{R}^i$  execute the assignment  $x_i \leftarrow v + \sum_{j \in [i-1]} w_j \cdot x_j$  for some  $v \in \mathbb{R}, w \in \mathbb{R}^{i-1}$ , and let  $a \in \mathcal{A}_{i-1}$  be arbitrary. We have

$$\begin{aligned} T_f(\gamma_{i-1}(a)) &= \{f(x) \mid x \in \gamma_{i-1}(a)\} \\ &= \{(x_1, \dots, x_{i-1}, v + \sum_{j \in [i-1]} w_j \cdot x_j) \mid (x_1, \dots, x_{i-1}) \in \gamma_{i-1}(a)\} \\ &= \{x \in \mathbb{R}^i \mid (x_1, \dots, x_{i-1}) \in \gamma_{i-1}(a) \wedge x_i = v + \sum_{j \in [i-1]} w_j \cdot x_j\} \\ &= \{x \in \mathbb{R}^i \mid (\forall k \in [i-1]. a_k^{\leq}(x) \leq x_k \wedge a_k^{\geq}(x) \geq x_k) \\ &\quad \wedge x_i = v + \sum_{j \in [i-1]} w_j \cdot x_j\} \\ &= \{x \in \mathbb{R}^i \mid \forall k \in [i]. a_k^{\prime \leq}(x) \leq x_k \wedge a_k^{\prime \geq}(x) \geq x_k\} \\ &= \gamma_i(T_f^\#(a)). \end{aligned}$$

Thus,  $T_f(\gamma_{i-1}(a)) = \gamma_i(T_f^\#(a))$ . □

**INVARIANT** We now prove that our abstract transformers preserve the invariant. For each of our abstract transformers  $T_f^\#$ , we have to show that for  $T_f^\#(a) = a'$ , we have  $\gamma_i(a') \subseteq \times_{j \in [i]} [l'_j, u'_j]$ . Note that the constraints  $(\forall k \in [i]. a_k^{\prime \leq}(x) \leq x_k \wedge a_k^{\prime \geq}(x) \geq x_k)$  include all constraints of  $a$ . We first assume that the invariant holds for  $a$ ; thus,  $(\forall k \in [i-1]. a_k^{\leq}(x) \leq x_k \wedge a_k^{\geq}(x) \geq x_k)$

implies the bounds  $(\forall k \in [i-1]. l_k \leq x_k \leq u_k)$ , which are equivalent to  $(\forall k \in [i-1]. l'_k \leq x_k \leq u'_k)$ , because our abstract transformers preserve the bounds of existing variables. It therefore suffices to show that  $(\forall k \in [i]. a'_k \leq(x) \leq x_k \wedge a'_k \geq(x) \geq x_k)$  implies  $l'_i \leq x_i \leq u'_i$ .

**Theorem 5.3.6.** *The ReLU abstract transformer preserves the invariant.*

*Proof.* If  $u_j \leq 0$ , we have  $a'_i \leq(x) = a'_i \geq(x) = 0$  and therefore  $(\forall k \in [i]. a'_k \leq(x) \leq x_k \wedge a'_k \geq(x) \geq x_k)$  implies  $0 = l'_i = a'_i \leq(x) \leq x_i \leq a'_i \geq(x) = u'_i = 0$ . If  $0 \leq l_j$ , we have  $a'_i \leq(x) = a'_i \geq(x) = x_j$  and therefore  $(\forall k \in [i]. a'_k \leq(x) \leq x_k \wedge a'_k \geq(x) \geq x_k)$  implies  $l'_i = l_j \leq x_j = x_i \leq u_j = u'_i$ . Otherwise, we have  $l_j < 0$  and  $0 < u_j$ , as well as  $a'^{\leq}(x)_i = \lambda \cdot x_j$ ,  $a'^{\geq}(x)_i = u_j \cdot \frac{x_j - l_j}{u_j - l_j}$ , and so  $(\forall k \in [i]. a'_k \leq(x) \leq x_k \wedge a'_k \geq(x) \geq x_k)$  implies  $l'_i = \lambda \cdot l_j \leq x_i \leq u_j = u'_i$ .  $\square$

**Theorem 5.3.7.** *The sigmoid and tanh abstract transformers preserve the invariant.*

*Proof.* The constraints  $(\forall k \in [i]. a'_k \leq(x) \leq x_k \wedge a'_k \geq(x) \geq x_k)$  imply  $l_j \leq x_j \leq u_j$  and by monotonicity of  $g$ , we obtain  $l'_i = g(l_j) \leq x_i \leq g(u_j) = u'_i$  using  $x_i = g(x_j)$ .  $\square$

**Theorem 5.3.8.** *The maxpool abstract transformer preserves the invariant.*

*Proof.* The maxpool transformer either sets  $a'_i \leq(x) = a'_i \geq(x) = x_k$  and  $l'_i = l_k$  and  $u'_i = u_k$ , in which case  $(\forall k \in [i]. a'_k \leq(x) \leq x_k \wedge a'_k \geq(x) \geq x_k)$  implies  $l'_i = l_k \leq x_k = x_i \leq u_k = u'_i$ , or it sets  $a'_i \leq(x) = x_k$ ,  $l'_i = l_k$  and  $u'_i = a'_i \geq(x)$ , such that  $(\forall k \in [i]. a'_k \leq(x) \leq x_k \wedge a'_k \geq(x) \geq x_k)$ , which implies  $l'_i \leq x_i \leq u'_i$ .  $\square$

**Theorem 5.3.9.** *The affine abstract transformer preserves the invariant.*

*Proof.* Note that  $s'$  and  $t'$  are finite, because in each step, the maximal index of a variable whose coefficient in, respectively,  $b_s$  and  $c_t$  is nonzero decreases by at least one. Assume  $\forall k \in [i]. a'_k \leq(x) \leq x_k \wedge a'_k \geq(x) \geq x_k$ . We have to show that  $b_{s'}(x) \leq x_i$  and  $c_{t'}(x) \geq x_i$ . It suffices to show that  $\forall s \in [s'] . b_s(x) \leq x_i$  and  $\forall t \in [t'] . c_t(x) \geq x_i$ .

To show  $\forall s \in [s'] . b_s(x) \leq x_i$ , we use induction on  $s$ . We have  $b_1(x) = a'_i \leq(x) \leq x_i$ . Assuming  $b_s(x) \leq x_i$  and  $b_s(x) = v' + \sum_{j \in [k]} w'_j \cdot x_j$  for some  $k \in [i-1], v' \in \mathbb{R}, w' \in \mathbb{R}^k$ , we have

$$\begin{aligned} x_i \geq b_s(x) &= v' + \sum_{j \in [k]} w'_j \cdot x_j \\ &= v' + \sum_{j \in [k]} \underbrace{(\max(0, w'_j))}_{\geq 0} \cdot x_j + \underbrace{(\min(w'_j, 0))}_{\leq 0} \cdot x_j \\ &\geq v' + \sum_{j \in [k]} (\max(0, w'_j) \cdot a'_j \leq(x) + \min(w'_j, 0) \cdot a'_j \geq(x)) \\ &= b_{s+1}(x). \end{aligned}$$

To show  $\forall t \in [t'] . c_t(x) \geq x_i$ , we use induction on  $t$ . We have  $c_1(x) = a_i^{\geq}(x) \geq x_i$ . Assuming  $c_t(x) \geq x_i$  and  $c_t(x) = v' + \sum_{j \in [k]} w'_j \cdot x_j$  for some  $k \in [i-1], v' \in \mathbb{R}, w' \in \mathbb{R}^k$ , we have

$$\begin{aligned} x_i \leq c_t(x) &= v' + \sum_{j \in [k]} w'_j \cdot x_j \\ &= v' + \sum_{j \in [k]} \underbrace{(\max(0, w'_j) \cdot x_j)}_{\geq 0} + \underbrace{(\min(w'_j, 0) \cdot x_j)}_{\leq 0} \\ &\leq v' + \sum_{j \in [k]} (\max(0, w'_j) \cdot a_j^{\geq}(x) + \min(w'_j, 0) \cdot a_j^{\leq}(x)) \\ &= c_{t+1}(x). \end{aligned}$$

Therefore,  $(\forall k \in [i]. a_k^{\leq}(x) \leq x_k \wedge a_k^{\geq}(x) \geq x_k)$  implies  $l'_i \leq x_i \leq u'_i$ .  $\square$

### 5.3.7 Soundness under Floating-Point Arithmetic

Our abstract domain and its transformers above are sound under real arithmetic but unsound under floating-point arithmetic if one does not take care of the rounding errors. To obtain soundness, let  $\mathbb{F}$  be the set of floating-point values and  $\oplus_f, \ominus_f, \otimes_f, \oslash_f$  be the floating-point interval addition, subtraction, multiplication, and division, respectively, as defined in [141] with lower bounds rounded towards  $-\infty$  and upper bounds rounded towards  $+\infty$ . For a real constant  $c$ , we use  $c^-, c^+ \in \mathbb{F}$  to denote the floating-point representation of  $c$  with rounding towards  $-\infty$  and  $+\infty$  respectively. We use the standard interval linear form, where the coefficients in the constraints are intervals instead of scalars, to define an abstract element  $a \in \mathcal{A}_n$  over  $n$  variables in our domain as a tuple  $a = \langle a^{\leq}, a^{\geq}, l, u \rangle$  where for  $i \in [n]$ :

$$\begin{aligned} a_i^{\leq}, a_i^{\geq} &\in \{x \mapsto [v^-, v^+] \oplus_f \sum_{j \in [i-1]} [w_j^-, w_j^+] \otimes_f x_j \mid v^-, v^+ \in \mathbb{F} \cup \{-\infty, +\infty\}, \\ &w^-, w^+ \in \mathbb{F}^{i-1}\} \end{aligned}$$

and  $l, u \in (\mathbb{F} \cup \{-\infty, +\infty\})^n$ . For a floating-point interval  $[l_i, u_i]$ , let  $\text{inf}$  and  $\text{sup}$  be functions that return its lower and upper bound. The concretization function  $\gamma_n: \mathcal{A}_n \rightarrow \mathcal{P}(\mathbb{F}^n)$  is given by

$$\gamma_n(a) = \{x \in \mathbb{F}^n \mid \forall i \in [n]. \text{inf}(a_i^{\leq}(x)) \leq x_i \wedge x_i \leq \text{sup}(a_i^{\geq}(x))\}.$$

We next modify our abstract transformers for soundness under floating-point arithmetic. It is straightforward to modify the maxpool transformer so we only show our modifications for the ReLU, sigmoid, tanh, and affine abstract transformers assigning to the variable  $x_i$ .

**RELU ABSTRACT TRANSFORMER** It is straightforward to handle the cases  $l_j \geq 0$  or  $u_j \leq 0$ . For the remaining case, we add the following constraints:

$$\begin{aligned} [\lambda, \lambda] \otimes_f x_j &\leq [1, 1] \otimes_f x_i, \\ [1, 1] \otimes_f x_i &\leq [\psi^-, \psi^+] \otimes_f x_j \oplus_f [\mu^-, \mu^+]. \end{aligned}$$

where  $\lambda \in \{0, 1\}$  and  $[\psi^-, \psi^+] = [u_j^-, u_j^+] \otimes_f ([u_j^-, u_j^+] \ominus_f [l_j^-, l_j^+])$ ,  $[\mu^-, \mu^+] = ([-l_j^+, -l_j^-] \otimes_f [u_j^-, u_j^+]) \otimes_f ([u_j^-, u_j^+] \ominus_f [l_j^-, l_j^+])$ . Finally, we set  $l_i = \lambda \cdot l_j^-$  and  $u_i = u_j^+$ .

**SIGMOID AND TANH ABSTRACT TRANSFORMERS** We consider the case when  $l_j < 0$ . We soundly compute an interval for the possible values of  $\lambda$  under any rounding mode as  $[\lambda^-, \lambda^+] = ([g(u_j)^-, g(u_j)^+] \ominus_f [g(l_j)^-, g(l_j)^+]) \otimes_f ([u_j^-, u_j^+] \ominus_f [l_j^-, l_j^+])$ . Similarly, both  $g'(l_j)$  and  $g'(u_j)$  are soundly abstracted by the intervals  $[g'(l_j)^-, g'(l_j)^+]$  and  $[g'(u_j)^-, g'(u_j)^+]$ , respectively. Because of the limitations of the floating-point format, it can happen that the upper polyhedral constraint with slope  $\lambda$  passing through  $l_j$  intersects the curve at a point  $< u_j$ . This happens frequently for smaller perturbations. To ensure soundness, we detect such cases and return the box  $[g(l_j)^-, g(u_j)^+]$ . Other computations for the transformers can be handled similarly.

**AFFINE ABSTRACT TRANSFORMER** The affine abstract transformer  $x_i \leftarrow v + \sum_{j \in [i-1]} w_j \cdot x_j$  for some  $w \in \mathbb{F}^{i-1}$  first adds the interval linear constraints  $\alpha_i^{\leq}(x) = \alpha_i^{\geq}(x) = [v^-, v^+] \oplus_f \sum_{j \in [i-1]} [w_j^-, w_j^+] \otimes_f x_j$ .

We modify the backsubstitution for the computation of  $l_i$  and  $u_i$ . Formally, if we want to obtain  $l'_i$ , we start with  $b_1(x) = \alpha_i^{\leq}(x)$ . If we have  $b_s(x) = [v'^-, v'^+] \oplus_f \sum_{j \in [k]} [w_j'^-, w_j'^+] \otimes_f x_j$  for some  $k \in [i-1]$ ,  $v'^-, v'^+, w_j'^-, w_j'^+ \in \mathbb{F}$ , then

$$b_{s+1}(x) = [v'^-, v'^+] \oplus_f \sum_{j \in [k]} \begin{cases} [w_j'^-, w_j'^+] \otimes_f \alpha_j^{\leq}(x), & \text{if } w_j'^- \geq 0, \\ [w_j'^-, w_j'^+] \otimes_f \alpha_j^{\geq}(x), & \text{if } w_j'^+ \leq 0, \\ [\theta_l^-, \theta_l^+], & \text{otherwise.} \end{cases}$$

Note that  $\oplus_f$  and  $\otimes_f$  as defined in [141] add extra error terms that are not shown above for simplicity so that our results contain all values that can arise by executing the different additions and multiplications in different orders. Here,  $[\theta_l^-, \theta_l^+] \in \mathbb{F}$  are the floating-point values of the lower bound of the interval  $[w_j'^-, w_j'^+] \otimes_f [l_j, u_j]$  rounded towards  $-\infty$  and  $+\infty$  respectively. We iterate until we reach  $b_{s'}$  with  $b_{s'}(x) = [v''^-, v''^+]$ , i.e.,  $s'$  is the smallest number with this property. We then set  $l'_i = v''^-$ . We compute  $u'_i$  analogously.



---

**Algorithm 5.1** Rotate image  $I$  by  $\theta$  degrees.

---

**procedure** ROTATE( $I, \theta$ )

**Input:**  $I \in [0, 1]^{m \times n}$ ,  $\theta \in [-\pi, \pi]$ , **Output:**  $R \in [0, 1]^{m \times n}$

**for**  $i \in \{1, \dots, m\}, j \in \{1, \dots, n\}$  **do**

$(x, y) \leftarrow (j - (n + 1)/2, (m + 1)/2 - i)$

$(x', y') \leftarrow (\cos(-\theta) \cdot x - \sin(-\theta) \cdot y, \sin(-\theta) \cdot x + \cos(-\theta) \cdot y)$

$(i'_{\text{low}}, i'_{\text{high}}) \leftarrow (\max(1, \lfloor (m + 1)/2 - y' \rfloor), \min(m, \lceil (m + 1)/2 - y' \rceil))$

$(j'_{\text{low}}, j'_{\text{high}}) \leftarrow (\max(1, \lfloor x' + (n + 1)/2 \rfloor), \min(n, \lceil x' + (n + 1)/2 \rceil))$

$t \leftarrow \sum_{i'=i'_{\text{low}}}^{i'_{\text{high}}} \sum_{j'=j'_{\text{low}}}^{j'_{\text{high}}} \max(0, 1 - \sqrt{(j' - x')^2 + (i' - y')^2})$

**if**  $t \neq 0$  **then**

$R_{i,j} \leftarrow (1/t) \cdot \sum_{i'=i'_{\text{low}}}^{i'_{\text{high}}} \sum_{j'=j'_{\text{low}}}^{j'_{\text{high}}} \max(0, 1 - \sqrt{(j' - x')^2 + (i' - y')^2}) \cdot I_{i',j'}$

**else**

$R_{i,j} \leftarrow 0$

**end if**

**end for**

**return**  $R$

**end procedure**

---

## 5.4 REFINEMENT OF ANALYSIS RESULTS

In this section, we show how to apply a form of abstraction refinement based on trace partitioning [169] to certify robustness for more complex adversarial regions, which cannot be exactly represented using a set of interval constraints. In particular, we will show how to handle adversarial regions that, in addition to permitting small perturbations to each pixel, allow the adversary to rotate the input image by an angle  $\theta \in [\alpha, \beta]$  within an interval.

**CERTIFYING ROBUSTNESS AGAINST IMAGE ROTATIONS** Consider Algorithm 5.1, which rotates an  $m \times n$ -pixel (grayscale) image by an angle  $\theta$ . To compute the intensity  $R_{i,j}$  of a given output pixel, it first computes the (real-valued) position  $(x', y')$  that would be mapped to the position of the center of the pixel. Then, it performs linear interpolation: it forms a convex combination of pixels in the neighborhood of  $(x', y')$ , such that the contribution of each pixel is proportional to the distance to  $(x', y')$ , cutting off contributions at distance 1.

Our goal is to certify that a neural network  $N: \mathbb{R}^{m \times n} \rightarrow \mathbb{R}^r$  classifies all images obtained by rotating an input image using Algorithm 5.1 with an angle  $\theta \in [\alpha, \beta] \subseteq [-\pi, \pi]$  in the same way. More generally, if we have an adversarial region  $X \subseteq \mathbb{R}^{m \times n}$  (represented using componentwise interval constraints), we would like to certify that for any image  $I \in X$  and any angle  $\theta \in [\alpha, \beta]$ , the neural network  $N$  classifies  $\text{ROTATE}(I, \theta)$  to a given class  $k$ . This induces a new adversarial region  $X' = \{\text{ROTATE}(I, \theta) \mid I \in X, \theta \in [\alpha, \beta]\}$ . Note that because we deal with regions (and not only concrete images) as well as rotations that employ linear interpolation,

we cannot simply enumerate all possible rotations as done for simpler rotation algorithms and concrete images [159].

**INTERVAL SPECIFICATION OF  $X'$**  We certify robustness against rotations by deriving lower and upper bounds on the intensities of all pixels of the rotated image. We then certify that the neural network classifies all images satisfying those bounds to class  $k$ . To obtain bounds, we apply abstract interpretation to Algorithm 5.1, using the interval domain (more powerful numerical domains could be applied). We use standard interval domain transformers, except to derive bounds on  $t$  and  $R_{i,j}$ , which we compute (at the same time), by enumerating all possible integer values of  $i'_{\text{low}}$ ,  $i'_{\text{high}}$ ,  $j'_{\text{low}}$  and  $j'_{\text{high}}$  (respecting the constraints  $i'_{\text{low}} + 1 \geq i'_{\text{high}}$  and  $j'_{\text{low}} + 1 \geq j'_{\text{high}}$ , and refining the intervals for  $x'$  and  $y'$  based on the known values  $i'_{\text{low}}$  and  $j'_{\text{low}}$ ) and joining the intervals resulting from each case. For each case, we compute intervals for  $R_{i,j}$  in two ways: once using interval arithmetic, restricting partial sums to the interval  $[0, 1]$ , and once by observing that a convex combination of pixel values will be contained in the union of intervals for the individual values. We intersect the intervals resulting from both approaches.

**REFINEMENT OF ABSTRACT INPUTS BY TRACE PARTITIONING** For large enough intervals  $[\alpha, \beta]$ , the derived bounds often become too imprecise. Thus, when our analyzer is invoked with these bounds, it may fail to certify the property, even though it actually holds. We can make the following simple observation: if we have  $n$  sets  $X'_1, \dots, X'_n$  that cover the adversarial region  $X'$ , i.e.  $X' \subseteq \bigcup_{i=1}^n X'_i$ , then it suffices to certify that the neural network  $N$  classifies all input images to class  $k$  for each individual input region  $X'_i$  for  $i \in \{1, \dots, n\}$ . We obtain  $X'_1, \dots, X'_n$  by subdividing the interval  $[\alpha, \beta]$  into  $n$  equal parts:  $\{\text{ROTATE}(I, \theta) \mid I \in X, \theta \in [(i-1)/n \cdot (\beta - \alpha) + \alpha, i/n \cdot (\beta - \alpha) + \alpha]\} \subseteq X'_i$ . Note that each  $X'_i$  is obtained by running the interval analysis on the rotation code with the given angle interval and the adversarial region  $X$ . After obtaining all  $X'_i$ 's, we run our neural network analyzer separately with each  $X'_i$  as input.

**BATCHING** As interval analysis tends to be imprecise for large input intervals, we usually need to subdivide the interval  $[\alpha, \beta]$  into many parts to obtain precise enough output intervals from the interval analysis (a form of trace partitioning [169]). Running our neural network analysis for each of these can be too expensive. Instead, we use a separate refinement step to obtain more precise interval bounds for larger input intervals. We further subdivide each of the  $n$  intervals into  $m$  parts each, for a total of  $n \cdot m$  intervals in  $n$  batches. For each of the  $n$  batches, we then run interval analysis  $m$  times, once for each part, and combine the results using a join, i.e., we compute the smallest common bounding box of all output regions in a batch. The additional refinement within each batch preserves dependencies

between variables that a plain interval analysis would ignore, and thus yields more precise boxes  $X'_1, \dots, X'_n$ , on which we run the neural network analysis.

Using the approach outlined above, we were able to certify, for the first time, that the neural network is robust to non-trivial rotations of all images inside an adversarial region. Interval-based regions for geometric transformations were also derived in the work of [149]. We note that in our follow-up work [15], we obtain tighter polyhedral regions based on a combination of sampling and Lipschitz optimization. We also handle other geometric transformations such as translation, scaling, shearing as well as any arbitrary composition of these. The robustness of the network is then analyzed using DeepPoly for the obtained polyhedral region. Our results using the polyhedral region are more precise than with the interval region presented here. Further, the approach of [15] when combined the GPU implementation GPUPoly [152] of DeepPoly enables the analysis of a 18-layer neural network containing more than 0.5M neurons within 30 minutes.

## 5.5 EXPERIMENTAL EVALUATION

In this section, we evaluate the effectiveness of our approach for certifying the robustness of a large, challenging, and diverse set of neural networks for adversarial regions generated by both changes in pixel intensity as well as image rotations. We implemented our method in the ERAN analyzer [3]. ERAN is written in Python and the abstract transformers of DeepPoly domain are implemented on top of the ELINA library [1, 190] for numerical abstractions. We have implemented both a sequential and a parallel version of our transformers. All code, networks, datasets, and results used in our evaluation are available at <https://github.com/eth-sri/eran>. We compared the precision and performance of DeepPoly against the three state-of-the-art systems that can scale to larger networks:

- AI<sup>2</sup> by [78] uses the Zonotope abstract domain [81] implemented in ELINA for performing abstract interpretation of fully-connected and convolutional ReLU networks. Their transformers are generic and based on standard numerical domains used for program analysis. Therefore they do not exploit the structure of ReLU. As a result, AI<sup>2</sup> is often slow and imprecise.
- Fast-Lin by [211] performs layerwise linear approximations tailored to exploit the structure of fully-connected ReLU networks. We note that Fast-Lin is not sound under floating-point arithmetic and does not support convolutional networks. Nonetheless, we still compare to it despite the fact it may contain false negatives [107] (adapting their method to be sound in floating-point arithmetic is non-trivial).
- DeepZ by [191] provides specialized Zonotope transformers for handling ReLU, sigmoid, and tanh activations, and supports both fully-connected and

convolutional networks. It is worth mentioning that although Fast-Lin and DeepZ employ very different techniques for robustness analysis, both can be shown to have the same precision on fully-connected neural networks with ReLU activations. On our benchmarks, DeepZ was often faster than Fast-Lin.

Our experimental results indicate that DeepPoly is always more precise and faster than all three competing tools on our benchmarks. This demonstrates the suitability of DeepPoly for the task of robustness certification of larger neural networks.

### 5.5.1 Experimental setup

All of our experiments for the feedforward networks were run on a 3.3 GHz 10 core Intel i9-7900X Skylake CPU with a main memory of 64 GB; our experiments for the convolutional networks were run on a 2.6 GHz 14 core Intel Xeon CPU E5-2690 with 512 GB of main memory. We next describe our experimental setup including the datasets, neural networks, and adversarial regions.

**EVALUATION DATASETS.** We used the popular MNIST [124] and CIFAR10 [118] image datasets for our experiments. MNIST contains grayscale images of size  $28 \times 28$  pixels and CIFAR10 consists of RGB images of size  $32 \times 32$  pixels. For our evaluation, we chose the first 100 images from the test set of each dataset. For the task of robustness certification, out of these 100 images, we considered only those that were correctly classified by the neural network.

**NEURAL NETWORKS.** Table 5.1 shows the MNIST and the CIFAR10 neural network architectures used in our experiments. The architectures considered in our evaluation contain up to 88K hidden units. We use networks trained with adversarial training, i.e., defended against adversarial attacks, as well as undefended networks. We used DiffAI by [144] and projected gradient descent (PGD) from [64] for adversarial training. In our evaluation, when we consider the certified robustness of the defended and undefended networks with the same architecture together, we append the suffix *Point* to the name of a neural network trained without adversarial training and the name of the training procedure (either DiffAI or PGD) to the name of a defended network. In the table, the FFNSigmoid and FFNTanh networks use sigmoid and tanh activations, respectively. All other networks use ReLU activations. The FFNSmall and FFNMed network architectures for both MNIST and CIFAR10 datasets were taken from [78] whereas the FFNBig architectures were taken from [211]. The ConvSmall, ConvBig, and ConvSuper architectures were taken from [144].

**ADVERSARIAL REGIONS** We consider the following adversarial regions:

1.  $L_\infty$ -norm [40]: This region is parameterized by a constant  $\epsilon$  and contains all perturbed images  $\bar{x}'$  where each pixel  $\bar{x}'_i$  has a distance of at most  $\epsilon$  from the

Table 5.1: Neural network architectures used in our experiments.

Dataset	Model	Type	#Hidden units	#Hidden layers
MNIST	FFNSmall	fully-connected	510	6
	FFNMed	fully-connected	1610	9
	FFNBig	fully-connected	3072	4
	FFNSigmoid	fully-connected	3000	6
	FFNTanh	fully-connected	3000	6
	ConvSmall	convolutional	3604	3
	ConvBig	convolutional	48064	6
	ConvSuper	convolutional	88544	6
CIFAR <sub>10</sub>	FFNSmall	fully-connected	610	6
	FFNMed	fully-connected	1810	9
	FFNBig	fully-connected	6144	7
	ConvSmall	convolutional	4852	3
	ConvBig	convolutional	62464	6

corresponding pixel  $\bar{x}_i$  in the original input  $\bar{x}$ . We use different values of  $\epsilon$  in our experiments. In general, we use smaller  $\epsilon$  values for the CIFAR<sub>10</sub> dataset compared to the MNIST dataset since the CIFAR<sub>10</sub> networks are known to be less robust against  $L_\infty$ -norm based adversarial regions with larger  $\epsilon$  values [211].

2. Rotation: The input image is first perturbed using a perturbation bounded by  $\epsilon$  in the  $L_\infty$ -norm. All resulting images are then rotated by Algorithm 5.1 using an arbitrary  $\theta \in [\alpha, \beta]$ . The region  $\mathcal{R}_{\bar{x}, \epsilon, [\alpha, \beta]}$  contains all images that can be obtained in this way.

### 5.5.2 $L_\infty$ -Norm Perturbation

We first compare the precision and performance of DeepPoly vs AI<sup>2</sup>, Fast-Lin, and DeepZ for robustness certification against  $L_\infty$ -norm based adversarial attacks on the MNIST FFNSmall network. We note that it is straightforward to parallelize Fast-Lin, DeepZ, and DeepPoly. However, the abstract transformers in AI<sup>2</sup> cannot be efficiently parallelized. To ensure fairness, we ran all four analyzers in single threaded mode. Fig. 5.5 compares the percentage of certified adversarial regions and the average runtime in seconds per  $\epsilon$ -value of all four analyzers. We used six different values for  $\epsilon$  shown on the x-axis. For all analyzers, the number of certified regions decreases with increasing values of  $\epsilon$ . As can be seen, DeepPoly is the fastest and the most precise analyzer on the FFNSmall network. DeepZ has

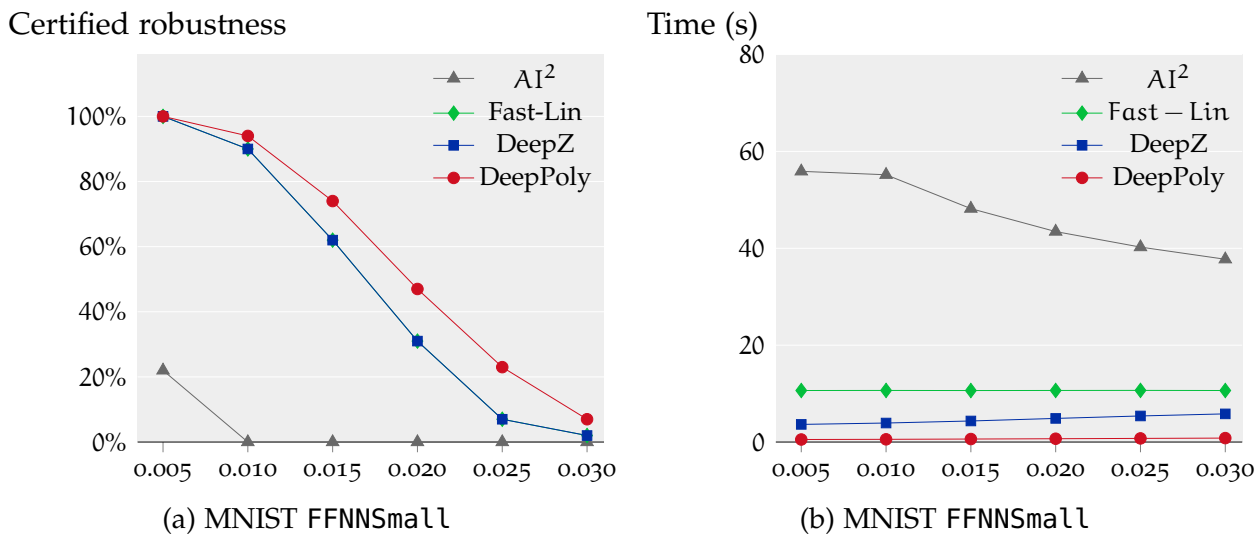


Figure 5.5: Certified robustness and average runtime for  $L_\infty$ -norm perturbations by DeepPoly against AI<sup>2</sup>, Fast-Lin, and DeepZ on the MNIST FFNNSmall. DeepZ and Fast-Lin are equivalent in robustness.

the exact same precision as Fast-Lin but is up to 2.5x faster. AI<sup>2</sup> has significantly worse precision and higher runtime than all other analyzers.

Based on our results in Fig. 5.5, we compare the precision and performance of the parallelized versions of DeepPoly and DeepZ for all of our remaining experiments.

**MNIST FULLY-CONNECTED NETWORKS** Fig. 5.6 considers the MNIST FFNNMed and FFNNBig networks and compares the percentage of adversarial regions on which the neural networks are certified to be robust and the average runtime, per  $\epsilon$  value, for both DeepPoly and DeepZ. Both networks were trained without adversarial training. DeepPoly certifies more than DeepZ on both networks. As an example, considering  $\epsilon = 0.01$ , we notice that DeepPoly certifies 69% of the regions on the FFNNMed network, whereas DeepZ certifies only 46%. The corresponding numbers on the FFNNBig network are 79% and 58% respectively. DeepPoly is also significantly faster than DeepZ and achieves a speedup of up to 4x and 2.5x on the FFNNMed and FFNNBig networks, respectively.

We compare the average percentage of ReLU inputs that can take both positive and negative values per  $\epsilon$ -value for the MNIST FFNNSmall and FFNNMed neural networks in Fig. 5.7. Since the ReLU transformer in both DeepPoly and DeepZ is inexact for such inputs, it is important to reduce their percentage. For both networks, DeepPoly produces strictly less inputs for which the ReLU transformer is inexact than DeepZ.

In Fig. 5.8, we compare the precision of DeepPoly and DeepZ on the MNIST FFNNsigmoid and FFNNtanh networks. Both networks were trained using PGD-based adversarial training. On both networks, DeepPoly is strictly more precise than DeepZ. For the FFNNsigmoid network, there is a sharp decline in the number of

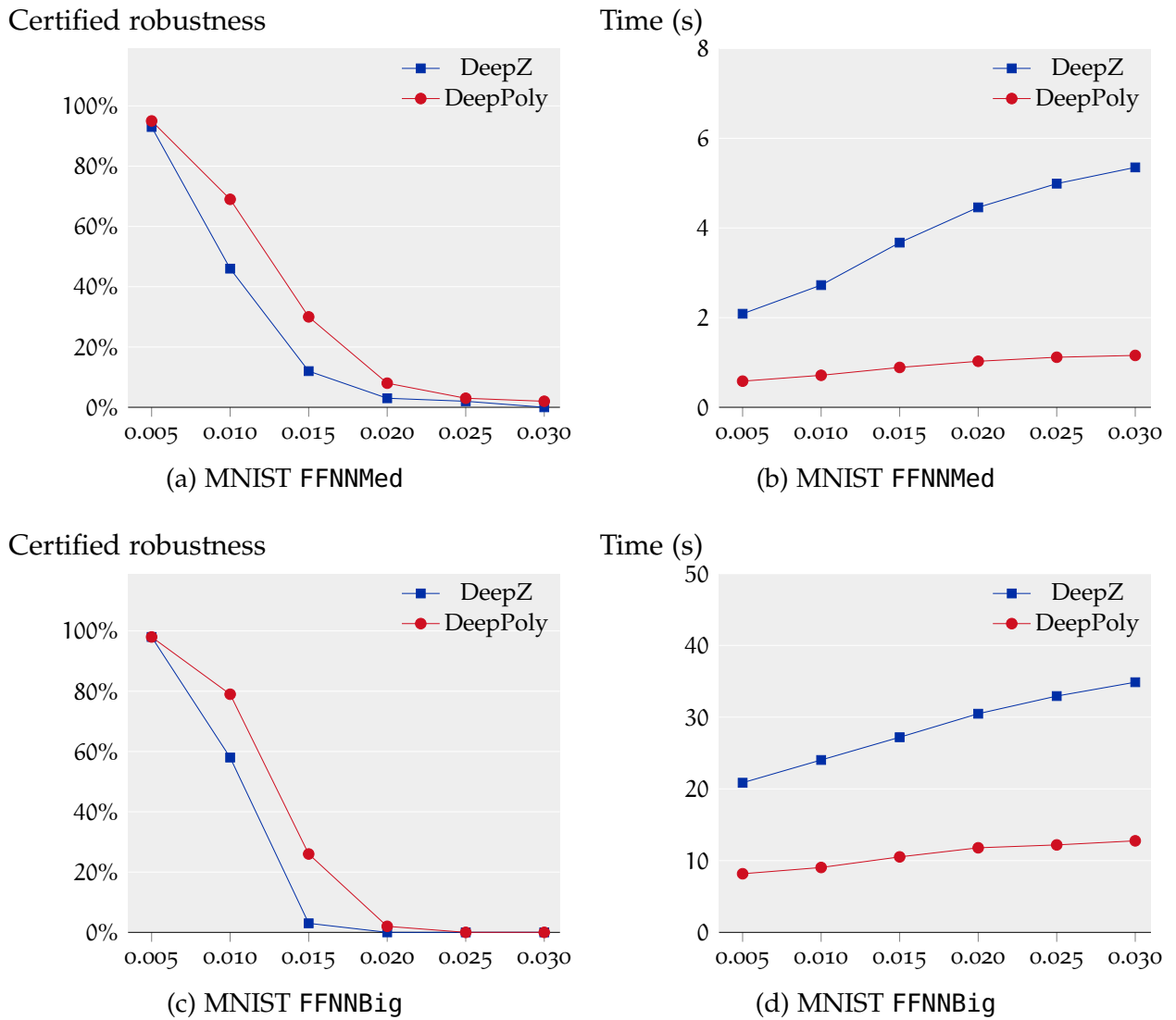


Figure 5.6: Certified robustness and average runtime for  $L_\infty$ -norm perturbations by DeepPoly and DeepZ on the MNIST FFNNMed and FFNNBig networks.

regions certified by DeepZ starting at  $\epsilon = 0.02$ . DeepZ certifies only 23% of the regions when  $\epsilon = 0.03$ ; in contrast, DeepPoly certifies 80%. Similarly, for the FFNNTanh network, DeepZ only certifies 1% of the regions when  $\epsilon = 0.015$ , whereas DeepPoly certifies 94%. We also note that DeepPoly is more than 2x faster than DeepZ on both these networks (we omit the relevant plots here as timings do not change with increasing values of  $\epsilon$ ): DeepZ has an average runtime of  $\leq 35$  seconds on both networks whereas DeepPoly has an average runtime of  $\leq 15$  seconds on both.

**MNIST CONVOLUTIONAL NETWORKS** Fig. 5.9 compares the precision and average runtime of DeepPoly vs DeepZ on the MNIST ConvSmall networks. We consider three types of ConvSmall networks based on their training method: (a) undefended (Point), (b) defended with PGD (PGD), and (c) defended with DiffAI (DiffAI). Note that our convolutional networks are more robust than the fully-connected networks

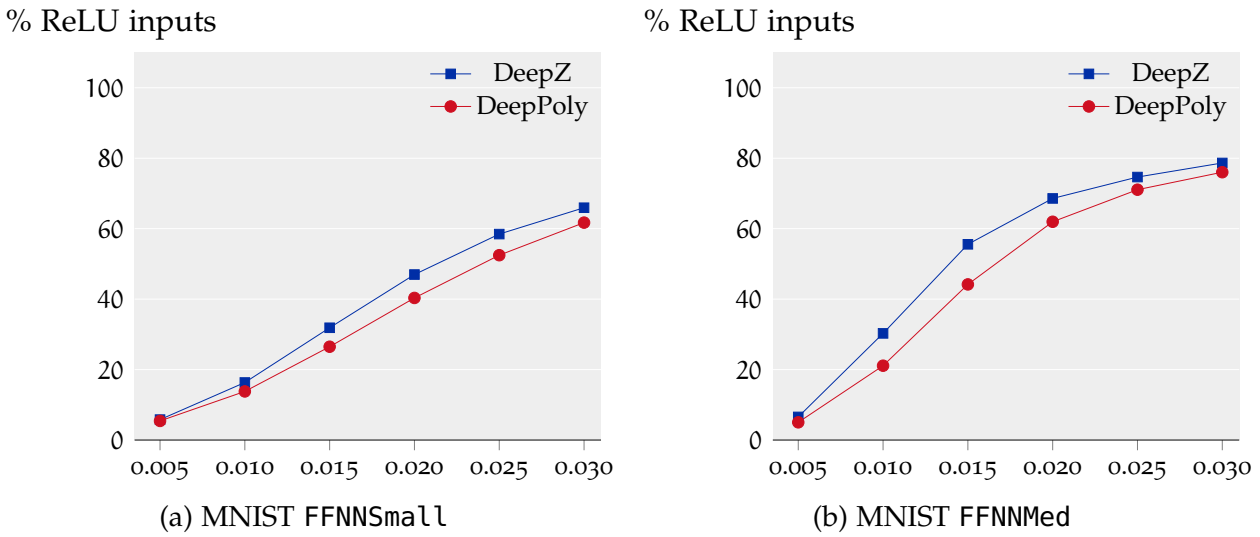


Figure 5.7: Average percentage of ReLU inputs that can take both positive and negative values for DeepPoly and DeepZ on the MNIST FFNSmall and FFNNMed networks.

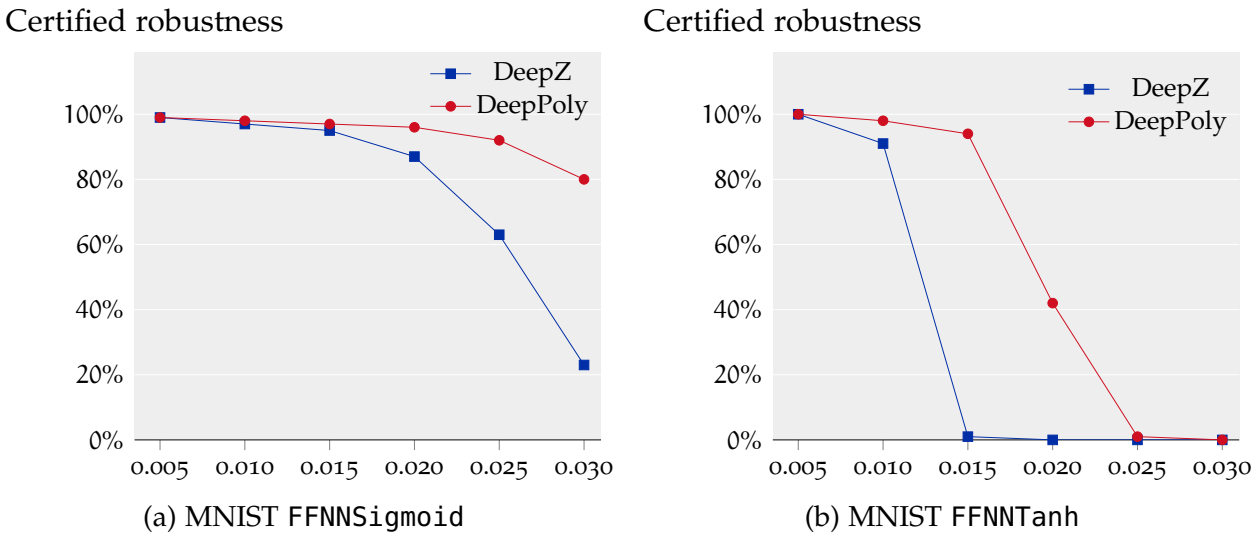


Figure 5.8: Certified robustness and average runtime for  $L_\infty$ -norm perturbations by DeepPoly and DeepZ on the MNIST FFNSigmoid and FFNNTanh networks.

and thus the values of  $\epsilon$  considered in our experiments are higher than those for fully-connected networks.

As expected, both DeepPoly and DeepZ certify more regions on the defended neural networks than on the undefended one. This is because the adversarially trained networks produce fewer inputs, where the ReLU transformer loses significant precision. We notice that `convSmall` trained with DiffAI is the most provably robust network. Overall, DeepPoly certifies more regions than DeepZ on all neural networks for all  $\epsilon$  values. The precision gap between DeepPoly and DeepZ increases with increasing  $\epsilon$ . For the largest  $\epsilon = 0.12$ , the percentage of regions certified by DeepZ on the Point, PGD, and DiffAI networks are 7%, 38%, and 53%



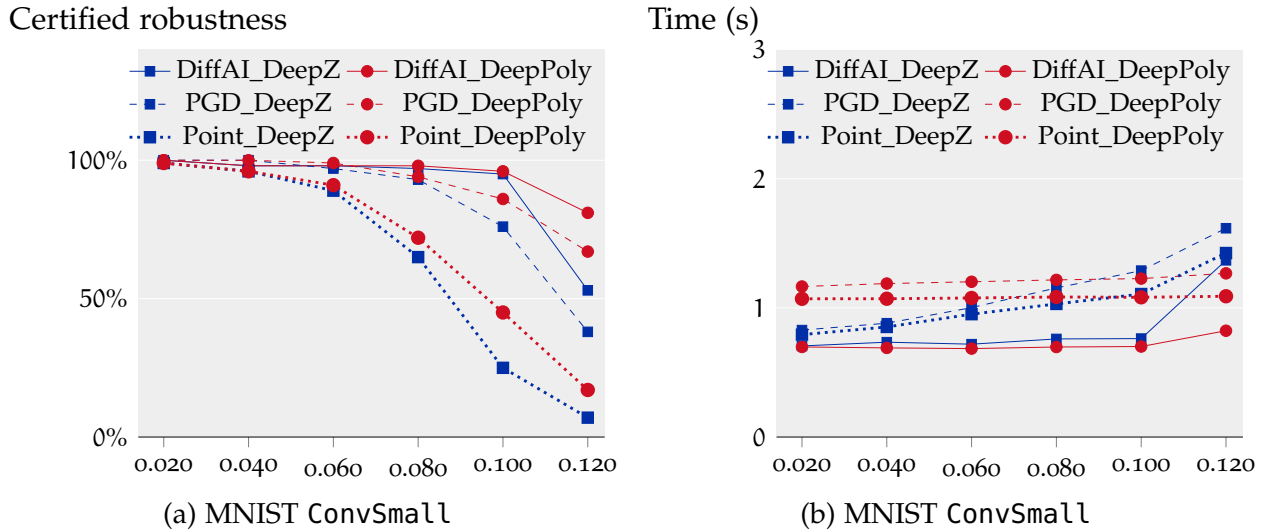


Figure 5.9: Certified robustness and average runtime for  $L_\infty$ -norm perturbations by DeepPoly and DeepZ on the MNIST ConvSmall networks.

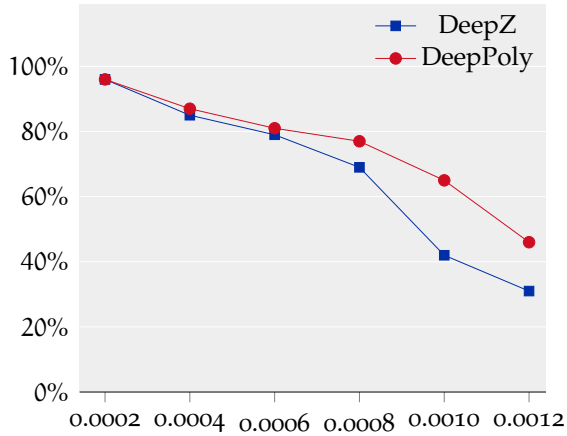
Table 5.2: Certified robustness by DeepZ and DeepPoly on the large convolutional networks trained with DiffAI.

Dataset	Model	$\epsilon$	% Certified robustness		Average runtime	
			DeepZ	DeepPoly	DeepZ	DeepPoly
MNIST	ConvBig	0.1	97	99	5	8
	ConvBig	0.2	79	88	7	8
	ConvBig	0.3	37	77	17	8
	ConvSuper	0.1	97	98	133	39
CIFAR10	ConvBig	0.006	50	52	39	23
	ConvBig	0.008	33	40	46	23

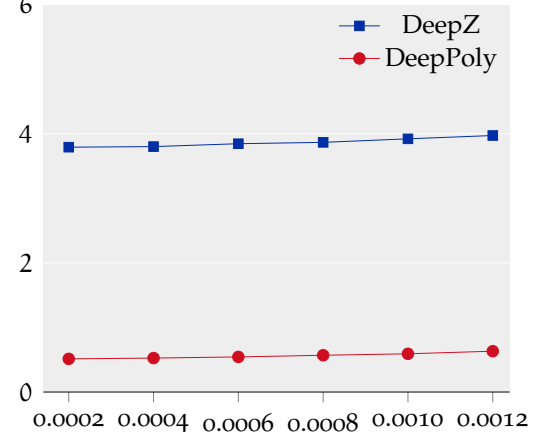
respectively whereas DeepPoly certifies 17%, 67%, and 81% regions respectively. The runtime of DeepZ increases with  $\epsilon$  while that of DeepPoly is not affected significantly. DeepPoly runs the fastest on the DiffAI network and is faster than DeepZ for all  $\epsilon$  values. DeepPoly is slower than DeepZ on the PGD and Point networks for smaller  $\epsilon$  values but faster on the largest  $\epsilon = 0.12$ .

Table 5.2 shows our experimental results on the larger MNIST convolutional networks trained using DiffAI. For the ConvBig network, DeepPoly certifies significantly more regions than DeepZ for  $\epsilon = 0.2$  and  $0.3$ . In particular, the percentage certified for  $\epsilon = 0.3$  with DeepPoly and DeepZ is 77% and 37%, respectively. For the ConvSuper network, DeepPoly certifies one more region than DeepZ for  $\epsilon = 0.1$ . In terms of the runtime, DeepZ runs slower with increasing value of  $\epsilon$  while DeepPoly is unaffected. DeepZ is slightly faster than DeepPoly on the ConvBig network

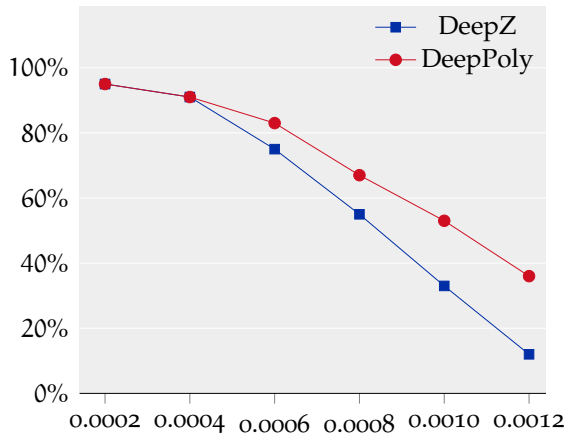
Certified robustness

(a) CIFAR<sub>10</sub> FFNSmall

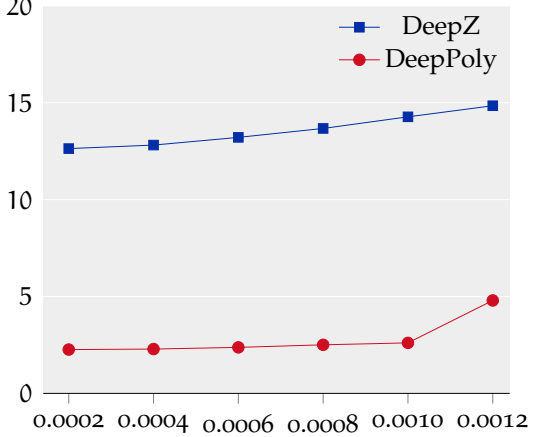
Time (s)

(b) CIFAR<sub>10</sub> FFNSmall

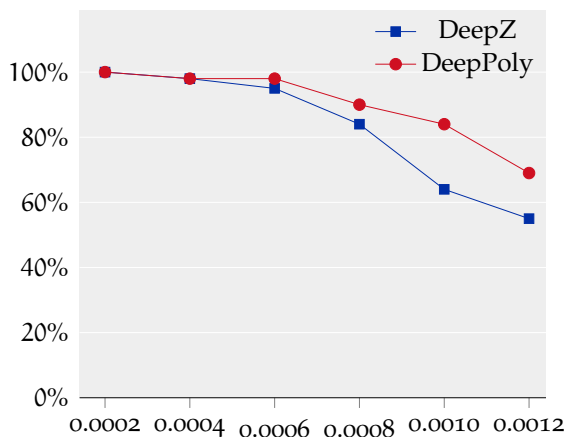
Certified robustness

(c) CIFAR<sub>10</sub> FFNNMed

Time (s)

(d) CIFAR<sub>10</sub> FFNNMed

Certified robustness

(e) CIFAR<sub>10</sub> FFNNBig

Time (s)

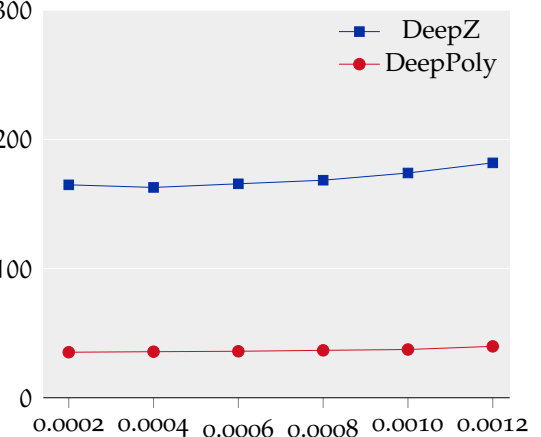
(f) CIFAR<sub>10</sub> FFNNBig

Figure 5.10: Certified robustness and average runtime for  $L_\infty$ -norm perturbations by DeepPoly and DeepZ on the CIFAR<sub>10</sub> fully-connected networks.

for  $\epsilon = 0.1$  and  $0.2$  but is 2x slower for  $\epsilon = 0.3$ . On the ConvSuper network, DeepPoly is 3.4x faster than DeepZ.

**CIFAR10 FULLY-CONNECTED NETWORKS** Fig. 5.10 compares DeepPoly against DeepZ on the CIFAR10 fully-connected networks. As with the MNIST fully-connected networks, DeepPoly certifies more regions than DeepZ and is faster on all the considered networks. Considering  $\epsilon = 0.001$ , DeepPoly certifies 65%, 53%, and 84% of the regions on the FFNNSmall, FFNNMed, and FFNNBig networks respectively whereas DeepZ certifies 42%, 33%, and 64% of the regions. Notice that the average runtime of both DeepPoly and DeepZ on the CIFAR10 FFNNMed is higher than on the MNIST FFNNMed network even though the number of hidden units is the same. The slowdown on the CIFAR10 networks is due to the higher number of input pixels. DeepPoly is up to 7x, 5x, and 4.5x faster than DeepZ on the FFNNSmall, FFNNMed, and FFNNBig networks, respectively.

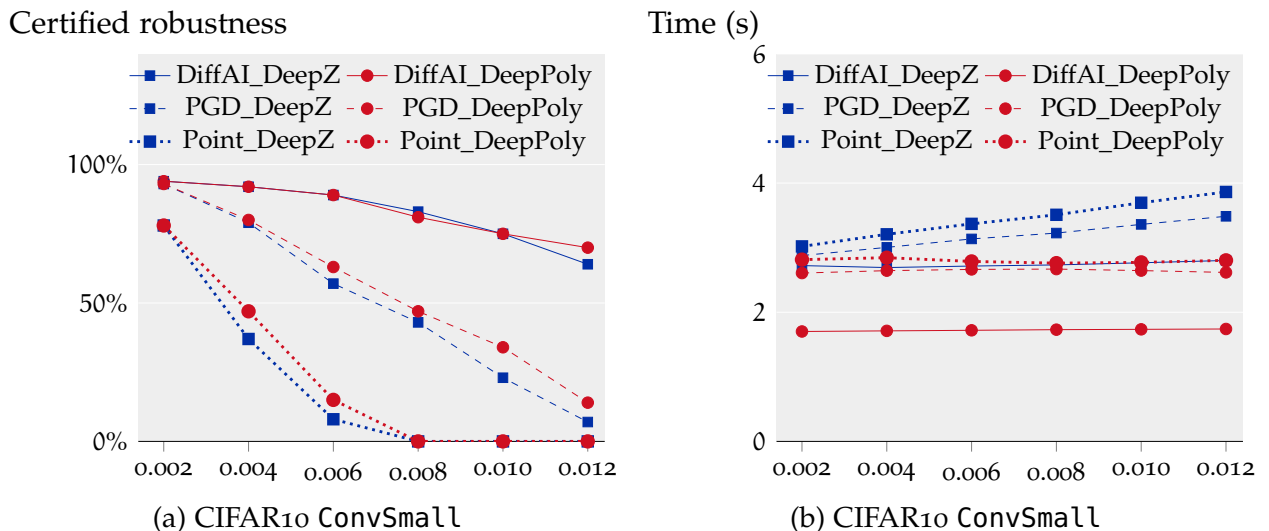


Figure 5.11: Certified robustness and average runtime for  $L_\infty$ -norm perturbations by DeepPoly and DeepZ on the CIFAR10 ConvSmall networks.

**CIFAR10 CONVOLUTIONAL NETWORKS** Fig. 5.11 evaluates DeepPoly and DeepZ on the CIFAR10 ConvSmall networks. We again consider undefended (Point) networks and networks defended with PGD and DiffAI as was the case for the corresponding MNIST network. We again notice that the ConvSmall network trained with DiffAI is the most provably robust network. DeepPoly certifies more regions than DeepZ for all values of  $\epsilon$  on the PGD and Point networks. DeepPoly is less precise than DeepZ on the DiffAI network for  $\epsilon = 0.008$  but certifies more for the largest  $\epsilon = 0.012$ . In terms of runtime, DeepPoly is faster than DeepZ on all the con-



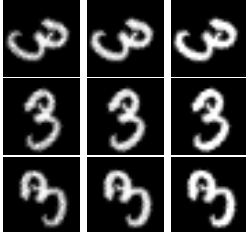
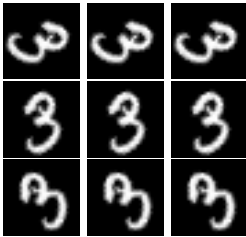
#Batches	Batch Size	Region(s) ( $l, \frac{1}{2}(l+u), u$ )	Analysis time	Certified?
1	1		0.5s + 1.9s	No
1	10000		22.2s + 1.8s	No
220	1		1.2s + 5m51s	No
220	300		2m29s + 5m30s	Yes

Figure 5.12: Results for robustness against rotations with the MNIST FFNNsmall network. Each row shows a different attempt to prove that the given image of the digit 3 can be perturbed within an  $L_\infty$  ball of radius  $\epsilon = 0.001$  and rotated by an arbitrary angle  $\theta$  between  $-45$  to  $65$  degrees without changing its classification. For the last two attempts, we show 4 representative combined regions (out of 220, one per batch). The running time is split into two components: (i) the time used for interval analysis on the rotation algorithm and (ii), the time used to prove the neural network robust with all of the computed bounding boxes using DeepPoly.

sidered ConvSmall networks for all  $\epsilon$  values. As was the case on the corresponding MNIST networks, DeepPoly runs fastest on the DiffAI network.

The last two rows in Table 5.2 compare the precision and performance of DeepPoly and DeepZ on the CIFAR10 ConvBig convolutional network trained with DiffAI. It can be seen that DeepPoly certifies more regions than DeepZ for both  $\epsilon = 0.006$  and  $\epsilon = 0.008$  and is also up to 2x faster.

### 5.5.3 Rotation perturbation

As described in Section 5.4, we can apply refinement to the input so to prove a neural network robust against rotations of a certain input image. Specifically, our analysis can prove that the MNIST FFNNsmall network classifies a given image of the digit 3 correctly, even if each pixel is first  $L_\infty$ -perturbed with  $\epsilon \leq 0.001$  and then rotated using an arbitrary angle  $\theta$  between  $-45$  and  $65$  degrees. Fig. 5.12

shows example regions and analysis times for several choices of parameters to the refinement approach. For example, #Batches = 220, Batch Size = 300 means that we split the interval  $[\alpha, \beta]$  into  $n = 220$  batches. To analyze a batch, we split the corresponding interval into  $m = 300$  input intervals for interval analysis, resulting in 300 regions for each batch. We then run DeepPoly on the smallest common bounding boxes of all regions in each batch, 220 times in total. Fig. 5.12 shows a few such bounding boxes in the Regions column. Note that it is not sufficient for certification to compute a single region that captures all rotated images. Fig. 5.12 shows two such attempts: one where we did not use batching (therefore, our interval analysis approach was applied to the rotation algorithm using an abstract  $\theta$  covering the entire range), and one where we used a batch size of 10,000 to compute the bounding box of the perturbations rather precisely. However, those perturbations cannot be captured well using interval constraints, therefore the bounding box contains many spurious inputs and the certification fails.

We then considered two certification attempts with 220 batches, with each batch covering a range of  $\theta$  of length 0.5 degrees. It was not sufficient to use a batch size of 1, as some input intervals become large. Using a batch size of 300, the neural network can be proved robust for this perturbation.

## 5.6 DISCUSSION

We introduced a new method for certifying deep neural networks which balances analysis precision and scalability. The core idea is an abstract domain based on combining floating-point polyhedra and intervals equipped with abstract transformers specifically designed for common neural network functions such as affine transforms, ReLU, sigmoid, tanh, and maxpool. These abstract transformers enable us to soundly handle both, fully-connected and convolutional networks.

We implemented our method in the ERAN analyzer, and evaluated it extensively on a wide range of networks of different sizes including defended and undefended networks. Our experimental results demonstrate that DeepPoly is more precise and faster than prior work. We also showed how to use DeepPoly to prove, for the first time, the robustness of a neural network when the input image is perturbed by complex transformations such as rotations employing linear interpolation.

In our follow-up work [152], we have extended the DeepPoly domain for handling residual networks as well as designed efficient algorithms for adapting DeepPoly on GPUs. The resulting implementation GPUPoly enabled precise and fast certification of large networks containing up to 1M neurons within a minute. In [15], we extended our robustness certification against rotation to cover more geometric transformations such as translation, scaling, and shearing as well as their arbitrary composition. Combined with GPUPoly, we can verify neural networks with up to 0.5M neurons against rotations within 30 minutes. All above results are

beyond the reach of any other existing certification method [8, 32, 36, 37, 67, 68, 69, 78, 113, 114, 135, 149, 163, 170, 175, 186, 197, 199, 206, 211, 212].

More recently, we designed new DeepPoly transformers for handling the specific non-linearities in the RNN architectures and audio preprocessing pipeline in [172]. This enables the certification of audio classifiers against intensity perturbations in the audio signal for the first time. We believe that DeepPoly domain can similarly be extended for handling other neural network architectures such as transformers [179], domains such as natural language processing [108], and specifications such as robustness against patches [217].

Overall, we believe this work is a promising step towards more effective reasoning about deep neural networks and a useful building block for proving interesting specifications as well as other applications of analysis (for example, training more robust networks).

---

## COMBINING ABSTRACTIONS WITH SOLVERS

---

The DeepPoly domain presented in Chapter 5 presents scalable approximations of the non-linearities employed in neural networks. This enables the analysis of larger networks than possible with exact methods based on SMT solving [37, 69, 113, 114], mixed-integer linear programming (MILP) [8, 32, 36, 49, 66, 135, 197], and Lipschitz optimization [170]. For deeper networks, the error from each approximation accumulates, exponentially with each layer in many practical cases, causing imprecision. In this chapter, we present a new approach for recovering the lost precision by combining abstract interpretation with precise solvers for ReLU based neural networks. Besides improving the precision of the approximations, the combination also improves the scalability of the solvers.

**THIS WORK: BOOSTING COMPLETE AND INCOMPLETE CERTIFIERS** Our first key idea is combining state-of-the-art overapproximation techniques used by incomplete methods together with MILP solvers. We refine the intermediate results computed via incomplete methods by calling the MILP solver. We provide results from the overapproximations to the MILP solver which improves its speed. This is because the MILP solvers must consider two paths per every ReLU input that can take both positive and negative values. The results from overapproximation can eliminate some of these branches while also reducing the search space for others. The above combination works well for refining the results in the first few layers, however, the MILP solver still does not scale for deeper layers as the number of branches becomes infeasible due to a combinatorial explosion.

The scalability issue can be addressed by employing convex relaxations of ReLU for refinement that are tighter than those employed by incomplete methods, for example, the DeepPoly domain, but more scalable than MILP solvers. A natural candidate is the most precise convex relaxation of ReLU output based on the convex hull of Polyhedra [57]. However, its computation is practically infeasible as it requires an exponential number of convex hull computations, each with a worst-case exponential complexity in the number of neurons. The most common convex relaxation of  $y_1 := \text{ReLU}(x_1)$  used in practice [175, 197] is the triangle relaxation from [69] shown in Fig. 5.4 (a). We note that other works such as [31, 186, 211, 212, 221] and the DeepPoly ReLU transformers shown in Fig. 5.4 (b) and (c) approximate this relaxation. The triangle relaxation creates constraints only between  $y_1$  and  $x_1$

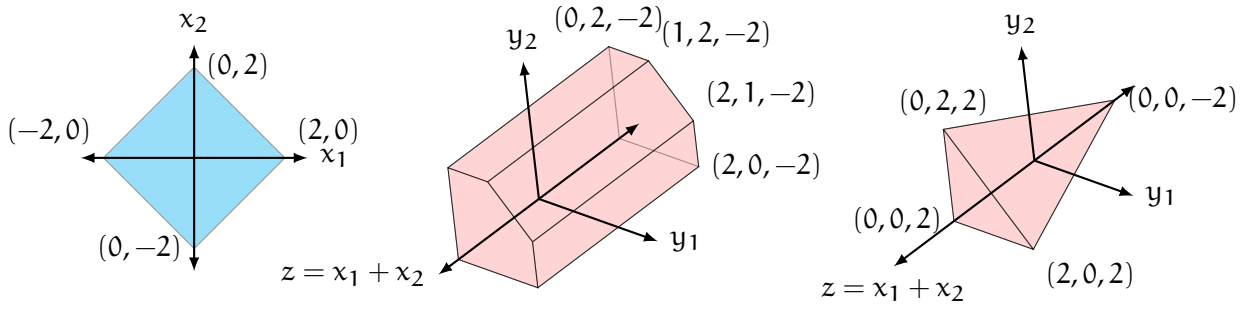


Figure 6.1: The input space for the ReLU assignments  $y_1 := \text{ReLU}(x_1)$ ,  $y_2 := \text{ReLU}(x_2)$  is shown on the left in blue. Shapes of the relaxations projected to 3D are shown on the right in red.

and is optimal in the  $x_1y_1$ -plane. Because of this optimality, recent work [175] refers to the triangle relaxation as the convex barrier, meaning the best convex approximation one can obtain when processing each ReLU separately. In our experiments, using this relaxation does not yield significant precision gains. Our main insight is that the triangle relaxation is *not optimal* when one considers multiple neurons at a time as it ignores all dependencies between  $x_1$  and any other neuron  $x_2$  in the same layer, and thus loses precision.

Our second key idea is proposing more precise but scalable convex relaxations than possible with prior work. We introduce a novel parameterized framework, called  $k$ -ReLU, for generating convex approximations that consider multiple ReLUs *jointly*. Here, the parameter  $k$  determines how many ReLUs are considered jointly with large  $k$  resulting in more precise output. For example, unlike prior work, our framework can generate a convex relaxation for  $y_1 := \text{ReLU}(x_1)$  and  $y_2 := \text{ReLU}(x_2)$  that is optimal in the  $x_1x_2y_1y_2$ -space. Next, we illustrate this point with an example.

**PRECISION GAIN WITH  $k$ -RELU ON AN EXAMPLE** Consider the input space of  $x_1x_2$  as defined by the blue area in Fig. 6.1 and the ReLU operations  $y_1 := \text{ReLU}(x_1)$  and  $y_2 := \text{ReLU}(x_2)$ . The input space is bounded by the relational constraints  $x_2 - x_1 \leq 2$ ,  $x_1 - x_2 \leq 2$ ,  $x_1 + x_2 \leq 2$  and  $-x_1 - x_2 \leq 2$ . The relaxations produced are in a four dimensional space of  $x_1x_2y_1y_2$ . For simplicity of presentation, we show the feasible shape of  $y_1y_2$  as a function of  $z = x_1 + x_2$ .

The triangle relaxation from [69] is in fact a special case of our framework with  $k = 1$ , that is, 1-ReLU. 1-ReLU independently computes two relaxations - one in the  $x_1y_1$  space and the other in the  $x_2y_2$  space. The final relaxation is the cartesian product of the feasible sets of the two individually computed relaxations and is oblivious to any correlations between  $x_1$  and  $x_2$ . The relaxation adds triangle constraints  $\{y_1 \geq 0, y_1 \geq x_1, y_1 \leq 0.5 \cdot x_1 + 1\}$  between  $x_1$  and  $y_1$  as well as  $\{y_2 \geq 0, y_2 \geq x_2, y_2 \leq 0.5 \cdot x_2 + 1\}$  between  $x_2$  and  $y_2$ .



Table 6.1: Volume of the output bounding box from kPoly on the MNIST FFNNMed network.

k	1-ReLU	2-ReLU	3-ReLU
Volume	$4.5272 \cdot 10^{14}$	$5.1252 \cdot 10^7$	$2.9679 \cdot 10^5$

In contrast, 2-ReLU considers the two ReLU’s *jointly* and captures the relational constraints between  $x_1$  and  $x_2$ . 2-ReLU computes the following relaxation:

$$\{y_1 \geq 0, y_1 \geq x_1, y_2 \geq 0, y_2 \geq x_2, 2 \cdot y_1 + 2 \cdot y_2 - x_1 - x_2 \leq 2\}$$

The result is shown in Fig. 6.1 (c). In this case the shape of  $y_1 y_2$  is not independent of  $x_1 + x_2$  as opposed to the triangle relaxation. At the same time, it is more precise than Fig. 6.1 (b) for all values of  $z$ . We note that the work of [163] computes semi definite relaxations that consider multiple ReLUs jointly, however these are not optimal and do not scale to the large networks used in our experiments.

The work in this chapter was published in [185, 187].

**MAIN CONTRIBUTIONS** Our main contributions are:

- A refinement-based approach for certifying neural network robustness that combines the strengths of fast overapproximation methods with MILP solvers and convex relaxations.
- A novel framework, called k-ReLU, that computes optimal convex relaxations for the output of k ReLU operations *jointly*. k-ReLU is generic and can be combined with existing certifiers for improved precision while maintaining scalability. Further, k-ReLU is also adaptive and can be tuned to balance precision and scalability by varying k.
- A method for computing approximations of the optimal relaxations for larger k, which is more precise than simply using  $l < k$ .
- An instantiation of k-ReLU with the DeepPoly domain [188] resulting in a certifier called kPoly.
- An evaluation, showing that kPoly is more precise than existing state-of-the-art incomplete certifiers [187, 188] on larger networks with up to 100K neurons against challenging adversarial perturbations (e.g.,  $L_\infty$  balls with  $\epsilon = 0.3$ ) and faster (while being complete) than state-of-the-art complete certifiers [197, 206] on smaller networks.

**PRECISION GAIN IN PRACTICE** Table 6.1 quantitatively compares the precision of kPoly instantiated with three relaxations:  $k = 1$ ,  $k = 2$ , and  $k = 3$ . We measure the volume of the output bounding box computed after propagating an  $L_\infty$ -norm based region with  $\epsilon = 0.015$  through the 9 layer deep MNIST FFNNMed network of Table 5.1. We observe that the output volume from 3-ReLU and 2-ReLU is respectively 9 and 7 orders of magnitude smaller than from 1-ReLU. We note that the networks we consider, as for example the FFNNMed network above, are especially challenging for state-of-the-art certifiers as these methods either lose unnecessary precision [31, 175, 186, 187, 188, 199, 211, 221] or simply do not scale [37, 67, 68, 163, 197, 206].

## 6.1 OVERVIEW

We now show, on a simple example, the working of our certifier kPoly combining the k-ReLU concept with refinement improves the results of state-of-the-art certifiers. In particular, we illustrate how the output kPoly instantiated with 1-ReLU is refined by instantiating it with 2-ReLU. This is possible as the 2-ReLU relaxation can capture extra relationships between neurons that 1-ReLU inherently cannot.

Consider the simple fully-connected neural network with ReLU activations shown in Fig. 6.2. The network has two inputs each taking values independently in the range  $[-1, 1]$ , one hidden layer and one output layer each containing two neurons. For simplicity, we split each layer into two parts: one for the affine transformation and the other for the ReLU (as in Fig. 5.3). The weights of the affine transformation are shown on the arrows and the biases are above or below the respective neuron. The goal is to certify that  $x_9 \leq 4$  holds for the output  $x_9$  with respect to all inputs.

We first show that 1-ReLU instantiated with the state-of-the-art DeepPoly [188] abstract domain fails to certify the property. We refer the reader to Chapter 5 for more details on the DeepPoly abstract domain. The bounds computed by our certifier using this instantiation are shown as annotations in Fig. 6.2, in the same format as in Fig. 5.3. We next show how our analysis proceeds layer-by-layer.

**FIRST LAYER** The certifier starts by computing the bounds for  $x_1$  and  $x_2$  which are simply taken from the input specification resulting in:

$$\begin{aligned} x_1 &\geq -1, x_1 \leq 1, l_1 = -1, u_1 = 1, \\ x_2 &\geq -1, x_2 \leq 1, l_2 = -1, u_2 = 1. \end{aligned}$$

**SECOND LAYER** Next, the affine assignments  $x_3 := x_1 + x_2$  and  $x_4 := x_1 - x_2$  are handled. DeepPoly handles affine transformations exactly and thus no precision is lost. The affine transformation results in the following bounds for  $x_3$  and  $x_4$ :

$$\begin{aligned} x_3 &\geq x_1 + x_2, x_3 \leq x_1 + x_2, l_3 = -2, u_3 = 2, \\ x_4 &\geq x_1 - x_2, x_4 \leq x_1 - x_2, l_4 = -2, u_4 = 2. \end{aligned}$$

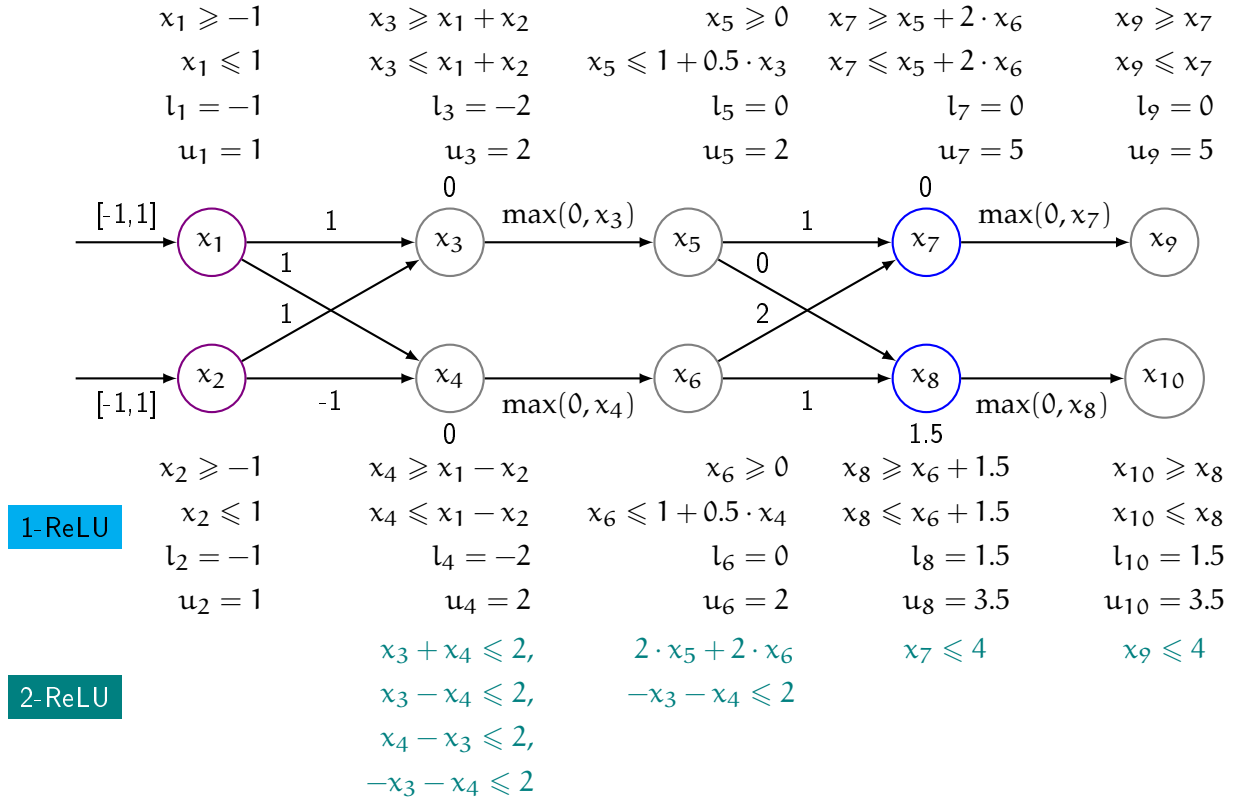


Figure 6.2: Certification of property  $x_9 \leq 2$ . Refining DeepPoly with 1-ReLU fails to prove the property whereas 2-ReLU adds extra constraints (in green) that help in verifying the property.

DeepPoly can precisely handle ReLU assignments when the input neuron takes only positive or negative values; otherwise, it loses precision. Since  $x_3$  and  $x_4$  can take both positive and negative values, the approximation from Fig. 5.4 (b) is applied which for  $x_5$  yields:

$$x_5 \geq 0, \quad x_5 \leq 1 + 0.5 \cdot x_3, \quad l_5 = 0, \quad u_5 = 2. \quad (6.1)$$

The lower and upper bounds are set to  $l_5 = 0$  and  $u_5 = 2$  respectively. Analogously, for  $x_6$  we obtain:

$$x_6 \geq 0, \quad x_6 \leq 1 + 0.5 \cdot x_4, \quad l_6 = 0, \quad u_6 = 2. \quad (6.2)$$

**THIRD LAYER** Next, the affine assignments  $x_7 := x_5 + 2x_6$  and  $x_8 := x_6 + 1.5$  are handled. DeepPoly adds the constraints:

$$\begin{aligned} x_7 &\geq x_5 + 2 \cdot x_6, & x_7 &\leq x_5 + 2 \cdot x_6, \\ x_8 &\geq x_6 + 1.5, & x_8 &\leq x_6 + 1.5. \end{aligned} \quad (6.3)$$

To compute the upper and lower bounds for  $x_7$  and  $x_8$ , DeepPoly uses back-substitution as described in Section 5.1. Doing so yields  $l_7 = 0, u_7 = 5$  and  $l_8 = 1.5, u_8 = 3.5$ .

**REFINEMENT WITH 1-RELU FAILS** Because DeepPoly discards one of the lower bounds from the triangle relaxations for the ReLU assignments in the previous layer, it is possible to refine lower and upper bounds for  $x_7$  and  $x_8$  by encoding the network up to the final affine transformation using the relatively tighter ReLU relaxations based on the triangle formulation and then computing bounds for  $x_7$  and  $x_8$  with respect to this formulation via an LP solver. However, this does not improve bounds and still yields  $l_7 = 0, u_7 = 5, l_8 = 1.5, u_8 = 3.5$ .

As the lower bounds for both  $x_7$  and  $x_8$  are non-negative, the DeepPoly ReLU approximation simply propagates  $x_7$  and  $x_8$  to the output layer. Therefore the final output is:

$$\begin{aligned} x_9 &\geq x_7, & x_9 &\leq x_7, & l_9 &= 0, & u_9 &= 5, \\ x_{10} &\geq x_8, & x_{10} &\leq x_8, & l_{10} &= 1.5, & u_{10} &= 3.5. \end{aligned}$$

Because the upper bound is  $u_9 = 5$ , the certifier fails to prove the property that  $x_9 \leq 4$  holds.

**REFINEMENT WITH 2-RELU CERTIFIES THE PROPERTY** Now we consider refinement with our 2-ReLU relaxation which considers the two ReLU assignments  $x_5 := \text{ReLU}(x_3)$  and  $x_6 := \text{ReLU}(x_4)$  *jointly*. Besides the box constraints for  $x_3$  and  $x_4$ , it also considers the constraints  $x_3 + x_4 \leq 2, x_3 - x_4 \leq -2, -x_3 - x_4 \leq 2, x_4 - x_3 \leq 2$  for computing the output of ReLU. The ReLU output contains the extra constraint  $2 \cdot x_5 + 2 \cdot x_6 - x_3 - x_4 \leq 2$  that 1-ReLU cannot capture. We again encode the network up to the final affine transformation with the tighter ReLU relaxations obtained using 2-ReLU and refine the bounds for  $x_7, x_8$  via an LP solver. Now, we obtain better upper bounds as  $u_7 = 4$ . The better bound for  $u_7$  is then propagated to  $u_9$  and is sufficient for proving the desired property.

We remark that while in this work we instantiate the k-ReLU concept with the DeepPoly relaxation, the idea can be applied to other relaxations [67, 78, 163, 175, 188, 191, 206, 211, 212, 221].

Alternatively, one can also refine the bounds for the neurons  $x_7$  and  $x_8$  by replacing 2-ReLU with the MILP encoding of ReLU from [197] and also adding the extra constraints from the DeepPoly analysis to speed it up. Doing so also certifies the property. However, the MILP encoding is less scalable than our k-ReLU framework and is feasible only for the first few layers. In our experiments in Section 6.5, we use the MILP encoding for refining up to the second layer and k-ReLU for the remaining layers of our deep networks.

## 6.2 REFINEMENT WITH SOLVERS

We now describe our refinement approach in more formal terms. As in Section 6.1, we will consider affine transformations and ReLU activations as separate layers. The key idea will be to combine abstract interpretation [55] with exact MILP or precise convex relaxation based formulations of the network, which are then solved, in

order to compute more precise results for neuron bounds. We begin by describing the core components of abstract interpretation that our approach requires.

Our approach requires an abstract domain  $\mathbb{A}_n$  over  $n$  variables (i.e., some set whose elements can be encoded symbolically) such as Interval, Zonotope, Deep-Poly, or Polyhedra. The abstract domain has a bottom element  $\perp \in \mathbb{A}_n$  as well as the following components:

- A (potentially non-computable) concretization function  $\gamma_n: \mathbb{A}_n \rightarrow \mathcal{P}(\mathbb{R}^n)$  that associates with each abstract element  $a \in \mathbb{A}_n$  the set of concrete points from  $\mathbb{R}^n$  that it abstracts. We have  $\gamma_n(\perp) = \emptyset$ .
- An abstraction function  $\alpha_n: \mathbb{B}_n \rightarrow \mathbb{A}_n$ , where  $\mathbb{X} \subseteq \gamma_n(\alpha_n(\mathbb{X}))$  for all  $\mathbb{X} \in \mathbb{B}_n$ . We assume that  $\alpha_n(\prod_i [l_i, u_i])$  is a computable function of  $\mathbf{l}, \mathbf{u} \in \mathbb{R}^n$ . Here,  $\mathbb{B}_n = \bigcup_{\mathbf{l}, \mathbf{u} \in \mathbb{R}^n} \prod_i [l_i, u_i]$  and  $\prod_i [l_i, u_i] = \{\mathbf{x} \in \mathbb{R}^n \mid l_i \leq x_i \leq u_i\}$ . (For many abstract domains,  $\alpha_n$  can be defined on a larger domain  $\mathbb{B}_n$ , but in this work, we only consider Interval input regions.)
- A bounding box function  $\iota_n: \mathbb{A}_n \rightarrow \mathbb{R}^n \times \mathbb{R}^n$ , where  $\gamma_n(a) \subseteq \prod_i [l_i, u_i]$  for  $(\mathbf{l}, \mathbf{u}) = \iota_n(a)$  for all  $a \in \mathbb{A}_n$ .
- A meet operation  $a \sqcap L$  for each  $a \in \mathbb{A}_n$  and linear constraints  $L$  over  $n$  real variables, where  $\{x \in \gamma_n(a) \mid L(x)\} \subseteq \gamma_n(a \sqcap L)$ .
- An affine abstract transformer  $T_{\mathbf{x} \mapsto \mathbf{A}\mathbf{x} + \mathbf{b}}^\# : \mathbb{A}_m \rightarrow \mathbb{A}_n$  for each transformation of the form  $(\mathbf{x} \mapsto \mathbf{A}\mathbf{x} + \mathbf{b}): \mathbb{R}^m \rightarrow \mathbb{R}^n$ , where

$$\{\mathbf{A}\mathbf{x} + \mathbf{b} \mid \mathbf{x} \in \gamma_n(a)\} \subseteq \gamma_n(T_{\mathbf{x} \mapsto \mathbf{A}\mathbf{x} + \mathbf{b}}^\#(a))$$

for all  $a \in \mathbb{A}_m$ .

- A ReLU abstract transformer  $T_{\text{ReLU}|_{\prod_i [l_i, u_i]}}^\# : \mathbb{A}_n \rightarrow \mathbb{A}_n$ , where

$$\{\text{ReLU}(\mathbf{x}) \mid \mathbf{x} \in \gamma_n(a) \cap \prod_i [l_i, u_i]\} \subseteq T_{\text{ReLU}|_{\prod_i [l_i, u_i]}}^\#(a)$$

for all abstract elements  $a \in \mathbb{A}_n$  and for all lower and upper bounds  $\mathbf{l}, \mathbf{u} \in \mathbb{R}^n$  on input activations of the ReLU operation.

**CERTIFICATION VIA ABSTRACT INTERPRETATION** As first shown by [78], any such abstract domain induces a method for robustness certification of neural networks with ReLU activations.

For example, assume that we want to certify that a given neural network  $f: \mathbb{R}^m \rightarrow \mathbb{R}^n$  considers class  $i$  more likely than class  $j$  for all inputs  $\bar{\mathbf{x}}$  with  $\|\bar{\mathbf{x}} - \mathbf{x}\|_\infty \leq \epsilon$  for a given  $\mathbf{x}$  and  $\epsilon$ . We can first use the abstraction function  $\alpha_m$  to compute a symbolic overapproximation of the set of possible inputs  $\bar{\mathbf{x}}$ , namely

$$\mathbf{a}_{\text{in}} = \alpha_m(\{\bar{\mathbf{x}} \in \mathbb{R}^m \mid \|\bar{\mathbf{x}} - \mathbf{x}\|_\infty \leq \epsilon\}).$$

Given that the neural network can be written as a composition of affine functions and ReLU layers, we can then propagate the abstract element  $\alpha_{\text{in}}$  through the corresponding abstract transformers to obtain a symbolic overapproximation  $\alpha_{\text{out}}$  of the concrete outputs of the neural network.

For example, if the neural network  $f(\mathbf{x}) = \mathbf{A}' \cdot \text{ReLU}(\mathbf{A}\mathbf{x} + \mathbf{b}) + \mathbf{b}'$  has a single hidden layer with  $h$  hidden neurons, we first compute  $\alpha' = T_{\mathbf{x} \rightarrow \mathbf{A}\mathbf{x} + \mathbf{b}}^{\#}(\alpha_{\text{in}})$ , which is a symbolic overapproximation of the inputs to the ReLU activation function. We then compute  $(\mathbf{l}, \mathbf{u}) = \iota_h(\alpha')$  to obtain opposite corners of a bounding box of all possible ReLU input activations, such that we can apply the ReLU abstract transformer:

$$\alpha'' = T_{\text{ReLU}|_{\prod_i [l_i, u_i]}}^{\#}(\alpha').$$

Finally, we apply the affine abstract transformer again to obtain  $\alpha_{\text{out}} = T_{\mathbf{x} \rightarrow \mathbf{A}'\mathbf{x} + \mathbf{b}'}^{\#}(\alpha'')$ . Using our assumptions, we can conclude that the set  $\gamma_n(\alpha_{\text{out}})$  contains all output activations that  $f$  can possibly produce when given any of the inputs  $\bar{\mathbf{x}}$ . Therefore, if  $\alpha_{\text{out}} \sqcap (x_i \leq x_j) = \perp$ , we have proved the property: for all  $\bar{\mathbf{x}}$ , the neural network considers class  $i$  more likely than class  $j$ .

**INCOMPLETENESS** While this approach is sound (i.e., whenever we prove the property, it actually holds), it is incomplete (i.e., we might not prove the property, even if it holds), because the abstract transformers produce a superset of the set of concrete outputs that the corresponding concrete executions produce. This can be quite imprecise for deep neural networks, because the overapproximations introduced in each layer accumulate.

**REFINING THE BOUNDS** To combat spurious overapproximation, we use mixed integer linear programming (MILP) to compute refined lower and upper bounds  $\mathbf{l}', \mathbf{u}'$  after applying each affine abstract transformer (except for the first layer). We then refine the abstract element using the meet operator of the underlying abstract domain and the linear constraints  $l'_j \leq x_j \leq u'_j$  for all input activations  $i$ , i.e., we replace the current abstract element  $\alpha$  by  $\alpha' = \alpha \sqcap (\bigwedge_j l'_j \leq x_j \leq u'_j)$ , and continue analysis with the refined abstract element.

Importantly, we obtain a more refined abstract transformer for ReLU than the one used in DeepPoly by leveraging the new lower and upper bounds. That is, using the tighter bounds  $l'_j, u'_j$  for  $x_j$ , we define the ReLU transformer, using the notation from Chapter 5, for  $x_i := \max(0, x_j)$  as follows:

$$\langle \alpha_i^{\leq}(x), \alpha_i^{\geq}(x), l_i, u_i \rangle = \begin{cases} \langle x_j, x_j, l'_j, u'_j \rangle, & \text{if } l'_j > 0, \\ \langle 0, 0, 0, 0 \rangle, & \text{if } u'_j \leq 0, \\ \langle \lambda \cdot x_j, u'_j \cdot (x_j - l'_j) / (u'_j - l'_j), \lambda \cdot l'_j, u'_j \rangle, & \text{otherwise.} \end{cases}$$

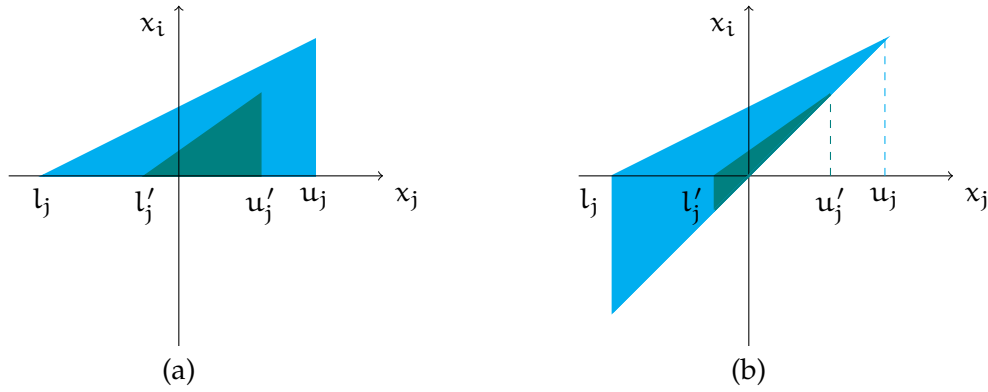


Figure 6.3: DeepPoly relaxations for  $x_i := \text{ReLU}(x_j)$  using the original bounds  $l_j, u_j$  (in blue) and the refined bounds  $l'_j, u'_j$  (in green) for  $x_j$ . The refined relaxations have smaller area in the  $x_i x_j$ -plane.

where  $\lambda = \{0, 1\}$ . The refined ReLU transformer benefits from the improved bounds. For example, when  $l_j < 0$  and  $u_j > 0$  holds for the original bounds then after refinement:

- If  $l'_j > 0$ , then the relational constraints are the same, however the interval bounds are more precise.
- Else if  $u'_j \leq 0$ , then the output is exact.
- Otherwise, as shown in Fig. 6.3, the approximation with the tighter  $l'_j$  and  $u'_j$  has smaller area (in green) in the input-output plane than the original transformer that uses the imprecise  $l_j$  and  $u_j$  (in blue).

**OBTAINING CONSTRAINTS FOR REFINEMENT** To enable refinement with MILP, we need to obtain constraints which fully capture the behavior of the neural network up to the last layer whose abstract transformer has been executed. In our encoding, we have one variable for each neuron and we write  $x_i^{(k)}$  to denote the variable corresponding to the activation of the  $i$ -th neuron in the  $k$ -th layer, where the input layer has  $k = 0$ . Similarly, we write  $l_i^{(k)}$  and  $u_i^{(k)}$  to denote the best derived lower and upper bounds for this neuron.

From the input layer, we obtain constraints of the form  $l_i^0 \leq x_i^{(0)} \leq u_i^0$ , from affine layers, we obtain constraints of the form  $x_i^{(k)} = \sum_j a_{ij}^{(k-1)} \cdot x_j^{(k-1)} + b_i^{(k-1)}$  and from ReLU layers we obtain constraints of the form  $x_i^{(k)} = \max(0, x_i^{(k-1)})$ .

**MILP** Let  $\varphi^{(k)}$  denote the conjunction of all constraints up to and including those from layer  $k$ . To obtain the best possible lower and upper bounds for layer  $k$  with  $p$  neurons, we need to solve the following  $2 \cdot p$  optimization problems:

$$\begin{aligned} \mathbf{l}'^{(k)} &= \min_{x_1^{(0)}, \dots, x_p^{(k)}} x_i^{(k)}, \text{ for } i = 1, \dots, p, \\ &\text{s.t. } \varphi^{(k)}(x_1^{(0)}, \dots, x_p^{(k)}) \\ \mathbf{u}'^{(k)} &= \max_{x_1^{(0)}, \dots, x_p^{(k)}} x_i^{(k)}, \text{ for } i = 1, \dots, p. \\ &\text{s.t. } \varphi^{(k)}(x_1^{(0)}, \dots, x_p^{(k)}) \end{aligned}$$

As was shown by [8, 197], such optimization problems can be encoded exactly as MILP instances using the bounds computed by abstract interpretation and the instances can then be solved using off-the-shelf MILP solvers [92] to compute  $\mathbf{l}'^{(k)}$  and  $\mathbf{u}'^{(k)}$ .

**CONVEX RELAXATIONS** While not introducing any approximation, unfortunately, current MILP solvers do not scale to larger neural networks. It becomes increasingly more expensive to refine bounds with the MILP-based formulation as the analysis proceeds deeper into the network. However, for soundness it is not crucial that the produced bounds are the best possible: for example, plain abstract interpretation uses sound bounds produced by the bounding box function  $\iota$  instead. Therefore, for deeper layers in the network, we explore the trade-off between precision and scalability by also considering an intermediate method, which is faster than exact MILP, but also more precise than abstract interpretation. We relax the constraints in  $\varphi^{(k)}$  using the convex set computed by our  $k$ -ReLU framework, formally introduced in Section 6.3, to obtain a set of weaker linear constraints  $\varphi_{\text{LP}}^{(k)}$ . We then use the solver to solve the relaxed optimization problems that are constrained by  $\varphi_{\text{LP}}^{(k)}$  instead of  $\varphi^{(k)}$ , producing possibly looser bounds  $\mathbf{l}'^{(k)}$  and  $\mathbf{u}'^{(k)}$ . Note that the encoding of subsequent layers depends on the bounds computed in previous layers, where tighter bounds reduce the amount of newly introduced approximation.

**ANYTIME MILP RELAXATION** MILP solvers usually provide the option to provide an explicit timeout  $T$  after which the solver must terminate. In return, the solver may not be able to solve the instance exactly, but it will instead provide lower and upper bounds on the objective function in a best-effort fashion. This provides another way to compute sound but inexact bounds  $\mathbf{l}'^{(k)}$  and  $\mathbf{u}'^{(k)}$ .

**NEURON SELECTION HEURISTIC FOR REFINEMENT** We select all neurons for refinement in an affine layer that can be proven to be only taking positive values using abstract interpretation.



**KPOLY: END-TO-END APPROACH** To certify deep neural networks, we combine MILP, LP relaxation, and abstract interpretation. We first pick numbers of layers  $k_{\text{MILP}}, k_{\text{LP}}, k_{\text{AI}}$  that sum to the total number of layers of the neural network. For the analysis of the first  $k_{\text{MILP}}$  layers, we refine bounds using anytime MILP relaxation with the neuron selection heuristic. As an optimization, we do not perform refinement after the abstract transformer for the first layer in case it is an affine transformation, as the abstract domain computes the tightest possible bounding box for an affine transformation of a box (this is always the case in our experiments). For the next  $k_{\text{LP}}$  layers, we refine bounds using LP relaxation (i.e., the network up to the layer to be refined is encoded using linear constraints computed via k-ReLU framework) combined with the neuron selection heuristic. For the remaining  $k_{\text{AI}}$  layers, we use abstract interpretation without additional refinement (however, this also benefits from refinement that was performed in previous layers), and compute the bounds using  $\iota$ .

**FINAL PROPERTY CERTIFICATION** Let  $k$  be the index of the last layer and  $p$  be the number of output classes. We can encode the final certification problem using the output abstract element  $\alpha_{\text{out}}$  obtained after applying the abstract transformer for the last layer in the network. If we want to prove that the output satisfies the property  $\psi$ , where  $\psi$  is given by a CNF formula  $\bigwedge_i \bigvee_j l_{i,j}$  with all literals  $l_{i,j}$  being linear constraints, it suffices to show that  $\alpha_{\text{out}} \sqcap (\bigwedge_j \neg l_{i,j}) = \perp$  for all  $i$ . If this fails, one can resort to complete verification using MILP: the property is satisfied if and only if the set of constraints  $\varphi^{(k)}(x_1^{(0)}, \dots, x_p^{(k)}) \wedge (\bigwedge_j \neg l_{i,j})$  is unsatisfiable for all  $i$ .

### 6.3 K-RELU RELAXATION FRAMEWORK

In this section we formally describe our k-ReLU framework for generating optimal convex relaxations in the input-output space for  $k$  ReLU operations jointly. In the next section, we discuss the instantiation of our framework with existing certifiers which enables more precise results.

We consider a ReLU based fully-connected, convolutional, or residual neural network with  $h$  neurons from a set  $\mathcal{H}$  (that is  $h = |\mathcal{H}|$ ) and a bounded input region  $\mathcal{J} \subseteq \mathbb{R}^m$  where  $m < h$  is the number of neural network inputs. As before, we treat the affine transformation and the ReLUs as separate layers. We consider a convex approximation method  $M$  that processes network layers in a topologically sorted sequence from the input to the output layer passing the output of predecessor layers as input to the successor layers. Let  $\mathcal{S} \subseteq \mathbb{R}^h$  be a convex set computed via  $M$  approximating the set of values that neurons up to layer  $l-1$  can take with respect to  $\mathcal{J}$  and  $\mathcal{B} \supseteq \mathcal{S}$  be the smallest bounding box around  $\mathcal{S}$ . We use  $\text{Conv}(\mathcal{S}_1, \mathcal{S}_2)$  and  $\mathcal{S}_1 \cap \mathcal{S}_2$  to denote the convex hull and the intersection of convex sets  $\mathcal{S}_1$  and  $\mathcal{S}_2$ , respectively.

Let  $\mathcal{X}, \mathcal{Y} \subseteq \mathcal{H}$  be respectively the set of input and output neurons in the  $l$ -th layer consisting of  $n$  ReLU assignments of the form  $y_i := \text{ReLU}(x_i)$  where  $x_i \in \mathcal{X}$  and  $y_i \in \mathcal{Y}$ . We assume that each input neuron  $x_i$  takes on both positive and negative values in  $\mathcal{S}$ . We define the polyhedra induced by the two branches of each ReLU assignment  $y_i := \text{ReLU}(x_i)$  as  $\mathcal{C}_i^+ = \{x_i \geq 0, y_i = x_i\} \subseteq \mathbb{R}^h$  and  $\mathcal{C}_i^- = \{x_i \leq 0, y_i = 0\} \subseteq \mathbb{R}^h$ . Let  $\mathcal{Q}_J = \{\bigcap_{i \in J} \mathcal{C}_i^{s(i)} \mid s \in J \rightarrow \{-, +\}\}$  (where  $J \subseteq [n]$ ) be the set of polyhedra  $Q \subseteq \mathbb{R}^h$  constructed by the intersection of polyhedra  $\mathcal{C}_i \subseteq \mathbb{R}^h$  for neurons  $x_i, y_i$  indexed by the set  $J$  such that each  $\mathcal{C}_i \in \{\mathcal{C}_i^+, \mathcal{C}_i^-\}$ .

**Example 6.3.1.** For the ReLU assignments  $y_i := \text{ReLU}(x_i)$  with  $1 \leq i \leq 2$ , and  $J = [2] = \{1, 2\}$ , we have that  $\mathcal{C}_1^+ = \{x_1 \geq 0, y_1 = x_1\}$ ,  $\mathcal{C}_1^- = \{x_1 \leq 0, y_1 = 0\}$ ,  $\mathcal{C}_2^+ = \{x_2 \geq 0, y_2 = x_2\}$ , and  $\mathcal{C}_2^- = \{x_2 \leq 0, y_2 = 0\}$ .  $\mathcal{Q}_J$  contains 4 polyhedra  $\{\mathcal{C}_1^+ \cap \mathcal{C}_2^+, \mathcal{C}_1^+ \cap \mathcal{C}_2^-, \mathcal{C}_1^- \cap \mathcal{C}_2^+, \mathcal{C}_1^- \cap \mathcal{C}_2^-\}$  where the individual polyhedron are:

$$\begin{aligned} \mathcal{C}_1^+ \cap \mathcal{C}_2^+ &= \{x_1 \geq 0, y_1 = x_1, x_2 \geq 0, y_2 = x_2\}, \\ \mathcal{C}_1^+ \cap \mathcal{C}_2^- &= \{x_1 \geq 0, y_1 = x_1, x_2 \leq 0, y_2 = 0\}, \\ \mathcal{C}_1^- \cap \mathcal{C}_2^+ &= \{x_1 \leq 0, y_1 = 0, x_2 \geq 0, y_2 = x_2\}, \\ \mathcal{C}_1^- \cap \mathcal{C}_2^- &= \{x_1 \leq 0, y_1 = 0, x_2 \leq 0, y_2 = 0\}. \end{aligned}$$

We note that  $\mathcal{Q}_J$  contains  $2^{|J|}$  polyhedra. We next formulate the best convex relaxation of the output after all  $n$  ReLU assignments in a layer.

### 6.3.1 Best convex relaxation

The best convex relaxation after the  $n$  ReLU assignments is given by

$$\mathcal{S}_{\text{best}} = \text{Conv}_{Q \in \mathcal{Q}_{[n]}}(Q \cap \mathcal{S}). \quad (6.4)$$

$\mathcal{S}_{\text{best}}$  considers all  $n$  assignments jointly. Computing it is practically infeasible as it involves computing  $2^n$  convex hulls each of which has exponential cost in the number of neurons  $h$  [190].

### 6.3.2 1-ReLU

We now describe the prior convex relaxation [69] through triangles (here called 1-ReLU) that handles the  $n$  ReLU assignments *separately*. Here, the input to the  $i$ -th assignment  $y_i := \text{ReLU}(x_i)$  is the polyhedron  $P_{1\text{-ReLU}} \supseteq \mathcal{S}$  where for each  $x_i \in \mathcal{X}$ ,  $P_{1\text{-ReLU}, i}$  contains only an interval constraint  $[l_i, u_i]$  that bounds  $x_i$ , that is,  $l_i \leq x_i \leq u_i$ . Here, the interval bounds are simply obtained from the bounding box  $\mathcal{B}$  of  $\mathcal{S}$ . The output of this method after  $n$  assignments is

$$\mathcal{S}_{1\text{-ReLU}} = \mathcal{S} \cap \bigcap_{i=1}^n \text{Conv}(P_{1\text{-ReLU}, i} \cap \mathcal{C}_i^+, P_{1\text{-ReLU}, i} \cap \mathcal{C}_i^-). \quad (6.5)$$

The projection of  $\text{Conv}(P_{1\text{-ReLU},i} \cap \mathcal{C}_i^+, P_{1\text{-ReLU},i} \cap \mathcal{C}_i^-)$  onto the  $x_i y_i$ -plane is a triangle minimizing the area as shown in Fig. 5.4 (a) and is the optimal convex relaxation in this plane. However, because the input polyhedron  $P_{1\text{-ReLU}}$  is a hyperrectangle (when projected to  $\mathcal{X}$ ), it does not capture relational constraints between different  $x_i$ 's in  $\mathcal{X}$  (meaning it typically has to substantially over-approximate the set  $\mathcal{S}$ ). Thus, as expected, the computed result  $\mathcal{S}_{1\text{-ReLU}}$  of the 1-ReLU method will incur significant imprecision when compared with the  $\mathcal{S}_{\text{best}}$  result.

### 6.3.3 $k$ -ReLU relaxations

We now describe our  $k$ -ReLU framework for computing a convex relaxation of the output of  $n$  ReLUs in one layer by considering groups of  $k$  ReLUs jointly with  $k > 1$ . For simplicity, we assume that  $n > k$  and  $k$  divides  $n$ . Let  $\mathcal{J}$  be a partition of the set of indices  $[n]$  such that each block  $\mathcal{J}_i \in \mathcal{J}$  contains exactly  $k$  indices. Let  $P_{k\text{-ReLU},i} \subseteq \mathbb{R}^h$  be a polyhedron containing interval and relational constraints over the neurons from  $\mathcal{X}$  indexed by  $\mathcal{J}_i$ . In our framework,  $P_{k\text{-ReLU},i}$  is derived via  $\mathcal{B}$  and  $\mathcal{S}$  and satisfies  $\mathcal{S} \subseteq P_{k\text{-ReLU},i}$ .

Our  $k$ -ReLU framework produces the following convex relaxation of the output:

$$\mathcal{S}_{k\text{-ReLU}} = \mathcal{S} \cap \bigcap_{i=1}^{n/k} \text{Conv}_{Q \in \mathcal{Q}_{\mathcal{J}_i}}(P_{k\text{-ReLU},i} \cap Q). \quad (6.6)$$

The result of (6.6) is the optimal convex relaxation for the output of  $n$  ReLUs for the given choice of  $\mathcal{S}$ ,  $k$ ,  $\mathcal{J}$ , and  $P_{k\text{-ReLU},i}$ .

**Theorem 6.3.1.** *For  $k > 1$  and a partition  $\mathcal{J}$  of indices, if there exists a  $\mathcal{J}_i$  for which  $P_{k\text{-ReLU},i} \subsetneq \bigcap_{u \in \mathcal{J}_i} P_{1\text{-ReLU},u}$  holds, then  $\mathcal{S}_{k\text{-ReLU}} \subsetneq \mathcal{S}_{1\text{-ReLU}}$ .*

*Proof.* Since  $P_{k\text{-ReLU},i} \subsetneq \bigcap_{u \in \mathcal{J}_i} P_{1\text{-ReLU},u}$  for  $\mathcal{J}_i$ , by monotonicity of intersection and convex hull,

$$\text{Conv}_{Q \in \mathcal{Q}_{\mathcal{J}_i}}(P_{k\text{-ReLU},i} \cap Q) \subsetneq \text{Conv}_{Q \in \mathcal{Q}_{\mathcal{J}_i}}\left(\left(\bigcap_{u \in \mathcal{J}_i} P_{1\text{-ReLU},u}\right) \cap Q\right) \quad (6.7)$$

For any  $Q \in \mathcal{Q}_{\mathcal{J}_i}$ , we have that either  $Q \subseteq \mathcal{C}_u^+$  or  $Q \subseteq \mathcal{C}_u^-$  for  $u \in \mathcal{J}_i$ . Thus, we can replace all  $Q$  on the right hand side of (6.7) with either  $\mathcal{C}_u^+$  or  $\mathcal{C}_u^-$  such that for all  $u \in \mathcal{J}_i$  both  $\mathcal{C}_u^+$  and  $\mathcal{C}_u^-$  are used at least in one substitution and obtain by monotonicity,

$$\begin{aligned} &\subseteq \text{Conv}_{u \in \mathcal{J}_i}\left(\left(\bigcap_{u \in \mathcal{J}_i} P_{1\text{-ReLU},u}\right) \cap \mathcal{C}_u^+, \left(\bigcap_{u \in \mathcal{J}_i} P_{1\text{-ReLU},u}\right) \cap \mathcal{C}_u^-\right) \\ &\subseteq \text{Conv}_{u \in \mathcal{J}_i}(P_{1\text{-ReLU},u} \cap \mathcal{C}_u^+, P_{1\text{-ReLU},u} \cap \mathcal{C}_u^-) \quad \left(\bigcap_{u \in \mathcal{J}_i} P_{1\text{-ReLU},u} \subseteq P_{1\text{-ReLU},u}\right). \end{aligned}$$

For remaining  $i$ , similarly  $\text{Conv}_{Q \in \mathcal{Q}_i}(P_{k\text{-ReLU},i} \cap Q) \subseteq \text{Conv}_{u \in \mathcal{J}_i}(P_{1\text{-ReLU},u} \cap \mathcal{C}_u^+, P_{1\text{-ReLU},u} \cap \mathcal{C}_u^-)$  holds. Since  $\subsetneq$  relation holds for at least one  $i$  and  $\subseteq$  holds for others,  $\mathcal{S}_{k\text{-ReLU}} \subsetneq \mathcal{S}_{1\text{-ReLU}}$  holds by the monotonicity of the intersection.  $\square$

Note that  $P_{1\text{-ReLU}}$  only contains interval constraints whereas  $P_{k\text{-ReLU}}$  contains both, the same interval constraints and extra relational constraints. Thus, any convex relaxation obtained using  $k\text{-ReLU}$  is typically strictly more precise than a  $1\text{-ReLU}$  one.

**PRECISE AND SCALABLE RELAXATIONS FOR LARGE  $k$**  For each  $\mathcal{J}_i$ , the optimal convex relaxation  $\mathcal{K}_i = \text{Conv}_{Q \in \mathcal{Q}_{\mathcal{J}_i}} (P_{k\text{-ReLU},i} \cap Q)$  from (6.6) requires computing the convex hull of  $2^k$  convex sets each of which has a worst-case exponential cost in terms of  $k$ . Thus, computing  $\mathcal{K}_i$  via (6.6) can become computationally expensive for large values of  $k$ . We propose an efficient relaxation  $\mathcal{K}'_i$  for each block  $\mathcal{J}_i \in \mathcal{J}$  (where  $|\mathcal{J}_i| = k$  as described earlier) based on computing relaxations for all subsets of  $\mathcal{J}_i$  that are of size  $2 \leq l < k$ . Let  $\mathcal{R}_i = \{\{j_1, \dots, j_l\} \mid j_1, \dots, j_l \in \mathcal{J}_i\}$  be the set containing all subsets of  $\mathcal{J}_i$  containing  $l$  indices. For each  $R \in \mathcal{R}_i$ , let  $P'_{l\text{-ReLU},R} \subseteq \mathbb{R}^h$  be a polyhedron containing interval and relational constraints between the neurons from  $\mathcal{X}$  indexed by  $R$  with  $\mathcal{S} \subseteq P'_{l\text{-ReLU},R}$ .

The relaxation  $\mathcal{K}'_i$  is computed by applying  $l\text{-ReLU}$   $\binom{k}{l}$  times as:

$$\mathcal{K}'_i = \bigcap_{R \in \mathcal{R}_i} \text{Conv}_{Q \in \mathcal{Q}_R} (P'_{l\text{-ReLU},R} \cap Q). \quad (6.8)$$

**Example 6.3.2.** Consider  $k = 4$  and  $l = 3$  with  $\mathcal{J}_i = \{1, 2, 3, 4\}$ , then  $\mathcal{R}_i = \{\{1, 2, 3\}, \{1, 2, 4\}, \{1, 3, 4\}, \{2, 3, 4\}\}$  contains all subsets of  $\mathcal{J}_i$  of size 3. We first compute  $P'_{3\text{-ReLU},R}$  for each  $R \in \mathcal{R}_i$ , for example,  $P'_{3\text{-ReLU},\{1,2,3\}}$  contains relational and interval constraints for the variables  $x_1, x_2, x_3$ . Our approximation  $\mathcal{K}'_i$  of the optimal  $4\text{-ReLU}$  output is computed using (6.8) by intersecting the result of  $3\text{-ReLU}$  for each  $R \in \mathcal{R}_i$ .

The layerwise convex relaxation  $\mathcal{S}'_{k\text{-ReLU}} = \mathcal{S} \cap \bigcap_{i=1}^{n/k} \mathcal{K}'_i$  via (6.8) is tighter than computing relaxation  $\mathcal{S}_{l\text{-ReLU}}$  via (6.6) with a partition  $\mathcal{J}'$  where for each block  $\mathcal{J}'_i \in \mathcal{J}'$  there exists  $\mathcal{R}_j$  corresponding to a block of  $\mathcal{J}$  such that  $\mathcal{J}'_i \in \mathcal{R}_j$  and  $P'_{l\text{-ReLU},\mathcal{J}'_i} \subseteq P_{l\text{-ReLU},\mathcal{J}'_i}$  where  $P_{l\text{-ReLU},\mathcal{J}'_i}$  is the polyhedron in (6.6) for computing  $\mathcal{S}_{l\text{-ReLU}}$ . In our instantiations, we ensure that this condition always holds for gaining precision.

#### 6.4 INSTANTIATING THE $k\text{-ReLU}$ FRAMEWORK

Our  $k\text{-ReLU}$  framework from Section 6.3 can be instantiated to produce different relaxations depending on the parameters  $\mathcal{S}, k, \mathcal{J}$ , and  $P_{k\text{-ReLU},i}$ . Fig. 6.4 shows the steps to instantiating our framework. The inputs to the framework are the convex set  $\mathcal{S}$  computed via a convex relaxation method  $M$  and the partition  $\mathcal{J}$  based on  $k$ . These inputs are first used to produce a set containing  $n/k$  polyhedra  $\{P_{k\text{-ReLU},i}\}$ . Each polyhedron  $P_{k\text{-ReLU},i}$  is then intersected with polyhedra from the set  $\mathcal{Q}_{\mathcal{J}_i}$  producing  $2^k$  polyhedra which are then combined via the convex hull (each called  $\mathcal{K}_i$ ). The  $\mathcal{K}_i$ 's are then combined with  $\mathcal{S}$  to produce the final relaxation that captures

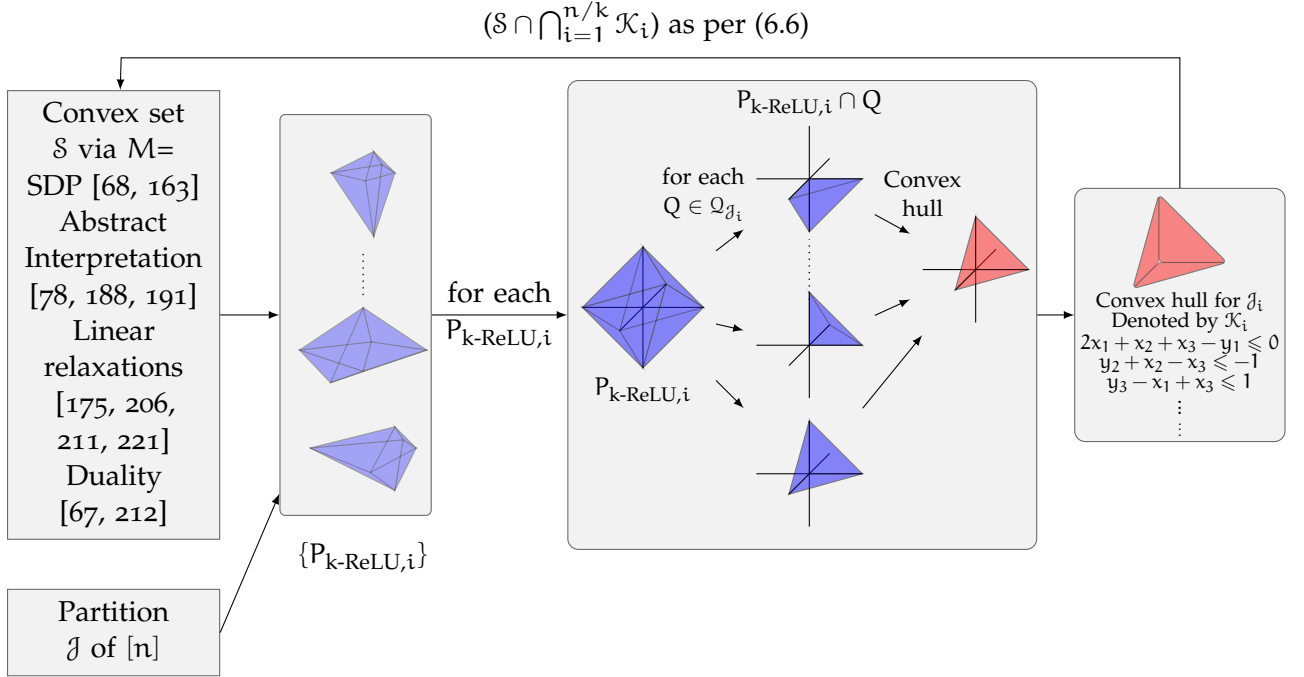


Figure 6.4: Steps to instantiating the k-ReLU framework.

the values which neurons can take after the ReLU assignments. This relaxation is tighter than that produced by applying  $M$  directly on the ReLU layer enabling precision gains.

#### 6.4.1 Computing key parameters

We next describe the choice of the parameters  $S, k, \mathcal{J}, P_{k\text{-ReLU},i}$  in our framework.

**Input convex set** Examples of a convex approximation method  $M$  for computing  $S$  include [67, 78, 163, 175, 188, 191, 206, 211, 212, 221]. In this work, we use the DeepPoly [188] relaxation for computing  $S$  which is a state-of-the-art precise and scalable certifier for neural networks.

**$k$  and partition  $\mathcal{J}$**  We use (6.6) to compute the output relaxation when  $k \in \{2, 3\}$ . For larger  $k$ , we compute the output based on (6.8). To maximize precision gain, we group those indices  $i$  together into a block where the triangle relaxation for  $y_i := \text{ReLU}(x_i)$  has the larger area in the  $x_i y_i$ -plane.

**Computing  $P_{k\text{-ReLU},i}$**  We note that for a fixed block  $\mathcal{J}_i$ , several polyhedra  $P_{k\text{-ReLU},i}$  are possible that produce convex relaxations with varying degree of precision. Ideally, one would like  $P_{k\text{-ReLU},i}$  to be the projection of  $S$  onto the variables in the set  $\mathcal{X}$  indexed by the block  $\mathcal{J}_i$ . However, computing this projection exactly is expensive and therefore we compute an overapproximation of it.

We use the method  $M$  to compute  $P_{k\text{-ReLU},i}$  by computing the upper bounds for linear relational expressions of the form  $\sum_{u=1}^k a_u \cdot x_u$  with respect to  $S$ . In our experiments, we found that setting  $a_u \in \{-1, 0, 1\}$  yields maximum precision (ex-

cept in the case where all possible  $\alpha_u$  are zero). Thus  $P_{k\text{-ReLU},i} \supseteq \mathcal{S}$  contains  $3^k - 1$  constraints which include the interval constraints for all  $x_u$ .

#### 6.4.2 Certification and refinement with $k$ -ReLU framework

We can use the constraints generated by our framework encoding the ReLU layers in the formula  $\varphi_{\text{LP}}^{(k)}$  defined in Section 6.2, which can be used for either refining the neuron bounds or for proving the property  $\psi$ .

### 6.5 EVALUATION

We implemented our refinement approach combining abstract interpretation with solvers in the form of a certifier called kPoly. kPoly runs DeepPoly analysis which is refined using MILP and  $k$ -ReLU based formulation of the ReLU layers of the network (Section 6.2). We note that in our instantiation of the  $k$ -ReLU framework, the DeepPoly domain serves as the convex relaxation method  $M$  (Fig. 6.4). Both the neuron bounds and the final certification results can be refined.

kPoly is written in Python and uses cdd [2, 4, 75] for computing convex hulls, and Gurobi [92] as the solver for refining the abstract interpretation results. We made kPoly publicly available as part of our ERAN [3] framework available at <https://github.com/eth-sri/eran>.

We evaluated kPoly for the task of robustness certification of challenging deep neural networks. We compare the speed and precision of kPoly for both complete and incomplete certification against two state-of-the-art certifiers: DeepPoly [188] and RefineZono [187]. DeepPoly has the same precision as [31, 221] whereas RefineZono refines the results of DeepZ [191] and is more precise than [191, 211, 212]. Both, DeepPoly and RefineZono are more scalable than [37, 67, 68, 163, 197, 206]. We show that kPoly is more precise than DeepPoly and RefineZono while also scaling to large networks. Our results show that kPoly achieves faster complete certification and more precise incomplete certification than prior work.

We next describe the neural networks, benchmarks and parameters used in our experiments.

**NEURAL NETWORKS** We used 7 MNIST [124], 3 CIFAR10 [118], and 1 ACAS Xu [110] based neural networks shown in Table 6.2. Our networks have fully-connected (FNNs), convolutional (CNNs), and residual architectures. All networks except ResNet are taken from the ERAN website; ResNet is taken from [https://github.com/locuslab/convex\\_adversarial](https://github.com/locuslab/convex_adversarial). Seven of the networks do not use adversarial training while the rest use different variants of it. The MNIST ConvBig network is trained with DiffAI [144], the two CIFAR10 convolutional networks are trained with PGD [136] and the residual network is trained via [212]. In the table, the MNIST FNNs are named in the format  $m \times n$ . These networks

Table 6.2: Neural network architectures and parameters used in our experiments.

Dataset	Model	Type	#Neurons	#Layers	Defense	Refine ReLU	k
MNIST	$2 \times 50$	fully-connected	110	3	None	✗	N/A
	$5 \times 100$	fully-connected	510	6	None	✓	3
	$8 \times 100$	fully-connected	810	9	None	✓	2
	$5 \times 200$	fully-connected	1 010	6	None	✓	2
	$8 \times 200$	fully-connected	1 610	9	None	✓	2
	ConvSmall	convolutional	3 604	3	None	✗	Adapt
	ConvBig	convolutional	48 064	6	DiffAI [144]	✗	5
CIFAR10	ConvSmall	convolutional	4 852	3	PGD [136]	✗	Adapt
	ConvBig	convolutional	62,464	6	PGD [136]	✗	5
	ResNet	Residual	107,496	13	Wong [212]	✗	Adapt
ACAS Xu	$6 \times 50$	fully-connected	300	6	None	✗	N/A

have  $m + 1$  layers where the first  $m$  layers have  $n$  neurons each and the last layer has 10 neurons. We note that the MNIST  $5 \times 100$  and  $8 \times 200$  are named as FFNSmall and FFNMed networks in Section 5.5 respectively. The largest network in our experiments contains  $> 100K$  neurons and has 13 layers.

**ROBUSTNESS PROPERTY** We consider the  $L_\infty$ -norm [40] based adversarial region around a correctly classified image from the test set parameterized by the radius  $\epsilon \in \mathbb{R}$ . Our goal is to certify that the network classifies all images in the adversarial region correctly.

**MACHINES** The runtimes of all experiments for the MNIST and ACAS Xu FNNs were measured on a 3.3 GHz 10 Core Intel i9-7900X Skylake CPU with a main memory of 64 GB whereas the experiments for the rest were run on a 2.6 GHz 14 core Intel Xeon CPU E5-2690 with 512 GB of main memory.

**BENCHMARKS** For each MNIST and CIFAR10 network, we selected the first 1000 images from the respective test set and filtered out incorrectly classified images. The number of correctly classified images by each network are shown in Table 6.3. We chose challenging  $\epsilon$  values for defining the adversarial region for each network. For the ACAS Xu network, we consider the property  $\phi_9$  as defined in [113]. We note that our benchmarks (e.g., the  $8 \times 200$  network with  $\epsilon = 0.015$ ) are quite challenging to handle for state-of-the-art certifiers (as we will see below).

Table 6.3: Number of certified adversarial regions and runtime of kPoly vs. DeepPoly and RefineZono.

Dataset	Model	#correct	$\epsilon$	DeepPoly [188]		RefineZono [187]		kPoly	
				certified(#)	time(s)	certified(#)	time(s)	certified(#)	time(s)
MNIST	$2 \times 50$	959	0.03	411	0.1	782	3.5	782	2.9
	$5 \times 100$	960	0.026	160	0.3	312	310	441	307
	$8 \times 100$	947	0.026	182	0.4	304	411	369	171
	$5 \times 200$	972	0.015	292	0.5	341	570	574	187
	$8 \times 200$	950	0.015	259	0.9	316	860	506	464
	ConvSmall	980	0.12	158	3	179	707	347	477
	ConvBig	929	0.3	711	21	648	285	736	40
CIFAR <sub>10</sub>	ConvSmall	630	2/255	359	4	347	716	399	86
	ConvBig	631	2/255	421	43	305	592	459	346
	ResNet	290	8/255	243	12	243	27	245	91

### 6.5.1 Complete certification

We next describe our results for the complete certification of the ACAS Xu  $6 \times 50$  and the MNIST  $2 \times 50$  network.

**ACAS XU  $6 \times 50$  NETWORK** As this network has only 5 inputs, we split the pre-condition defined by  $\phi_9$  into smaller input regions by splitting each input dimension independently. Our splitting heuristic is similar to the one used in Neurify [206] which is state-of-the-art for certifying ACAS Xu networks. We certify that the post-condition defined by  $\phi_9$  holds for each region with DeepPoly domain analysis. kPoly certifies that  $\phi_9$  holds for the network in 14 seconds. RefineZono uses the same splits with the DeepZ domain and verifies in 10 seconds. We note that both these timings are faster than Neurify which takes  $> 100$  seconds.

**MNIST  $2 \times 50$  NETWORK** For complete certification of this network, kPoly first runs DeepPoly analysis on the whole network collecting the bounds for all neurons in the network. If DeepPoly fails to certify the network, then the collected bounds are used to encode the robustness certification as a MILP instance (discussed in Section 6.2). RefineZono is based on the same approach but uses the DeepZ domain [186]. We use  $\epsilon = 0.03$  for the  $L_\infty$ -norm attack. We note that complete certification for this benchmark with RefineZono was previously reported in [187] to be slightly faster than MIPVerify [197] which is a state-of-the-art complete certifier for MNIST and CIFAR<sub>10</sub> networks.

The first row of Table 6.3 shows our results. Both kPoly and RefineZono certify the neural network to be robust on 782 regions. The average runtime of kPoly and



RefineZono is 2.9 and 3.3 seconds respectively. DeepPoly is faster than both kPoly and RefineZono but is also quite imprecise and certifies only 411 regions.

### 6.5.2 Incomplete certification

Both our works RefineZono and kPoly refine abstract interpretation results with precise solvers but with different domains. Further, RefineZono uses only 1-ReLU approximation in  $\varphi_{LP}^{(k)}$  while kPoly uses  $k > 1$ . Next, we list the parameter values for kPoly used in our experiments.

**KPOLY PARAMETERS** We refine both the DeepPoly ReLU relaxation and the certification results for the MNIST FNNs. All neurons that are input to a ReLU operation and can take positive values based on the abstract interpretation results are selected for refinement. As an optimization, we use the MILP ReLU encoding from [197] when refining the ReLU relaxation for the second ReLU layer. Thus  $k_{MILP} = 2$  for these networks and  $k_{LP} = m - k_{MILP}$ ,  $k_{AI} = 0$  where  $m$  is the number of layers. Only the certification results are refined for the rest, thus  $k_{LP} = k_{MILP} = 0$ ,  $k_{AI} = m$ .

The last column of Table 6.2 shows the value of  $k$  for all networks. We use the entry N/A for the ACAS Xu  $6 \times 50$  and the MNIST  $2 \times 50$  network as the  $k$ -ReLU framework was not used for refinement on these. The entry Adapt means that  $k$  was not fixed for all layers but computed dynamically. For the MNIST  $5 \times 100$  network, we use  $k = 3$  for encoding all ReLU layers and use  $k = 2$  for refining the remaining FNNs. For the MNIST and CIFAR10 ConvBig networks, we encode the first 3 ReLU layers with 1-ReLU while the remaining are encoded with 5-ReLU. We use  $l = 3$  in (6.8) for encoding 5-ReLU. For the remaining 3 CNNs, we encode the first ReLU layer with 1-ReLU while the remaining layers are encoded adaptively. Here, we choose a value of  $k$  for which the total number of calls to 3-ReLU is  $\leq 500$ . Next, we discuss our experimental results shown in Table 6.3.

**KPOLY VS DEEPPOLY AND REFINEZONO** Table 6.3 compares the precision in the number of adversarial regions certified and the average runtime per region in seconds for kPoly, DeepPoly, and RefineZono. We refine the certification results with kPoly and RefineZono only when DeepPoly and DeepZ fail to certify respectively. It can be seen in the table that kPoly is more precise than both DeepPoly and RefineZono on all networks. RefineZono is more precise than DeepPoly on the networks trained without adversarial training. On the  $8 \times 200$  and MNIST ConvSmall networks, kPoly certifies 506 and 347 regions respectively whereas RefineZono certifies 316 and 179 regions respectively. The precision gain with kPoly over RefineZono and DeepPoly is less on networks trained with adversarial training. kPoly certifies 25, 40, 38, and 2 regions more than DeepPoly on the last 4 CNNs in Table 6.3. kPoly is faster than RefineZono on all networks and has an average runtime of  $< 8$  minutes. We note that the runtime of kPoly is not necessarily determined by the number

of neurons but rather by the complexity of the refinement instances. In the table, the larger runtimes of kPoly are on the MNIST  $8 \times 200$  and ConvSmall networks. These are quite small compared to the CIFAR10 ResNet network where kPoly has an average runtime of only 91 seconds.

**1-RELU VS K-RELU** We consider the first 100 regions for the MNIST ConvSmall network and compare the number of regions certified by kPoly when run with k-ReLU and 1-ReLU. We note that kPoly run with 1-ReLU is equivalent to [175]. kPoly with 1-ReLU certifies 20 regions whereas with k-ReLU it certifies 35. kPoly with 1-ReLU has an average runtime of 9 seconds.

**EFFECT OF HEURISTIC FOR  $\mathcal{J}$**  We ran kPoly based on k-ReLU with random partitioning  $\mathcal{J}_r$  using the same setup as for 1-ReLU. We observed that kPoly produced worse bounds and certified 34 regions.

## 6.6 RELATED WORK

We next discuss works related to ours in Chapter 5 and 6.

### 6.6.1 *Neural Network Certification*

There is plethora of work on neural network certification mostly where the input regions can be encoded as a boxes such as  $L_\infty$ -norm based. The approaches can be broadly classified into two types: complete and incomplete.

**COMPLETE CERTIFIERS** Complete certifiers are based on MILP solvers [8, 32, 36, 49, 66, 135, 197], SMT solving [37, 69, 113, 114], Lipschitz optimization [170], and input and neuron refinement [206, 207]. In our experience, MILP solvers [8, 197] scale the most for complete certification with high dimensional inputs such as MNIST or CIFAR10 networks, while input refinement [206, 207] works the best for lower dimensional inputs such as those for ACAS Xu. In our approach for complete certification in ERAN, we use both. Our results in Section 6.5 indicate that ERAN gets state-of-the-art complete certification results. We believe that our performance can be further improved by designing new algorithms for the MILP solvers that take advantage of the particular structure of the problem instances [32, 36, 135].

**INCOMPLETE CERTIFIERS** The incomplete certifiers sacrifice the exactness of the complete certifiers to gain extra scalability. The challenge is then to design a certifier that is as precise as possible but also scales. The approaches here are based on duality [67, 212], convex relaxations [7, 31, 68, 78, 127, 163, 175, 186, 188, 199, 211, 221], and combination of relaxations with solvers [187, 206]. DeepPoly and DeepZ analysis are among the most precise and scalable certifiers. We note that the work of

[211, 212], although based on different principles, obtains the same precision and similar speed as DeepZ. Similarly, the work of [31, 221] obtains the same precision and similar speed as DeepPoly. We note that our refinement approach presented in this chapter allow us to be more precise than all competing incomplete certifiers and our GPU implementation of DeepPoly in [152] allows scaling to larger benchmarks with the precision of DeepPoly.

A complementary approach to the above is modifying the neural network to make them easier to certify [85, 90, 154, 180, 215]. We believe that a combination of this approach with ERAN can further improve the certification results.

**OTHER SPECIFICATIONS** The work of [160] considers the certification of various non-linear specifications. There has been increasing interest in certifying against adversarial regions generated by geometric transformations. The work of [158] was the first to tackle these regions; however, they did not consider linear interpolation which is often applied together with geometric transformations. Our work [188] was the first to also consider linear interpolation and produced an interval approximation of the resulting non-convex adversarial region. The work of [149] also produces interval regions. In our recent work [15], we obtain more precise polyhedral regions, making our approach more scalable and precise, and currently the state-of-the-art for geometric certification.

The authors in [5] were among the first to consider certification of ReLU based recurrent neural networks (RNNs). The work of [117] was the first to consider certification of RNNs employed in natural language processing (NLP) tasks having sigmoid and tanh activations. Their approach is based on an extension of their previous work [221]. [108] uses the simpler Interval domain for the certification of RNNs for NLP tasks. Our recent work [172] shows how the DeepPoly domain can be adapted for the robustness certification of RNNs. Our results show that our approach gets better precision and speed than [117] and is also the first one to consider the certification of the audio classifiers. We note that the recent work of [103] also considers the robustness of audio classifiers.

[217] is the first work to consider certifying robustness against adversarial patches. The authors use the Interval domain in their work. [165] and [166] consider the robustness certification of support vector machines and decision tree ensembles respectively via abstract interpretation. The work of [65] trains models robust to data poisoning attacks based on abstract interpretation.

Further works on robustness certification include those on binarized neural networks [106, 153], transformers [179], video classifiers [214], fairness of models [171], generative properties [145, 193, 216], and runtime monitoring [133].

**PROBABILISTIC GUARANTEES VIA SMOOTHING** Recently, there has been a growing interest in an orthogonal line of work to ours based on randomized smoothing. The approach is inspired by the concept of differential privacy and considers a randomized classifier. Because of the randomness, the classifier can fail

on a previously correctly classified input with a small probability. Therefore the guarantees on robustness are probabilistic here whereas our approach provides deterministic guarantees. The first work on randomized smoothing for neural networks was presented by [125]. The considered adversarial regions were intensity-based, like in our setting. Follow-up work of [53] improves the probabilistic bounds of [125]. The work of [174] used adversarial training to further improve the performance of smoothed classifiers. Randomized smoothing for geometric robustness has been recently considered in the work of [72, 129].

We note that recently, the authors in [17] consider quantitative certification of neural networks which is orthogonal to our qualitative approach.

### 6.6.2 *Constructing adversarial examples*

Another alternative line of work is that of *empirical* certification of neural networks. Here, neural network robustness is demonstrated by the lack of an empirical adversarial example using the strongest attacks within an adversarial region. As an example, [19] under-approximates the behavior of the network under  $L_\infty$ -norm based perturbation and formally defines metrics of adversarial frequency and adversarial severity to evaluate the robustness of a neural network against adversarial attack. However, no formal robustness guarantees are provided, Thus the empirical guarantees can be broken by a better attack. For example, recent work by [198] broke robustness guarantees of several existing works.

There is considerable interest in constructing adversarial examples for image classifiers. The adversarial regions here are usually based on intensity changes [34, 40, 64, 86, 120, 136, 155, 158, 173, 196], geometric [70], and 3D transformations [9]. Beyond image classifiers, there are works on crafting adversarial attacks on videos [210], speech [41, 128, 161], NLP [150], malware classification [88] and probabilistic forecasting models [62]. We refer the reader to [38, 43] for a more detailed survey on adversarial attacks.

### 6.6.3 *Adversarial training*

There is growing interest in *adversarial training* where neural networks are trained against a model of adversarial attacks. Here, a *robustness* loss is added to the normal training loss. The robustness loss is calculated as the worst-case loss in an adversarial region. It is not possible to exactly compute this loss; thus, it is either estimated with a lower bound or an upper bound. Using the lower bound leads to better empirical robustness [39, 64, 86, 89, 136] whereas the upper bound [59, 60, 87, 130, 144, 147, 148, 162, 212, 213, 220] leads to models that are relatively easier to certify. In both cases, there is a loss in the standard accuracy of the trained model. The main challenge is then to produce models that are both robust and accurate.

Interestingly, the work of [144, 147, 148] trains neural networks against adversarial attacks using abstract interpretation. The work of [148] currently produces state-of-the-art models for CIFAR10 and MNIST using our DeepZ abstraction and using a certification method similar to RefineZono. We believe that the results of [148] can be further improved by using the DeepPoly domain and the more precise kPoly certification method. We note that recent work by [11] provides theoretical results on the existence of a neural network that can be certified with abstract interpretation to the same degree as a "normally" trained network with exact methods.

Beyond robustness, the work of [73, 132] trains the network so that it satisfies a logical property.

## 6.7 DISCUSSION

We presented a refinement approach combining solver-based precise methods with abstract interpretation for neural network certification. We designed a novel parametric framework k-ReLU for obtaining scalable convex relaxations that are more precise than those produced by the single neuron triangle convex relaxation. k-ReLU is generic and can be instantiated with existing convex relaxation methods. The key idea of k-ReLU is to consider multiple ReLUs *jointly*. We presented our state-of-the-art certifier kPoly, integrated into ERAN, which combines the DeepPoly domain, MILP, the k-ReLU framework, and input refinement for achieving state-of-the-art complete and incomplete certification. Our results are beyond the reach of existing certifiers.

We note that our k-ReLU framework is more general and can be extended for computing more precise convex relaxations of other non-linearities commonly applied in neural networks such as sigmoid, tanh, and maxpool. This is because the existing approximations of these non-linearities [188, 221] are also single neuron based and would benefit from considering multiple neurons jointly. For example, more precise sigmoid approximations than DeepPoly (Section 5.3) for the sigmoid assignments  $y_1 := \text{sigmoid}(x_1)$  and  $y_2 := \text{sigmoid}(x_2)$  with  $l_{x_1}, l_{x_2} < 0$  and  $u_{x_1}, u_{x_2} > 0$  can be produced: compute the DeepPoly approximations for both assignments by considering the input neuron values to be in the intervals  $[l_{x_i}, 0]$  and  $[0, u_{x_i}]$  with  $i \in \{1, 2\}$ . The resulting approximations can then be combined using the convex hull operation.

Similarly, our overall refinement approach can also be extended beyond the neural network architectures presented in this chapter. We believe that our concepts can be leveraged in the future to design state-of-the-art certifiers for handling the certification of networks from other applications and against richer specifications.



---

## CONCLUSION AND FUTURE WORK

---

In this dissertation, we presented new methods for enabling automated reasoning in two practically critical problem domains: programs and deep learning models. Our approach is based on the framework of numerical abstract interpretation and involves designing specialized algorithms that exploit the structure of problem instances arising during the analysis. Our key contribution for numerical program analysis is the development of a new theory of online decomposition, which is based on the common observation that program transformations only affect a subset of the program variables. Our theory is quite general and can be used for speeding up all existing subpolyhedra domains, often by orders of magnitude, without any precision loss. In a second step, we leveraged data-driven machine learning for obtaining heuristics that improve the speed of numerical program analysis without sacrificing too much precision. This required establishing a new connection between the concepts in static analysis and reinforcement learning.

We adopted an inherently different approach for neural network certification, where online decomposition does not work. We designed a new abstract domain for precise and scalable analysis of neural networks containing custom convex relaxations of common non-linearities used in neural networks such as ReLU, sigmoid, and tanh. We also provided a framework for computing convex relaxations of ReLU that are more precise than prior work based on considering multiple ReLUs jointly, which was ignored previously. We developed a new combination of our relaxations with exact solvers to achieve state-of-the-art certification results.

**SYSTEMS** We have released two state-of-the-art systems based on our contribution in this dissertation: ELINA for numerical program analysis and ERAN for neural network certification. ELINA contains implementations of the popular Polyhedra, Octagon, and Zone domains and enables precise relational analysis of large real-world Linux device drivers in a few seconds while prior work often timed out or ran out of memory. ERAN is a flexible and extensible certifier and supports several application domains, specifications, neural network architectures, and both incomplete and complete certification of neural networks. It can precisely analyze large neural networks containing hundreds of thousands of neurons in a few seconds, producing certification results beyond the reach of other competing certifiers.

Next, we discuss several extensions of our work in both problem domains.

## 7.1 NUMERICAL PROGRAM ANALYSIS

Our work opens up several open problems in numerical program analysis. We list some of these below:

**ONLINE DECOMPOSITION BEYOND NUMERICAL PROGRAM ANALYSIS.** The work of [56] extends our results on the applicability of online decomposition for speeding up numerical program analysis and shows that it can be applied for decomposing all abstract domains. However, the authors do not identify conditions where the decomposition does not lose precision when extended to non-numerical domains or reduced product of numerical and non-numerical domains. We believe that this is an interesting direction for future work.

**SEMANTIC ONLINE DECOMPOSITION.** The finest partitions possible with our theory of online decomposition are sensitive to the set of constraints used to represent an abstract element. A possible direction of future work could be to investigate whether the abstract domains permit partitions based on semantic criteria independent of the particular representation of the abstract element. This can lead to a new theory and potential speedups over the methods presented in this work.

**FLOATING POINT POLYHEDRA.** The Polyhedra domain discussed in Chapter 2 abstracts a set of rational points but does not capture floating-point behavior needed for analyzing hybrid and embedded systems. Interval Polyhedra [48] can be used for implementing floating-point polyhedra domain; however, the complexity of the underlying algorithms is doubly exponential. It can be an interesting problem to investigate whether specialized algorithms can be designed to reduce some of the complexity barriers for making the floating-point Polyhedra domain practically efficient.

**DOMAIN SPECIFIC LANGUAGE (DSL) FOR DECOMPOSING NUMERICAL DOMAINS.** We provided a mechanical recipe for constructing decomposed transformers from original non-decomposed transformers in Chapter 3. However, generating an optimized implementation of decomposed transformers still requires substantial effort. We believe that this process can be automated in the future via the design of a DSL where one can specify the mathematical definition of abstract elements and transformers of a given domain. The DSL can then produce a decomposed implementation of the abstract domain.

**AUTOMATED SYNTHESIS OF NUMERICAL TRANSFORMERS.** Designing abstract transformers requires significant expertise and effort. We note that the transform-



ers of numerical domains satisfy certain mathematical properties [35] of soundness and precision which can be leveraged to automatically synthesize sound-by-construction abstract transformers. This combined with the DSL can enable automated generation of fast, decomposed transformers for numerical domains.

**MACHINE LEARNING FOR SYSTEMS** We believe that our approach of using machine learning for speeding up numerical analysis presented in Chapter 4 is more general and can be used to learn adaptive policies for balancing different tradeoffs in system design. Examples include tuning the degree of compartmentalization in operating systems [202] for improved performance without sacrificing system security and balancing the accuracy vs. performance tradeoff in IoT applications [26]. Another direction is to automate the learning process presented in Chapter 4 via generative models for approximations and dataset generation.

## 7.2 NEURAL NETWORK CERTIFICATION

We next discuss several directions and open problems for future research in neural network certification:

**EXTENDING ERAN.** ERAN currently handles a subset of the different dimensions of the neural network certification problem shown in Fig. 1.7. In the future, ERAN can be extended by designing custom methods to support more application domains (e.g., NLP, finance), specifications (e.g., robustness against patches, or fairness), neural network architectures (e.g., transformers, GANs). Further probabilistic abstract interpretation [58] could be applied for providing probabilistic guarantees on forecasting models [62].

**SPECIALIZED SOLVER FOR NEURAL NETWORKS.** We believe that using off-the-shelf solvers for complete certification of neural networks produces suboptimal results as these are not designed for handling the transformations in the neural networks. Better results can be obtained by designing custom heuristics that exploit the structure of transformations applied in the neural networks. For example, [135] uses graph neural networks to learn branching rules for the MILP solver for ReLU networks, which improves its scalability.

**PROOF TRANSFER.** For certain specifications, it might be possible to use the mathematical certificate of correctness obtained via the analysis for one problem instance to prove another instance. Examples include proving robustness against patches [217] where the different adversarial regions overlap and checking whether a network was tampered with [139].

**DESIGNING BETTER TRAINING METHODS.** Another direction for future investigation is of training neural networks that are both provably robust and accurate. With existing methods, there is a significant drop in network accuracy when training the network to be provably robust. This problem is particularly grave for larger networks and more complex datasets like ImageNet. We note that our custom abstractions have already been applied for training state-of-the-art provable neural networks [148]. In the future, custom training methods and better abstractions can be designed for obtaining networks that are more provable.

**DESIGNING BETTER ADVERSARIAL ATTACKS.** Attacks on neural networks provide an upper bound on their provable robustness whereas incomplete methods provide a lower bound. For larger networks, there is a substantial gap between the two bounds. Designing better attacks can reduce this gap. In the future, this direction can be explored by designing custom attacks that leverage the information from the certification method to reduce the search area for finding an attack. These counterexamples can also be combined with the training method to achieve better accuracy and robustness. Further, our relaxations can be used for producing robust adversarial examples reproducible in the real-world [63].

**DESIGNING SPECIALIZED ALGORITHMS FOR INTERPRETABILITY.** Improving network interpretability is crucial for making deep learning models more trustworthy. In this direction, one can look into designing specialized algorithms for explaining the decisions of a deep neural network that identify the set of inputs and hidden neuron patterns causing a particular decision and generate explanations [167] in the form of symbolic constraints over the inputs and hidden neurons. The symbolic constraints can be combined with network training for generating more interpretable networks.

### 7.3 FORMAL REASONING ABOUT CYBER-PHYSICAL SYSTEMS

An increasing number of cyber-physical systems these days, such as those used in autonomous driving, medical diagnosis, and robots, contain both software and machine learning components. An exciting direction of future research is to combine techniques from both program analysis and neural network certification to establish formal guarantees on the correctness of not only the individual components but the entire system [177]. For example, in the case of self-driving cars, a generative model of the environment can be learned, and then formal reasoning can be used to establish that certain safety properties are satisfied by the car with respect to the obtained model. The interactions between the components make the problem harder, and therefore specialized combinations may need to be designed.

---

## BIBLIOGRAPHY

---

- [1] ELINA: ETH Library for Numerical Analysis. <http://elina.ethz.ch>. pages 50, 54, 56, 79, 103, 137
- [2] Extended convex hull. *Computational Geometry*, 20(1):13 – 23, 2001. pages 28, 164
- [3] ERAN: ETH Robustness Analyzer for Neural Networks, 2018. pages 137, 164
- [4] pycddlib, 2018. pages 164
- [5] M. E. Akintunde, A. Kevorchian, A. Lomuscio, and E. Pirovano. Verification of rnn-based neural agent-environment systems. In *Proc. AAAI Conference on Artificial Intelligence (AAAI)*, pages 6006–6013, 2019. pages 169
- [6] F. Amato, A. López, E. M. Peña-Méndez, P. Vaňhara, A. Hampl, and J. Havel. Artificial neural networks in medical diagnosis. *Journal of Applied Biomedicine*, 11(2):47 – 58, 2013. pages 11, 113
- [7] G. Anderson, S. Pailoor, I. Dillig, and S. Chaudhuri. Optimization and abstraction: A synergistic approach for analyzing neural network robustness. In *Proc. Programming Language Design and Implementation (PLDI)*, page 731–744, 2019. pages 13, 106, 107, 168
- [8] R. Anderson, J. Huchette, C. Tjandraatmadja, and J. P. Vielma. Strong mixed-integer programming formulations for trained neural networks. In *Proc. Integer Programming and Combinatorial Optimization (IPCO)*, volume 11480 of *Lecture Notes in Computer Science*, pages 27–42, 2019. pages 13, 114, 148, 149, 158, 168
- [9] A. Athalye, L. Engstrom, A. Ilyas, and K. Kwok. Synthesizing robust adversarial examples. In J. G. Dy and A. Krause, editors, *Proc. International Conference on Machine Learning (ICML)*, volume 80 of *Proceedings of Machine Learning Research*, pages 284–293, 2018. pages 170
- [10] D. Avis. *A Revised Implementation of the Reverse Search Vertex Enumeration Algorithm*, pages 177–198. 2000. pages 28, 86
- [11] M. Baader, M. Mirman, and M. T. Vechev. Universal approximation with certified networks. In *Proc. International Conference on Learning Representations (ICLR)*, 2020. pages 171

- [12] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Sci. Comput. Program.*, 72(1-2):3–21, 2008. pages 2, 25, 28, 45, 50, 79
- [13] R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. Precise widening operators for convex polyhedra. *Science of Computer Programming*, 58(1):28 – 56, 2005. pages 25, 76, 100
- [14] M. Balcan, T. Dick, T. Sandholm, and E. Vitercik. Learning to branch. In *Proc. International Conference on Machine Learning (ICML)*, pages 344–353, 2018. pages 108
- [15] M. Balunovic, M. Baader, G. Singh, T. Gehr, and M. T. Vechev. Certifying geometric robustness of neural networks. In *Proc. Neural Information Processing Systems (NeurIPS)*, pages 15287–15297, 2019. pages viii, 16, 137, 147, 169
- [16] M. Balunovic, P. Bielik, and M. T. Vechev. Learning to solve SMT formulas. In *Proc. Neural Information Processing Systems (NeurIPS)*, pages 10317–10328, 2018. pages 108
- [17] T. Baluta, Z. L. Chua, K. S. Meel, and P. Saxena. Scalable quantitative verification for deep neural networks. *CoRR*, abs/2002.06864, 2020. pages 170
- [18] F. Banterle and R. Giacobazzi. A fast implementation of the octagon abstract domain on graphics hardware. In *Proc. International Static Analysis Symposium (SAS)*, volume 4634 of *Lecture Notes in Computer Science*, pages 315–335. Springer, 2007. pages 87
- [19] O. Bastani, Y. Ioannou, L. Lampropoulos, D. Vytiniotis, A. V. Nori, and A. Criminisi. Measuring neural net robustness with constraints. In *Proc. Neural Information Processing Systems (NIPS)*, pages 2621–2629, 2016. pages 170
- [20] A. Becchi and E. Zaffanella. A direct encoding for nnc polyhedra. In *Proc. Computer Aided Verification (CAV)*, pages 230–248, 2018. pages 86
- [21] A. Becchi and E. Zaffanella. An efficient abstract domain for not necessarily closed polyhedra. In A. Podelski, editor, *Proc. Static Analysis Symposium (SAS)*, pages 146–165, 2018. pages 86
- [22] A. Becchi and E. Zaffanella. Revisiting polyhedral analysis for hybrid systems. In *Proc. Static Analysis Symposium (SAS)*, pages 183–202, 2019. pages 86
- [23] R. Beckett, A. Gupta, R. Mahajan, and D. Walker. Abstract interpretation of distributed network control planes. *Proc. ACM Program. Lang.*, 4(POPL), 2019. pages 1

- [24] D. Beyer. Reliable and reproducible competition results with benchexec and witnesses (report on sv-comp 2016). In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 887–904, 2016. pages 50, 79, 104
- [25] P. Bielik, V. Raychev, and M. Vechev. Learning a static analyzer from data. pages 233–253, 2017. pages 107
- [26] K. Birman, B. Hariharan, and C. D. Sa. Cloud-hosted intelligence for real-time iot applications. *ACM SIGOPS Oper. Syst. Rev.*, 53(1):7–13, 2019. pages 175
- [27] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proc. Programming Language Design and Implementation (PLDI)*, pages 196–207, 2003. pages 1, 2, 7, 21, 49, 86
- [28] M. Böhme, V. Pham, and A. Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proc. Conference on Computer and Communications Security (CCS)*, page 1032–1043, 2016. pages 108
- [29] B. Boigelot and I. Mainz. Efficient symbolic representation of convex polyhedra in high-dimensional spaces. In S. K. Lahiri and C. Wang, editors, *Proc. Automated Technology for Verification and Analysis (ATVA)*, pages 284–299, 2018. pages 86
- [30] M. Bojarski, D. D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba. End to end learning for self-driving cars. *CoRR*, abs/1604.07316, 2016. pages 11, 113
- [31] A. Boopathy, T.-W. Weng, P.-Y. Chen, S. Liu, and L. Daniel. Cnn-cert: An efficient framework for certifying robustness of convolutional neural networks. In *Proc. AAAI Conference on Artificial Intelligence (AAAI)*, pages 3240–3247, 2019. pages 13, 149, 152, 164, 168, 169
- [32] E. Botoeva, P. Kouvaros, J. Kronqvist, A. Lomuscio, and R. Misener. Efficient verification of relu-based neural networks via dependency analysis. In *Proc. AAAI Conference on Artificial Intelligence (AAAI)*, pages 3291–3299, 2020. pages 13, 114, 148, 149, 168
- [33] R. Boutonnet and N. Halbwachs. Disjunctive relational abstract interpretation for interprocedural program analysis. In *Proc. Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 136–159, 2019. pages 86
- [34] T. B. Brown, D. Mané, A. Roy, M. Abadi, and J. Gilmer. Adversarial patch. *CoRR*, abs/1712.09665, 2017. pages 170

- [35] A. Bugariu, V. Wüstholtz, M. Christakis, and P. Müller. Automatically testing implementations of numerical abstract domains. In *Proc. ACM/IEEE Automated Software Engineering (ASE)*, pages 768–778, 2018. pages 175
- [36] R. Bunel, J. Lu, I. Turkaslan, P. H. S. Torr, P. Kohli, and M. P. Kumar. Branch and bound for piecewise linear neural network verification. *J. Mach. Learn. Res.*, 21:42:1–42:39, 2020. pages 13, 114, 148, 149, 168
- [37] R. Bunel, I. Turkaslan, P. H. Torr, P. Kohli, and M. P. Kumar. A unified view of piecewise linear neural network verification. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, pages 4795–4804, 2018. pages 13, 114, 148, 149, 152, 164, 168
- [38] N. Carlini, A. Athalye, N. Papernot, W. Brendel, J. Rauber, D. Tsipras, I. J. Goodfellow, A. Madry, and A. Kurakin. On evaluating adversarial robustness. *CoRR*, abs/1902.06705, 2019. pages 170
- [39] N. Carlini, G. Katz, C. Barrett, and D. L. Dill. Ground-truth adversarial examples. *CoRR*, abs/1709.10207, 2017. pages 170
- [40] N. Carlini and D. A. Wagner. Towards evaluating the robustness of neural networks. In *Proc. IEEE Symposium on Security and Privacy (SP)*, pages 39–57, 2017. pages 12, 115, 124, 138, 165, 170
- [41] N. Carlini and D. A. Wagner. Audio adversarial examples: Targeted attacks on speech-to-text. In *Proc. IEEE Security and Privacy Workshops, (SP)*, pages 1–7. IEEE Computer Society, 2018. pages 170
- [42] K. Chae, H. Oh, K. Heo, and H. Yang. Automatically generating features for learning program analysis heuristics for c-like languages. *Proc. ACM Program. Lang.*, 1(OOPSLA):101:1–101:25, 2017. pages 89, 105, 106
- [43] A. Chakraborty, M. Alam, V. Dey, A. Chattopadhyay, and D. Mukhopadhyay. Adversarial attacks and defences: A survey. *CoRR*, abs/1810.00069, 2018. pages 170
- [44] A. Chawdhary and A. King. Compact difference bound matrices. In *Proc. Asian Symposium on Programming Languages and Systems (APLAS)*, pages 471–490, 2017. pages 87
- [45] A. Chawdhary, E. Robbins, and A. King. Simple and efficient algorithms for octagons. In *Proc. Asian Symposium on Programming Languages and Systems (APLAS)*, volume 8858 of *Lecture Notes in Computer Science*, pages 296–313. Springer, 2014. pages 87
- [46] A. Chawdhary, E. Robbins, and A. King. Incrementally closing octagons. *Formal Methods Syst. Des.*, 54(2):232–277, 2019. pages 87

- [47] J. Chen, J. Wei, Y. Feng, O. Bastani, and I. Dillig. Relational verification using reinforcement learning. *Proc. ACM Program. Lang.*, 3(OOPSLA):141:1–141:30, 2019. pages 107
- [48] L. Chen, A. Miné, and P. Cousot. A sound floating-point polyhedra abstract domain. In *Proc. Asian Symposium on Programming Languages and Systems (APLAS)*, volume 5356 of *Lecture Notes in Computer Science*, pages 3–18, 2008. pages 174
- [49] C.-H. Cheng, G. Nührenberg, and H. Ruess. Maximum resilience of artificial neural networks. In *Proc. Automated Technology for Verification and Analysis (ATVA)*, 2017. pages 13, 114, 149, 168
- [50] N. Chernikova. Algorithm for discovering the set of all the solutions of a linear programming problem. *USSR Computational Mathematics and Mathematical Physics*, 8(6):282 – 293, 1968. pages 28
- [51] R. Clarisó and J. Cortadella. The octahedron abstract domain. *Science of Computer Programming*, 64:115 – 139, 2007. pages 6, 21, 55, 58
- [52] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In E. A. Emerson and A. P. Sistla, editors, *Proc. Computer Aided Verification (CAV)*, pages 154–169, 2000. pages 108
- [53] J. M. Cohen, E. Rosenfeld, and J. Z. Kolter. Certified adversarial robustness via randomized smoothing. In K. Chaudhuri and R. Salakhutdinov, editors, *Proc. International Conference on Machine Learning (ICML)*, volume 97, pages 1310–1320, 2019. pages 170
- [54] P. Cousot and R. Cousot. *Static determination of dynamic properties of programs*, pages 106–130. 1976. pages 6, 58
- [55] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. Symposium on Principles of Programming Languages (POPL)*, page 238–252, 1977. pages 1, 154
- [56] P. Cousot, R. Giacobazzi, and F. Ranzato. A<sup>2</sup>i: Abstract<sup>2</sup> interpretation. *PACMPL*, 3(POPL):42:1–42:31, 2019. pages 9, 87, 174
- [57] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. Symposium on Principles of Programming Languages (POPL)*, pages 84–96, 1978. pages 6, 57, 58, 114, 118, 149
- [58] P. Cousot and M. Monerau. Probabilistic abstract interpretation. In *Proc. Programming Languages and Systems*, pages 169–193, 2012. pages 175

- [59] F. Croce, M. Andriushchenko, and M. Hein. Provable robustness of relu networks via maximization of linear regions. In *Proc. International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 89 of *Proceedings of Machine Learning Research*, pages 2057–2066, 2019. pages 170
- [60] F. Croce and M. Hein. Provable robustness against all adversarial  $\ell_p$ -perturbations for  $p \geq 1$ . In *Proc. International Conference on Learning Representations (ICLR)*, 2020. pages 170
- [61] C. Cummins, P. Petoumenos, A. Murray, and H. Leather. Compiler fuzzing through deep learning. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, page 95–105, 2018. pages 108
- [62] R. Dang-Nhu, G. Singh, P. Bielik, and M. Vechev. Adversarial attacks on probabilistic autoregressive forecasting models. 2020. pages viii, 13, 170, 175
- [63] D. I. Dimitrov, G. Singh, T. Gehr, and M. Vechev. Scalable inference of symbolic adversarial examples, 2020. pages viii, 176
- [64] Y. Dong, F. Liao, T. Pang, H. Su, J. Zhu, X. Hu, and J. Li. Boosting adversarial attacks with momentum. In *Proc. Computer Vision and Pattern Recognition (CVPR)*, pages 9185–9193, 2018. pages 138, 170
- [65] S. Drews, A. Albarghouthi, and L. D’Antoni. Proving data-poisoning robustness in decision trees. In *Proc. Programming Language Design and Implementation (PLDI)*, pages 1083–1097. ACM, 2020. pages 169
- [66] S. Dutta, S. Jha, S. Sankaranarayanan, and A. Tiwari. Output range analysis for deep feedforward neural networks. In *Proc. NASA Formal Methods (NFM)*, 2018. pages 13, 114, 149, 168
- [67] K. Dvijotham, R. Stanforth, S. Gowal, T. Mann, and P. Kohli. A dual approach to scalable verification of deep networks. In *Proc. Uncertainty in Artificial Intelligence (UAI)*, pages 162–171, 2018. pages 13, 114, 148, 152, 154, 163, 164, 168
- [68] K. D. Dvijotham, R. Stanforth, S. Gowal, C. Qin, S. De, and P. Kohli. Efficient neural network verification with exactness characterization. In *Proc. Uncertainty in Artificial Intelligence, UAI*, page 164, 2019. pages 13, 114, 148, 152, 163, 164, 168
- [69] R. Ehlers. Formal verification of piece-wise linear feed-forward neural networks. In *Automated Technology for Verification and Analysis (ATVA)*, 2017. pages xvi, 13, 114, 119, 120, 148, 149, 150, 160, 168



- [70] L. Engstrom, D. Tsipras, L. Schmidt, and A. Madry. A rotation and a translation suffice: Fooling cnns with simple transformations. *CoRR*, abs/1712.02779, 2017. pages 170
- [71] P. Ferrara, F. Logozzo, and M. Fähndrich. Safer unsafe code for .net. *SIGPLAN Not.*, 43:329–346, 2008. pages 58
- [72] M. Fischer, M. Baader, and M. Vechev. Certification of semantic perturbations via randomized smoothing, 2020. pages 170
- [73] M. Fischer, M. Balunovic, D. Drachsler-Cohen, T. Gehr, C. Zhang, and M. T. Vechev. DL2: training and querying neural networks with logic. In *Proc. International Conference on Machine Learning (ICLR)*, volume 97 of *Proceedings of Machine Learning Research*, pages 1931–1941, 2019. pages 171
- [74] T. Fischer and C. Krauss. Deep learning with long short-term memory networks for financial market predictions. *European Journal of Operational Research*, 270(2):654–669, 2018. pages 11
- [75] K. Fukuda and A. Prodon. Double description method revisited. In M. Deza, R. Euler, and I. Manoussakis, editors, *Combinatorics and Computer Science*, pages 91–111, 1996. pages 28, 86, 164
- [76] G. Gange, J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey. *Exploiting Sparsity in Difference-Bound Matrices*, pages 189–211. 2016. pages 87
- [77] P. Garg, D. Neider, P. Madhusudan, and D. Roth. Learning invariants using decision trees and implication counterexamples. In *Proc. Symposium on Principles of Programming Languages (POPL)*, page 499–512, 2016. pages 108
- [78] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. Vechev. AI2: Safety and robustness certification of neural networks with abstract interpretation. In *Proc. IEEE Symposium on Security and Privacy (SP)*, volume 00, pages 948–963, 2018. pages 1, 13, 14, 114, 121, 124, 137, 138, 148, 154, 155, 163, 168
- [79] A. Geramifard, T. J. Walsh, and S. Tellex. *A Tutorial on Linear Function Approximators for Dynamic Programming and Reinforcement Learning*. Now Publishers Inc., Hanover, MA, USA, 2013. pages 92
- [80] E. Gershuni, N. Amit, A. Gurfinkel, N. Narodytska, J. A. Navas, N. Rinetzky, L. Ryzhyk, and M. Sagiv. Simple and precise static analysis of untrusted linux kernel extensions. In *Proc. Programming Language Design and Implementation (PLDI)*, page 1069–1084, 2019. pages 1, 2

- [81] K. Ghorbal, E. Goubault, and S. Putot. The zonotope abstract domain `taylor1+`. In *Proc. Computer Aided Verification (CAV)*, pages 627–633, 2009. pages 14, 59, 114, 137
- [82] K. Ghorbal, E. Goubault, and S. Putot. A logical product approach to zonotope intersection. In *Proc. Computer Aided Verification (CAV)*, page 212–226, 2010. pages 59
- [83] R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *J. ACM*, 47(2):361–416, Mar. 2000. pages 59
- [84] P. Godefroid, H. Peleg, and R. Singh. Learn&fuzz: Machine learning for input fuzzing. In *Proc. Automated Software Engineering (ASE)*, page 50–59, 2017. pages 108
- [85] S. Gokulanathan, A. Feldsher, A. Malca, C. W. Barrett, and G. Katz. Simplifying neural networks with the marabou verification engine. *CoRR*, abs/1910.12396, 2019. pages 169
- [86] I. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. In *Proc. International Conference on Learning Representations (ICLR)*, 2015. pages 11, 113, 170
- [87] S. Gowal, K. Dvijotham, R. Stanforth, R. Bunel, C. Qin, J. Uesato, R. Arandjelovic, T. A. Mann, and P. Kohli. On the effectiveness of interval bound propagation for training verifiably robust models. *CoRR*, abs/1810.12715, 2018. pages 170
- [88] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. D. McDaniel. Adversarial perturbations against deep neural networks for malware classification. *CoRR*, abs/1606.04435, 2016. pages 170
- [89] S. Gu and L. Rigazio. Towards deep neural network architectures robust to adversarial examples. In *International Conference on Learning Representations (ICLR), Workshop Track Proceedings*, 2015. pages 170
- [90] D. Guidotti, F. Leofante, L. Pulina, and A. Tacchella. Verification of neural networks: Enhancing scalability through pruning. *CoRR*, abs/2003.07636, 2020. pages 17, 169
- [91] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas. The SeaHorn verification framework. In *Proc. Computer Aided Verification (CAV)*, pages 343–361, 2015. pages 10, 50, 79
- [92] Gurobi Optimization, LLC. Gurobi optimizer reference manual, 2018. pages 158, 164

- [93] N. Halbwachs, D. Merchat, and L. Gonnord. Some ways to reduce the space dimension in polyhedra computations. *Formal Methods in System Design (FMSD)*, 29(1):79–95, 2006. pages 8, 21, 85
- [94] N. Halbwachs, D. Merchat, and C. Parent-Vigouroux. Cartesian factoring of polyhedra in linear relation analysis. In *Proc. Static Analysis Symposium (SAS)*, pages 355–365, 2003. pages 8, 21, 85
- [95] J. He, M. Balunovic, N. Ambroladze, P. Tsankov, and M. T. Vechev. Learning to fuzz from symbolic execution with application to smart contracts. In *Proc. Conference on Computer and Communications Security (CCS)*, page 531–548, 2019. pages 108
- [96] J. He, G. Singh, M. Püschel, and M. Vechev. Learning fast and precise numerical analysis. In *Proc. Programming Language Design and Implementation (PLDI)*, page 1112–1127. Association for Computing Machinery, 2020. pages viii, 11, 106, 107
- [97] T. A. Henzinger and P.-H. Ho. A note on abstract interpretation strategies for hybrid automata. In *Proc. Hybrid Systems II*, pages 252–264, 1995. pages 1
- [98] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. Principles of Programming Languages (POPL)*, pages 58–70, 2002. pages 108
- [99] K. Heo, H. Oh, and H. Yang. Learning a variable-clustering strategy for Octagon from labeled data generated by a static analysis. In *Proc. Static Analysis Symposium (SAS)*, pages 237–256, 2016. pages 86, 89, 106
- [100] K. Heo, H. Oh, and H. Yang. Resource-aware program analysis via online abstraction coarsening. In *Proc. International Conference on Software Engineering (ICSE)*, 2019. pages 106, 107
- [101] J. M. Howe and A. King. Logahedra: A new weakly relational domain. In *Proc. Automated Technology for Verification and Analysis (ATVA)*, pages 306–320, 2009. pages 58
- [102] J. L. Imbert. Fourier’s elimination: Which to choose? *Principles and Practice of Constraint Programming*, pages 117–129, 1993. pages 26, 32, 68
- [103] Y. Jacoby, C. W. Barrett, and G. Katz. Verifying recurrent neural networks using invariant inference. *CoRR*, abs/2004.02462, 2020. pages 169
- [104] B. Jeannet and A. Miné. APRON: A library of numerical abstract domains for static analysis. In *Proc. Computer Aided Verification (CAV)*, volume 5643, pages 661–667, 2009. pages 25, 28, 45, 50, 79

- [105] M. Jeon, S. Jeong, and H. Oh. Precise and scalable points-to analysis via data-driven context tunneling. *Proc. ACM Program. Lang.*, 2(OOPSLA):140:1–140:29, 2018. pages 106, 107
- [106] K. Jia and M. Rinard. Efficient exact verification of binarized neural networks. *CoRR*, abs/2005.03597, 2020. pages 169
- [107] K. Jia and M. Rinard. Exploiting verified neural networks via floating point numerical error. *CoRR*, abs/2003.03021, 2020. pages 17, 137
- [108] R. Jia, A. Raghunathan, K. Göksel, and P. Liang. Certified robustness to adversarial word substitutions. In K. Inui, J. Jiang, V. Ng, and X. Wan, editors, *Proc. Empirical Methods in Natural Language Processing (EMNLP)*, pages 4127–4140, 2019. pages 148, 169
- [109] J.-H. Jourdan. Sparsity preserving algorithms for octagons. *Electronic Notes in Theoretical Computer Science*, 331:57 – 70, 2017. Workshop on Numerical and Symbolic Abstract Domains (NSAD). pages 87
- [110] K. D. Julian, M. J. Kochenderfer, and M. P. Owen. Deep neural network compression for aircraft collision avoidance systems. *CoRR*, abs/1810.04240, 2018. pages 15, 164
- [111] J. Julien Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Static analysis by abstract interpretation of embedded critical software. *SIGSOFT Softw. Eng. Notes*, 36(1):1–8, 2011. pages 1
- [112] M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976. pages 58
- [113] G. Katz, C. W. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. In *Proc. International Conference on Computer Aided Verification (CAV)*, pages 97–117, 2017. pages 13, 114, 148, 149, 165, 168
- [114] G. Katz, D. A. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljić, D. L. Dill, M. J. Kochenderfer, and C. Barrett. The marabou framework for verification and analysis of deep neural networks. In *Proc. Computer Aided Verification (CAV)*, pages 443–452, 2019. pages 13, 114, 148, 149, 168
- [115] E. B. Khalil, P. L. Bodic, L. Song, G. L. Nemhauser, and B. Dilkina. Learning to branch in mixed integer programming. In *Proc. AAAI Conference on Artificial Intelligence (AAAI)*, page 724–731, 2016. pages 108

- [116] E. B. Khalil, H. Dai, Y. Zhang, B. Dilkina, and L. Song. Learning combinatorial optimization algorithms over graphs. In *Proc. Neural Information Processing Systems (NIPS)*, pages 6348–6358, 2017. pages 108
- [117] C. Ko, Z. Lyu, L. Weng, L. Daniel, N. Wong, and D. Lin. POPQORN: quantifying robustness of recurrent neural networks. In *Proc. International Conference on Machine Learning (ICML)*, volume 97 of *Proceedings of Machine Learning Research*, pages 3468–3477, 2019. pages 169
- [118] A. Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009. pages 138, 164
- [119] S. Kulkarni, R. Mangal, X. Zhang, and M. Naik. Accelerating program analyses by cross-program training. In *Proc. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 359–377, 2016. pages 107
- [120] A. Kurakin, I. J. Goodfellow, and S. Bengio. Adversarial examples in the physical world. In *Proc. International Conference on Learning Representations (ICLR)*. OpenReview.net, 2017. pages 170
- [121] M. G. Lagoudakis and M. L. Littman. Learning to select branching rules in the dpll procedure for satisfiability. *Electronic Notes in Discrete Mathematics*, 9:344 – 359, 2001. pages 108
- [122] V. Laviro and F. Logozzo. Subpolyhedra: A (more) scalable approach to infer linear inequalities. In *Proc. Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 5403, pages 229–244, 2009. pages 6, 21
- [123] H. Le Verge. A note on Chernikova’s algorithm. Technical Report 635, IRISA, 1992. pages 28
- [124] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. In *Proc. of the IEEE*, pages 2278–2324, 1998. pages 115, 138, 164
- [125] M. Lécuyer, V. Atlidakis, R. Geambasu, D. Hsu, and S. Jana. Certified robustness to adversarial examples with differential privacy. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 656–672, 2019. pages 170
- [126] G. Lederman, M. N. Rabe, S. Seshia, and E. A. Lee. Learning heuristics for quantified boolean formulas through reinforcement learning. In *Proc. International Conference on Learning Representations (ICLR)*, 2020. pages 108
- [127] J. Li, J. Liu, P. Yang, L. Chen, X. Huang, and L. Zhang. Analyzing deep neural networks with symbolic propagation: Towards higher precision and faster verification. In *Proc. Static Analysis Symposium (SAS)*, volume 11822 of *Lecture Notes in Computer Science*, pages 296–319, 2019. pages 13, 168

- [128] J. Li, S. Qu, X. Li, J. Szurley, J. Z. Kolter, and F. Metze. Adversarial music: Real world audio adversary against wake-word detection system. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, pages 11908–11918, 2019. pages 170
- [129] L. Li, M. Weber, X. Xu, L. Rimanic, T. Xie, C. Zhang, and B. Li. Provable robust learning based on transformation-specific smoothing. *CoRR*, abs/2002.12398, 2020. pages 170
- [130] L. Li, Z. Zhong, B. Li, and T. Xie. Robustra: Training provable robust neural networks over reference adversarial space. In *Proc. International Joint Conference on Artificial Intelligence (IJCAI)*, pages 4711–4717, 2019. pages 170
- [131] P. Liang, O. Tripp, and M. Naik. Learning minimal abstractions. In *Proc. Symposium on Principles of Programming Languages (POPL)*, pages 31–42, 2011. pages 89, 106, 107
- [132] X. Lin, H. Zhu, R. Samanta, and S. Jagannathan. ART: abstraction refinement-guided training for provably correct neural networks. *CoRR*, abs/1907.10662, 2019. pages 171
- [133] J. Liu, L. Chen, A. Miné, and J. Wang. Input validation for neural networks via runtime local robustness verification. *CoRR*, abs/2002.03339, 2020. pages 17, 169
- [134] F. Logozzo and M. Fähndrich. Pentagons: A weakly relational abstract domain for the efficient validation of array accesses. In *Proc. Symposium on Applied Computing*, pages 184–188, 2008. pages 1, 2, 6, 7, 24, 58
- [135] J. Lu and M. P. Kumar. Neural network branching for neural network verification. In *Proc. International Conference on Learning Representations (ICLR)*, 2020. pages 5, 13, 108, 114, 148, 149, 168, 175
- [136] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu. Towards deep learning models resistant to adversarial attacks. In *Proc. International Conference on Learning Representations (ICLR)*, 2018. pages 164, 165, 170
- [137] A. Maréchal, D. Monniaux, and M. Périn. Scalable minimizing-operators on polyhedra via parametric linear programming. In *Proc. Static Analysis Symposium (SAS)*, pages 212–231, 2017. pages 49, 86
- [138] A. Maréchal and M. Périn. Efficient elimination of redundancies in polyhedra by raytracing. In *Proc. Verification, Model Checking, and Abstract Interpretation, (VMCAI)*, pages 367–385, 2017. pages 49, 86
- [139] E. L. Merrer and G. Tredan. Tampernn: Efficient tampering detection of deployed neural nets, 2019. pages 175

- [140] A. Miné. A new numerical abstract domain based on difference-bound matrices. In *Proc. Programs As Data Objects (PADO)*, pages 155–172, 2001. pages 6, 24, 55, 57, 58, 76, 83
- [141] A. Miné. Relational abstract domains for the detection of floating-point runtime errors. In *Proc. European Symposium on Programming (ESOP)*, pages 3–17, 2004. pages 26, 133, 134
- [142] A. Miné. The octagon abstract domain. *Higher Order and Symbolic Computation*, 19(1):31–100, 2006. pages 6, 24, 55, 57, 58, 76
- [143] A. Miné, E. Rodriguez-Carbonell, and A. Simon. Speeding up polyhedral analysis by identifying common constraints. *Electronic Notes in Theoretical Computer Science*, 267(1):127 – 138, 2010. pages 57, 85
- [144] M. Mirman, T. Gehr, and M. Vechev. Differentiable abstract interpretation for provably robust neural networks. In *Proc. International Conference on Machine Learning (ICML)*, pages 3575–3583, 2018. pages 116, 122, 138, 164, 165, 170, 171
- [145] M. Mirman, T. Gehr, and M. Vechev. Robustness certification of generative models, 2020. pages 169
- [146] M. Mirman, G. Singh, and M. Vechev. A provable defense for deep residual networks, 2019. pages viii
- [147] M. Mirman, G. Singh, and M. T. Vechev. A provable defense for deep residual networks. *CoRR*, abs/1903.12519, 2019. pages 170, 171
- [148] M. V. Mislav Balunovic. Adversarial training and provable defenses: Bridging the gap. In *Proc. International Conference on Learning Representations (ICLR)*, 2020. pages 5, 17, 116, 122, 170, 171, 176
- [149] J. Mohapatra, T. Weng, P. Chen, S. Liu, and L. Daniel. Towards verifying robustness of neural networks against semantic perturbations. *CoRR*, abs/1912.09533, 2019. pages 137, 148, 169
- [150] J. X. Morris, E. Lifland, J. Y. Yoo, and Y. Qi. Textattack: A framework for adversarial attacks in natural language processing, 2020. pages 170
- [151] T. S. Motzkin, H. Raiffa, G. L. Thompson, and R. M. Thrall. The double description method. In *Proc. Contributions to the theory of games, vol. 2*, pages 51–73. 1953. pages 23
- [152] C. Müller, G. Singh, M. Püschel, and M. Vechev. Neural network robustness verification on gpus, 2020. pages viii, 116, 137, 147, 169

- [153] N. Narodytska, S. P. Kasiviswanathan, L. Ryzhyk, M. Sagiv, and T. Walsh. Verifying properties of binarized deep neural networks. In *Proc. AAAI Conference on Artificial Intelligence (AAAI)*, pages 6615–6624, 2018. pages 169
- [154] N. Narodytska, H. Zhang, A. Gupta, and T. Walsh. In search for a sat-friendly binarized neural network architecture. In *Proc. International Conference on Learning Representations (ICLR)*, 2020. pages 169
- [155] A. M. Nguyen, J. Yosinski, and J. Clune. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In *Proc. IEEE Computer Vision and Pattern Recognition (CVPR)*, pages 427–436, 2015. pages 170
- [156] H. Oh, W. Lee, K. Heo, H. Yang, and K. Yi. Selective context-sensitivity guided by impact pre-analysis. In *Proc. Programming Language Design and Implementation (PLDI)*, pages 475–484, 2014. pages 2, 89, 105, 106
- [157] H. Oh, H. Yang, and K. Yi. Learning a strategy for adapting a program analysis via bayesian optimisation. In *Proc. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 572–588, 2015. pages 89, 105, 106
- [158] K. Pei, Y. Cao, J. Yang, and S. Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *Proc. Symposium on Operating Systems Principles (SOSP)*, pages 1–18, 2017. pages 169, 170
- [159] K. Pei, Y. Cao, J. Yang, and S. Jana. Towards practical verification of machine learning: The case of computer vision systems. *CoRR*, abs/1712.01785, 2017. pages 136
- [160] C. Qin, K. D. Dvijotham, B. O’Donoghue, R. Bunel, R. Stanforth, S. Gowal, J. Uesato, G. Swirszcz, and P. Kohli. Verification of non-linear specifications for neural networks. In *Proc. International Conference on Learning Representations (ICLR)*, 2019. pages 169
- [161] Y. Qin, N. Carlini, G. W. Cottrell, I. J. Goodfellow, and C. Raffel. Imperceptible, robust, and targeted adversarial examples for automatic speech recognition. In *Proc. International Conference on Machine Learning (ICML)*, volume 97 of *Proceedings of Machine Learning Research*, pages 5231–5240, 2019. pages 170
- [162] A. Raghunathan, J. Steinhardt, and P. Liang. Certified defenses against adversarial examples. In *Proc. International Conference on Machine Learning (ICML)*, 2018. pages 170



- [163] A. Raghunathan, J. Steinhardt, and P. S. Liang. Semidefinite relaxations for certifying robustness to adversarial examples. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 10877–10887. 2018. pages 13, 114, 148, 151, 152, 154, 163, 164, 168
- [164] F. Ranzato and F. Tapparo. Strong preservation as completeness in abstract interpretation. In *Proc. European Symposium on Programming (ESOP)*, pages 18–32, 2004. pages 59
- [165] F. Ranzato and M. Zanella. Robustness verification of support vector machines. In *Proc. Static Analysis Symposium (SAS)*, volume 11822 of *Lecture Notes in Computer Science*, pages 271–295, 2019. pages 169
- [166] F. Ranzato and M. Zanella. Abstract interpretation of decision tree ensemble classifiers. In *Proc. AAAI Conference on Artificial Intelligence (AAAI)*, pages 5478–5486, 2020. pages 169
- [167] M. T. Ribeiro, S. Singh, and C. Guestrin. "why should I trust you?": Explaining the predictions of any classifier. In *Proc. Knowledge Discovery and Data Mining (KDD)*, pages 1135–1144, 2016. pages 176
- [168] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953. pages 7
- [169] X. Rival and L. Mauborgne. The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst.*, 29(5), 2007. pages 116, 135, 136
- [170] W. Ruan, X. Huang, and M. Kwiatkowska. Reachability analysis of deep neural networks with provable guarantees. In *Proc. International Joint Conference on Artificial Intelligence, (IJCAI)*, 2018. pages 13, 114, 148, 149, 168
- [171] A. Ruoss, M. Balunovic, M. Fischer, and M. T. Vechev. Learning certified individually fair representations. *CoRR*, abs/2002.10312, 2020. pages 169
- [172] W. Ryou, J. Chen, M. Balunovic, G. Singh, A. M. Dan, and M. T. Vechev. Fast and effective robustness certification for recurrent neural networks. *CoRR*, abs/2005.13300, 2020. pages viii, 16, 148, 169
- [173] S. Sabour, Y. Cao, F. Faghri, and D. J. Fleet. Adversarial manipulation of deep representations. In Y. Bengio and Y. LeCun, editors, *Proc. International Conference on Learning Representations (ICLR)*, 2016. pages 170
- [174] H. Salman, J. Li, I. P. Razenshteyn, P. Zhang, H. Zhang, S. Bubeck, and G. Yang. Provably robust deep learning via adversarially trained smoothed classifiers. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, pages 11289–11300, 2019. pages 170

- [175] H. Salman, G. Yang, H. Zhang, C. Hsieh, and P. Zhang. A convex relaxation barrier to tight robustness verification of neural networks. In *Proc. Neural Information Processing Systems (NeurIPS)*, pages 9832–9842, 2019. pages 13, 15, 16, 114, 148, 149, 150, 152, 154, 163, 168
- [176] S. Sankaranarayanan, M. A. Colón, H. Sipma, and Z. Manna. Efficient strongly relational polyhedral analysis. In *Proc. Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 111–125, 2006. pages 6, 21
- [177] S. A. Seshia, S. Jha, and T. Dreossi. Semantic adversarial deep learning. *IEEE Des. Test*, 37(2):8–18, 2020. pages 176
- [178] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana. NEUZZ: efficient fuzzing with neural program smoothing. In *Proc. IEEE Symposium on Security and Privacy (S&P)*, pages 803–817, 2019. pages 108
- [179] Z. Shi, H. Zhang, K. Chang, M. Huang, and C. Hsieh. Robustness verification for transformers. In *Proc. International Conference on Learning Representations (ICLR)*, 2020. pages 148, 169
- [180] D. Shriver, D. Xu, S. G. Elbaum, and M. B. Dwyer. Refactoring neural networks for verification. *CoRR*, abs/1908.08026, 2019. pages 17, 169
- [181] X. Si, H. Dai, M. Raghothaman, M. Naik, and L. Song. Learning loop invariants for program verification. In *Proc. Neural Information Processing Systems (NeurIPS)*, pages 7751–7762, 2018. pages 108
- [182] A. Simon and A. King. Exploiting sparsity in polyhedral analysis. In *Proc. Static Analysis Symposium (SAS)*, pages 336–351, 2005. pages 85
- [183] A. Simon and A. King. The two variable per inequality abstract domain. *Higher Order Symbolic Computation (HOSC)*, 23:87–143, 2010. pages 55, 58
- [184] A. Simon, A. Venet, G. Amato, F. Scozzari, and E. Zaffanella. Efficient constraint/generator removal from double description of polyhedra. *Electronic Notes in Theoretical Computer Science*, 307:3 – 15, 2014. pages 43, 86
- [185] G. Singh, R. Ganvir, M. Püschel, and M. Vechev. Beyond the single neuron convex barrier for neural network certification. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 15098–15109. 2019. pages vii, 15, 151
- [186] G. Singh, T. Gehr, M. Mirman, M. Püschel, and M. Vechev. Fast and effective robustness certification. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, pages 10825–10836. 2018. pages vii, 13, 14, 15, 114, 122, 148, 149, 152, 166, 168

- [187] G. Singh, T. Gehr, M. Püschel, and M. Vechev. Boosting robustness certification of neural networks. In *Proc. International Conference on Learning Representations (ICLR)*, 2019. pages vii, 13, 15, 151, 152, 164, 166, 168
- [188] G. Singh, T. Gehr, M. Püschel, and M. Vechev. An abstract domain for certifying neural networks. *Proc. ACM Program. Lang.*, 3(POPL):41:1–41:30, 2019. pages vii, 13, 15, 16, 59, 116, 151, 152, 154, 163, 164, 166, 168, 169, 171
- [189] G. Singh, M. Püschel, and M. Vechev. Making numerical program analysis fast. In *Proc. Programming Language Design and Implementation (PLDI)*, pages 303–313, 2015. pages vii, 21, 55, 56, 63, 79, 81, 87, 95, 107
- [190] G. Singh, M. Püschel, and M. Vechev. Fast polyhedra abstract domain. In *Proc. Principles of Programming Languages (POPL)*, pages 46–59, 2017. pages vii, 8, 21, 55, 56, 63, 79, 95, 107, 137, 160
- [191] G. Singh, M. Püschel, and M. Vechev. A practical construction for decomposing numerical abstract domains. *Proc. ACM Program. Lang.*, 2(POPL):55:1–55:28, 2017. pages vii, 9, 56, 59, 95, 107, 121, 137, 154, 163, 164
- [192] G. Singh, M. Püschel, and M. Vechev. Fast numerical program analysis with reinforcement learning. In *Proc. Computer Aided Verification CAV*, pages 211–229, 2018. pages vii, 11, 91
- [193] M. Sotoudeh and A. V. Thakur. Computing linear restrictions of neural networks. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, pages 14132–14143, 2019. pages 169
- [194] M. Stoer and F. Wagner. A simple min-cut algorithm. *J. ACM*, 44(4):585–591, 1997. pages 96
- [195] R. S. Sutton and A. G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998. pages 89, 92, 102
- [196] P. Tabacof and E. Valle. Exploring the space of adversarial images. In *Proc. International Joint Conference on Neural Networks (IJCNN)*, pages 426–433, 2016. pages 170
- [197] V. Tjeng, K. Y. Xiao, and R. Tedrake. Evaluating robustness of neural networks with mixed integer programming. In *Proc. International Conference on Learning Representations, (ICLR)*, 2019. pages 13, 114, 148, 149, 151, 152, 154, 158, 164, 166, 167, 168
- [198] F. Tramèr, N. Carlini, W. Brendel, and A. Madry. On adaptive attacks to adversarial example defenses. *CoRR*, abs/2002.08347, 2020. pages 170

- [199] H. Tran, X. Yang, D. M. Lopez, P. Musau, L. V. Nguyen, W. Xiang, S. Bak, and T. T. Johnson. NNV: the neural network verification tool for deep neural networks and learning-enabled cyber-physical systems. *CoRR*, abs/2004.05519, 2020. pages 13, 114, 148, 152, 168
- [200] C. Urban and A. Miné. An abstract domain to infer ordinal-valued ranking functions. In *Proc. European Symposium on Programming (ESOP)*, pages 412–431, 2014. pages 2, 24
- [201] C. Urban and A. Miné. A decision tree abstract domain for proving conditional termination. In *Proc. Static Analysis Symposium (SAS)*, pages 302–318, 2014. pages 2, 24
- [202] N. Vasilakis, B. Karel, N. Roessler, N. Dautenhahn, A. DeHon, and J. M. Smith. Towards fine-grained, automated application compartmentalization. In *Proc. Programming Languages and Operating Systems (PLOS)*, PLOS'17, page 43–50, 2017. pages 175
- [203] A. Venet. Abstract cofibered domains: Application to the alias analysis of untyped programs. In R. Cousot and D. A. Schmidt, editors, *Proc. Static Analysis Symposium (SAS)*, volume 1145 of *Lecture Notes in Computer Science*, pages 366–382, 1996. pages 22
- [204] A. Venet and G. Brat. Precise and efficient static array bound checking for large embedded C programs. In *Proc. Programming Language Design and Implementation (PLDI)*, pages 231–242, 2004. pages 2, 87
- [205] A. J. Venet. The Gauge domain: Scalable analysis of linear inequality invariants. In *Proc. Computer Aided Verification (CAV)*, pages 139–154, 2012. pages 6
- [206] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana. Efficient formal safety analysis of neural networks. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, pages 6369–6379. 2018. pages 13, 114, 148, 151, 152, 154, 163, 164, 166, 168
- [207] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana. Formal security analysis of neural networks using symbolic intervals. In *Proc. USENIX Security Symposium (USENIX Security 18)*, pages 1599–1614, 2018. pages 168
- [208] C. J. C. H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8(3):279–292, 1992. pages 92
- [209] S. Wei, P. Mardziel, A. Ruef, J. S. Foster, and M. Hicks. Evaluating design tradeoffs in numeric static analysis for java. In *Proc. European Symposium on Programming (ESOP)*, pages 653–682, 2018. pages 2, 24

- [210] Z. Wei, J. Chen, X. Wei, L. Jiang, T. Chua, F. Zhou, and Y. Jiang. Heuristic black-box adversarial attacks on video recognition models. In *Proc. AAAI Conference on Artificial Intelligence (AAAI)*, pages 12338–12345, 2020. pages 170
- [211] L. Weng, H. Zhang, H. Chen, Z. Song, C.-J. Hsieh, L. Daniel, D. Boning, and I. Dhillon. Towards fast computation of certified robustness for ReLU networks. In *Proc. International Conference on Machine Learning (ICML)*, volume 80, pages 5276–5285, 2018. pages 13, 114, 137, 138, 139, 148, 149, 152, 154, 163, 164, 168, 169
- [212] E. Wong and J. Z. Kolter. Provable defenses against adversarial examples via the convex outer adversarial polytope. In *Proc. International Conference on Machine Learning (ICML)*, volume 80, pages 5283–5292, 2018. pages 13, 114, 148, 149, 154, 163, 164, 165, 168, 169, 170
- [213] E. Wong, F. R. Schmidt, J. H. Metzen, and J. Z. Kolter. Scaling provable adversarial defenses. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, page 8410–8419, 2018. pages 170
- [214] M. Wu and M. Kwiatkowska. Robustness guarantees for deep neural networks on videos. *CoRR*, abs/1907.00098, 2019. pages 169
- [215] K. Y. Xiao, V. Tjeng, N. M. M. Shafiullah, and A. Madry. Training for faster adversarial robustness verification via inducing relu stability. In *Proc. International Conference on Learning Representations (ICLR)*, 2019. pages 169
- [216] Y. Yang and M. Rinard. Correctness verification of neural networks. *CoRR*, abs/1906.01030, 2019. pages 169
- [217] P. yeh Chiang\*, R. Ni\*, A. Abdelkader, C. Zhu, C. Studor, and T. Goldstein. Certified defenses for adversarial patches. In *Proc. International Conference on Learning Representations (ICLR)*, 2020. pages 148, 169, 175
- [218] H. Yu and D. Monniaux. An efficient parametric linear programming solver and application to polyhedral projection. In *Proc. Static Analysis Symposium (SAS)*, pages 203–224, 2019. pages 49, 86
- [219] E. Zaffanella. On the efficiency of convex polyhedra. *Electronic Notes in Theoretical Computer Science*, 334:31 – 44, 2018. Seventh Workshop on Numerical and Symbolic Abstract Domains (NSAD 2017). pages 86
- [220] H. Zhang, H. Chen, C. Xiao, S. Gowal, R. Stanforth, B. Li, D. S. Boning, and C. Hsieh. Towards stable and efficient training of verifiably robust neural networks. In *Proc. International Conference on Learning Representations (ICLR)*, 2020. pages 170

- [221] H. Zhang, T.-W. Weng, P.-Y. Chen, C.-J. Hsieh, and L. Daniel. Efficient neural network robustness certification with general activation functions. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*. 2018. pages 13, 149, 152, 154, 163, 164, 168, 169, 171
- [222] H. Zhu, S. Magill, and S. Jagannathan. A data-driven CHC solver. In *Proc. Programming Language Design and Implementation (PLDI)*, page 707–721, 2018. pages 108